HUBBLE 产品手册 V3.17

Hubble 团队

2024-12-13

目录

1 快速上手 **12**
1.1 基础 SQL . . . 12
1.1.1 分类 . . . 12
1.1.2 用 dbeaver 工具连接数据库 . . . 12
1.1.3 用 ssh 工具连接数据库 . . . 12
1.1.4 查看/创建/删除数据库 . . . 12
1.1.5 创建/查看/删除表 . . . 13
1.1.6 创建/查看/删除索引 . . . 14
1.1.7 数据的增删改查 . . . 14
1.2 Hubble 常用命令 . . . 15
1.2.1 hubble cert . . . 15
1.2.2 hubble start . . . 15
1.2.3 hubble init . . . 15
1.2.4 hubble node . . . 15
1.2.5 hubble sql . . . 16
1.2.6 hubble version . . . 16

2 部署 **17**
2.1 软硬件环境配置 . . . 17
2.1.1 Linux 操作系统版本要求 . . . 17
2.1.2 开发和测试环境-推荐配置 . . . 17
2.1.3 生产环境-推荐配置 . . . 17
2.1.4 数据库管理系统一体机 . . . 17
2.1.4.1 硬件建议 . . . 18
2.1.5 网络端口要求 . . . 19
2.1.6 客户端 Web 浏览器要求 . . . 19
2.2 安装前服务配置 . . . 19
2.2.1 系统级别配置 . . . 20
2.2.1.1 关闭交换分区 . . . 20
2.2.1.2 关闭服务器的防火墙 . . . 20
2.2.1.3 关闭透明大页 . . . 20
2.2.1.4 禁用 selinux . . . 20
2.2.1.5 sysctl 配置 . . . 21
2.2.1.6 ulimit 配置 . . . 22
2.2.1.7 主机与 IP 配置 . . . 22

HUBBLE

2.2.1.8 hubble 账户与组配置 . . . 23
2.2.1.9 时间同步 . . . 23
2.2.1.10 NFS 配置 (如果有网络共享盘跳过此步) . . . 24
2.2.2 服务器之间配置 . . . 25
2.2.2.1 主机之间主机名同步 . . . 25
2.2.2.2 Hubble 数据库用户免密配置 . . . 26
2.3 Hubble 集群安装 . . . 26
2.3.1 安装 Hubble . . . 26
2.3.2 创建 Hubble 所需目录 . . . 26
2.3.3 添加 Hubble 数据库配置文件 . . . 27
2.3.4 添加普通用户启停特权 . . . 28
2.3.5 证书配置 . . . 28
2.3.6 启动服务 . . . 29
2.3.7 初始化服务 . . . 29
2.3.8 连接数据库，创建用户 . . . 29
2.3.9 测试验证 . . . 30
2.3.10 设置开机自启动 . . . 30
2.4 Hubble 单机安装 . . . 30
2.4.1 安装 Hubble . . . 30
2.4.2 创建 Hubble 所需目录 . . . 30
2.4.3 添加 Hubble 数据库配置文件配置 _1 . . . 31
2.4.4 添加 Hubble 数据库配置文件配置 _2 . . . 31
2.4.5 添加 Hubble 数据库配置文件配置 _3 . . . 32
2.4.6 添加 Hubble 用户启停特权 . . . 33
2.4.7 证书配置 . . . 33
2.4.8 启动服务 . . . 34
2.4.9 初始化服务 . . . 34
2.4.10 连接数据库，创建用户 . . . 34
2.4.11 测试验证 . . . 34
2.4.12 设置开机自启动 . . . 35

3 运维 35
3.1 数据备份与恢复 . . . 35
3.1.1 数据备份 . . . 35
3.1.1.1 备份语法 . . . 35
3.1.1.2 备份数据说明 . . . 35
3.1.1.3 全量备份示例 . . . 36
3.1.1.4 增量备份 . . . 36
3.1.1.5 指定时间戳备份 . . . 36
3.1.2 数据恢复 . . . 36
3.1.2.1 依赖对象 . . . 37
3.1.2.2 备份数据路径 . . . 37
3.1.2.3 参数说明 . . . 37
3.1.2.4 解决命名冲突 . . . 38
3.1.2.5 示例 . . . 38
3.2 集群扩容缩容 . . . 39
3.2.1 集群扩容 . . . 39

HUBBLE

3.2.1.1 新节点准备 . . . 39
3.2.1.2 目录准备 . . . 40
3.2.1.3 证书配置 . . . 40
3.2.1.4 添加 hubble 数据库配置文件 . . . 40
3.2.1.5 添加普通用户启停特权 . . . 41
3.2.1.6 启动服务 . . . 41
3.2.1.7 测试验证 . . . 41
3.2.1.8 设置开机自启动 . . . 42
3.2.2 集群缩容 . . . 42
3.2.2.1 退役单个活动节点 (安全模式) . . . 42
3.3 监控告警 . . . 44
3.3.1 管理员界面区域 . . . 44
3.3.2 管理员界面访问 . . . 45
3.3.3 仪表盘 . . . 45
3.3.3.1 整体预览 . . . 45
3.3.3.2 节点列表 . . . 46
3.3.3.3 LIVE 节点列表 . . . 46
3.3.4 监控信息 . . . 46
3.3.4.1 概况 . . . 46
3.3.4.2 事件列表 . . . 47
3.3.4.3 监控信息-概况仪表板 . . . 49
3.3.4.4 硬件仪表盘 . . . 51
3.3.4.5 运行时仪表板 . . . 55
3.3.4.6 SQL 仪表板 . . . 58
3.3.4.7 存储仪表板 . . . 60
3.3.4.8 复制仪表板 . . . 61
3.3.4.9 数据变动仪表板 . . . 63
3.3.5 数据库列表 . . . 66
3.3.5.1 表视图 . . . 66
3.3.5.2 权限视图 . . . 66
3.3.6 任务列表 . . . 66
3.3.6.1 任务细节 . . . 66
3.3.6.2 过滤结果 . . . 67
3.3.7 语句列表 . . . 67
3.3.7.1 有效期 . . . 67
3.3.7.2 按应用过滤 . . . 67
3.3.7.3 参数 . . . 68
3.3.7.4 语句明细页面 . . . 68
3.4 日常巡检 . . . 72
3.4.1 巡检意义 . . . 72
3.4.2 巡检范围 . . . 72
3.4.3 登录的网址 . . . 72
3.4.4 页面服务 . . . 72
3.4.4.1 概况 . . . 72
3.4.4.2 监控 . . . 72
3.4.4.3 语句 . . . 73
3.4.4.4 任务 . . . 73

HUBBLE

3.4.5 异常处理检测 74
3.4.5.1 节点实例查看 74
3.4.5.2 运行任务查看 74
3.4.5.3 运行 sql 查看 74
3.4.6 服务启停 75
3.4.7 服务器端 75
3.5 时区调整 75
3.5.1 SQL 中设置时区 75
3.5.2 JAVA 中设置时区 76
3.5.3 在 Dbeaver 设置时区 76
3.6 租户管理 77
3.6.1 定义 77
3.6.2 租户的作用 77
3.6.3 Hubble 租户分类 77
3.6.4 复制区域 77
3.6.5 参数 77
3.6.6 变量 77
3.6.7 示例 77
3.6.7.1 按目录分租户 77
3.6.7.2 按节点分租户 78

4 数据迁移 82
4.1 迁移概述 82
4.1.1 导入文件的存储 82
4.1.2 架构和应用程序更改 82
4.1.3 数据类型大小 82
4.2 CSV 迁移 82
4.2.1 步骤一 CSV 格式数据准备 82
4.2.2 步骤二：将数据文件放置在集群可访问到的位置 83
4.2.3 步骤三：将数据迁移到 Hubble 数据库中 83
4.2.3.1 hubble 库中表的数据迁移 83
4.2.3.2 多文件导入 83
4.2.4 配置参数 84
4.2.4.1 列定界符 84
4.2.4.2 跳过注释行 84
4.2.4.3 跳过标题行 84
4.2.4.4 空字符串 84
4.2.4.5 文件压缩格式 85
4.2.4.6 行限制数 85
4.3 Mysql 迁移 85
4.3.1 MySQL 迁移至 Hubble 85
4.3.1.1 步骤一：导出需要迁移的表的数据 85
4.3.1.2 步骤二：导出需要迁移的表的建表语句 85
4.3.1.3 步骤三：在 Hubble 数据库中建表 86
4.3.1.4 步骤四：把数据导入 Hubble 中 87
4.3.2 部分常见函数的转换 87
4.3.3 Mysql 中有自增字段的表的导入 87

HUBBLE

4.3.4 MySQL 问题总结 . . . 88
4.4 Oracle 迁移 . . . 89
4.4.1 步骤 1: 导出 dmp 文件 . . . 89
4.4.2 步骤 2: 将 dmp 文件转换为 sql 格式 . . . 89
4.4.3 步骤 3: 导出表数据 . . . 89
4.4.4 步骤 4: 配置表数据并将其转换为 CSV . . . 90
4.4.5 步骤 5:Oracle 映射到 Hubble 数据类型 . . . 91
4.4.6 步骤 6: 数据导入 . . . 92
4.5 DB2 迁移 . . . 93
4.5.1 步骤一: 导出建表语句 . . . 93
4.5.2 步骤二: 导出 csv 文件 . . . 93
4.5.3 步骤三:DB2 映射 Hubble 数据类型 . . . 94
4.5.4 步骤四: 将导出的数据文件放置于集群可访问到的位置 . . . 95
4.5.5 步骤五: 导入数据 . . . 95
4.6 导入性能最佳实践 . . . 96
4.6.1 将数据拆分为多个文件 . . . 96
4.6.2 选择一种高性能的导入格式 . . . 96
4.6.2.1 将模式与数据分开导入 . . . 97

5 开发文档 97
5.1 数据库设计 . . . 97
5.1.1 概述 . . . 97
5.1.1.1 数据库架构对象 . . . 97
5.1.1.2 控制对对象的访问 . . . 98
5.1.1.3 对象大小和缩放 . . . 98
5.1.2 创建数据库 . . . 99
5.1.2.1 准备工作 . . . 99
5.1.2.2 创建数据库 . . . 99
5.1.3 创建表 . . . 100
5.1.3.1 开始前的准备 . . . 100
5.1.3.2 创建表 . . . 100
5.1.4 创建用户定义的架构 . . . 104
5.1.4.1 准备工作 . . . 104
5.1.4.2 创建用户定义的架构 . . . 104
5.1.5 列族 . . . 105
5.1.5.1 默认行为 . . . 105
5.1.5.2 手动操作 . . . 105
5.1.6 二级索引 . . . 106
5.1.6.1 准备工作 . . . 107
5.1.6.2 创建二级索引 . . . 107
5.1.6.3 最佳实践 . . . 107
5.1.6.4 示例 . . . 108
5.1.7 计算列 . . . 108
5.1.7.1 使用计算列意义 . . . 108
5.1.7.2 说明事项 . . . 108
5.1.7.3 定义计算列 . . . 109
5.1.7.4 例子 . . . 109

HUBBLE

5.1.8 索引行的子集 . . . 115
5.1.8.1 部分索引如何工作 . . . 115
5.1.8.2 创建部分索引 . . . 115
5.1.8.3 唯一的部分索引 . . . 115
5.1.8.4 索引提示 . . . 116
5.1.8.5 限制 . . . 116
5.1.8.6 示例 . . . 116
5.1.9 倒排索引 . . . 116
5.1.9.1 GIN 索引如何工作 . . . 117
5.1.9.2 部分 GIN 索引 . . . 118
5.1.9.3 多列 GIN 索引 . . . 119
5.1.9.4 示例 . . . 119
5.1.10 更改或删除数据库中的对象 . . . 122
5.1.10.1 准备 . . . 122
5.1.10.2 改变数据库模式对象 . . . 122
5.1.10.3 删除数据库模式对象 . . . 123
5.2 读取数据 . . . 125
5.2.1 子查询 . . . 125
5.2.1.1 子查询中的数据写入 . . . 125
5.2.1.2 相关子查询 . . . 125
5.2.2 物化视图与临时视图 . . . 126
5.2.2.1 视图的作用 . . . 126
5.2.2.2 视图的工作原理 . . . 128
5.2.2.3 物化视图 . . . 131
5.2.2.4 临时视图 . . . 133
5.2.3 分页查询 . . . 134
5.2.3.1 键集分页 . . . 134
5.2.3.2 举例 . . . 134
5.2.3.3 性能比较 . . . 135
5.2.4 临时表 . . . 135
5.2.4.1 说明 . . . 135
5.2.4.2 临时模式 . . . 136
5.2.4.3 示例 . . . 136
5.2.5 截止系统时间查询 . . . 138
5.2.5.1 用于 AS OF SYSTEM TIME 减少与长时间运行的查询的冲突 . . . 138
5.2.5.2 概要 . . . 138
5.2.5.3 示例 . . . 138
5.2.6 选择数据行 . . . 141
5.2.6.1 简单选择 . . . 141
5.2.6.2 排序 . . . 142
5.2.6.3 限制查询 . . . 142
5.2.6.4 join . . . 142
5.2.7 键集分页 . . . 143
5.2.7.1 键集分页 . . . 143
5.2.7.2 示例 . . . 143
5.3 写入数据 . . . 144
5.3.1 删除数据 . . . 144

HUBBLE

5.3.1.1 准备工作 . . . 144
5.3.1.2 使用 DELETE . . . 144
5.3.2 插入数据 . . . 147
5.3.2.1 插入行 . . . 147
5.3.2.2 批量插入 . . . 149

6 参考 149
6.1 优化 . . . 149
6.1.1 开发使用规范 . . . 149
6.1.1.1 开发设计 . . . 149
6.1.1.2 SQL 规范建议 . . . 150
6.1.1.3 部署规范 . . . 152
6.1.2 索引 . . . 153
6.1.2.1 定义 . . . 153
6.1.2.2 索引工作原理 . . . 153
6.1.2.3 索引的应用 . . . 153
6.1.2.4 优缺点 . . . 153
6.1.2.5 索引列 . . . 153
6.1.2.6 索引示例 . . . 154
6.1.3 向量化引擎 . . . 156
6.1.3.1 配置方式 . . . 156
6.1.3.2 支持的数据类型 . . . 157
6.1.3.3 示例 . . . 157
6.1.4 语句性能优化 . . . 157
6.1.4.1 概述 . . . 157
6.1.4.2 sql 语句执行原则 . . . 158
6.1.4.3 使用 explain 进行语句优化 . . . 158
6.1.4.4 逻辑优化 . . . 161
6.1.5 主键 . . . 162
6.1.5.1 定义 . . . 162
6.1.5.2 目的 . . . 162
6.1.5.3 特性 . . . 162
6.1.5.4 主键选择原则 . . . 162
6.1.5.5 注意 . . . 162
6.1.5.6 主键示例 . . . 162
6.1.6 使用 EXPLAIN 进行 SQL 调优 . . . 164
6.1.6.1 问题类型 . . . 164
6.1.7 SQL 性能最佳实践 . . . 171
6.1.7.1 DML 最佳实践 . . . 171
6.1.7.2 批量插入最佳实践 . . . 172
6.1.7.3 批量删除最佳实践 . . . 172
6.1.7.4 列族 . . . 173
6.1.7.5 唯一 ID 最佳实践 . . . 173
6.1.8 函数索引 . . . 175
6.1.8.1 函数索引示例 . . . 175
6.2 安全 . . . 177
6.2.1 SQL 审计日志 . . . 177

HUBBLE

6.2.1.1 参数说明 . . . 177
6.2.1.2 操作步骤 . . . 177
6.2.2 安全证书更换 . . . 180
6.2.2.1 何时轮换证书 . . . 180
6.2.2.2 轮换客户证书 . . . 180
6.2.2.3 轮换节点证书 . . . 181
6.2.2.4 轮换 CA 证书 . . . 181
6.3 SQL . . . 183
6.3.1 SQL 语法和语言结构 . . . 183
6.3.1.1 公用表表达式 . . . 183
6.3.1.2 空值介绍 . . . 186
6.3.1.3 选择查询 . . . 191
6.3.1.4 常数介绍 . . . 196
6.3.1.5 标量表达式 . . . 198
6.3.1.6 表表达式 . . . 206
6.3.1.7 更改列 . . . 210
6.3.1.8 注释 . . . 213
6.3.1.9 schema 对象名 . . . 215
6.3.1.10 属性 . . . 217
6.3.1.11 关键字 . . . 221
6.3.2 SQL 语句 . . . 230
6.3.2.1 DCL . . . 230
6.3.2.2 DDL . . . 240
6.3.2.3 DML . . . 288
6.3.2.4 DQL . . . 298
6.3.2.5 SYSTEM . . . 308
6.3.2.6 JOBS . . . 318
6.3.3 数据类型 . . . 330
6.3.3.1 支持的类型 . . . 330
6.3.3.2 数据类型转换 . . . 331
6.3.3.3 ARRAY . . . 331
6.3.3.4 BIT . . . 334
6.3.3.5 BOOL . . . 336
6.3.3.6 BYTES . . . 337
6.3.3.7 COLLATE . . . 339
6.3.3.8 DATE . . . 341
6.3.3.9 DECIMAL . . . 343
6.3.3.10 FLOAT . . . 345
6.3.3.11 INET . . . 346
6.3.3.12 INT . . . 348
6.3.3.13 INTERVAL . . . 349
6.3.3.14 JSONB . . . 351
6.3.3.15 SERIAL . . . 356
6.3.3.16 STRING . . . 359
6.3.3.17 TIME . . . 362
6.3.3.18 TIMESTAMP/TIMESTAMPTZ . . . 363
6.3.3.19 UUID . . . 365

HUBBLE

6.3.3.20 ENUM . . . 369
6.3.4 函数 . . . 370
6.3.4.1 特殊的语法形式 . . . 370
6.3.4.2 条件运算符和类函数运算符 . . . 373
6.3.4.3 数组函数 . . . 375
6.3.4.4 比较函数 . . . 383
6.3.4.5 时间函数 . . . 384
6.3.4.6 ID 函数 . . . 389
6.3.4.7 数学类函数 . . . 389
6.3.4.8 字符串函数 . . . 397
6.3.4.9 聚合函数 . . . 404
6.3.4.10 JSON 函数 . . . 427
6.3.5 数据库信息 . . . 432
6.3.5.1 包含的数据信息 . . . 432
6.3.5.2 administrable_role_authorizations . . . 432
6.3.5.3 applicable_roles . . . 432
6.3.5.4 check_constraints . . . 432
6.3.5.5 columns . . . 433
6.3.5.6 column_privileges . . . 433
6.3.5.7 constraint_column_usage . . . 434
6.3.5.8 enabled_roles . . . 434
6.3.5.9 key_column_usage . . . 434
6.3.5.10 referential_constraints . . . 434
6.3.5.11 role_table_grants . . . 435
6.3.5.12 schema_privileges . . . 435
6.3.5.13 schemata . . . 435
6.3.5.14 sequences . . . 436
6.3.5.15 statistics . . . 436
6.3.5.16 table_constraints . . . 436
6.3.5.17 table_privileges . . . 437
6.3.5.18 tables . . . 437
6.3.5.19 user_privileges . . . 437
6.3.5.20 views . . . 438
6.3.6 事务 . . . 438
6.3.6.1 事务隔离级别 . . . 438
6.3.6.2 事务开启、提交与回滚 . . . 438
6.3.6.3 事务操作 . . . 438
6.3.6.4 事务优先级 . . . 438
6.3.7 约束 . . . 439
6.3.7.1 支持的约束 . . . 439
6.3.7.2 约束条件的使用 . . . 439
6.3.7.3 创建主键时定义约束的名称 . . . 440
6.3.7.4 列出约束条件 . . . 440
6.3.7.5 移除约束条件 . . . 440
6.3.7.6 改变约束条件 . . . 441
6.3.8 窗口函数 . . . 441
6.3.8.1 简述 . . . 441

HUBBLE

6.3.8.2 窗口的定义 . . . 441
6.3.8.3 语法及其参数介绍 . . . 441
6.3.8.4 说明 . . . 442
6.3.8.5 作用 . . . 442
6.3.8.6 窗口函数机制 . . . 442
6.3.8.7 常见示例 . . . 443
6.3.9 分区表 . . . 446
6.3.9.1 分区的分类 . . . 446
6.3.9.2 分区的查询 . . . 447
6.3.9.3 分区表举例 . . . 448
6.3.10 视图 . . . 449
6.3.10.1 创建视图 . . . 450
6.3.10.2 列表视图 . . . 450
6.3.10.3 查询视图 . . . 450
6.3.10.4 重命名视图 . . . 451
6.3.10.5 删除视图 . . . 451
6.3.11 会话 . . . 451
6.3.11.1 会话变量 . . . 451
6.3.11.2 sql 语句 . . . 455
6.3.12 名称解析 . . . 455
6.3.12.1 命名层次结构 . . . 456
6.3.12.2 从以前版本的数据库迁移命名空间 . . . 456
6.3.12.3 名称解析的工作原理 . . . 456
6.3.12.4 名称解析参数 . . . 456
6.4 工具 . . . 457
6.4.1 dbeaver 连接数据库 . . . 457
6.4.1.1 下载 . . . 457
6.4.1.2 安装步骤 . . . 457
6.4.1.3 工具连接案例 . . . 457

7 故障诊断 463
7.1 诊断报告 . . . 463
7.1.1 分享的内容 . . . 463
7.1.2 选择退出诊断报告 . . . 463
7.1.2.1 在集群初始化时 . . . 463
7.1.2.2 集群初始化后 . . . 464
7.1.3 检查诊断报告的状态 . . . 464
7.2 慢 sql 设置分析 . . . 464
7.2.1 识别慢 SQL . . . 464
7.2.2 设置与查看慢 sql 方法 . . . 464
7.2.2.1 方式 1 . . . 464
7.2.2.2 方式 2 . . . 465

8 常见问题 FAQ 465
8.1 集群常见问题 . . . 465
8.1.1 节点启动常见问题 . . . 465
8.1.1.1 节点无法启动报 `500ms` . . . 465

HUBBLE

8.1.1.2 节点宕机报 `open file descriptor limit` . . . 465
8.1.1.3 节点启动报 `bind: address already in use` . . . 466
8.1.1.4 节点加入到现有的 Hubble 集群 `Store directory already exists` . . . 466
8.2 sql 常见问题 . . . 466
8.2.1 Hubble 是否支持 SELECT FOR UPDATE . . . 466
8.2.2 Hubble 支持的字符集类型 . . . 466
8.2.3 Hubble 如何定位历史数据 . . . 466
8.2.4 sql 优化 . . . 466
8.2.5 多个查询同时进行，防止死锁 . . . 466
8.2.6 文件多次导入失败 . . . 467
8.2.7 Hubble 数据库支持 JSON 或 Protobuf 数据类型吗? . . . 467
8.2.8 可以使用 Hubble 作为键值存储吗? . . . 467
8.2.9 如何将数据批量插入 Hubble? . . . 467
8.2.10 如何将最后一个 ID/SERIAL 值插入到表中? . . . 468
8.2.11 什么是事务争用? . . . 468
8.3 数据库常见问题 . . . 468
8.3.1 选择 Hubble . . . 468
8.3.1.1 什么是 Hubble 数据库? . . . 468
8.3.1.2 什么时候 Hubble 数据库是一个好的选择? . . . 468
8.3.2 关于 Hubble 数据库 . . . 468
8.3.2.1 Hubble 数据库是如何扩展的? . . . 468
8.3.2.2 Hubble 数据库如何在发生故障后继续提供服务? . . . 469
8.3.2.3 Hubble 数据库的一致性如何? . . . 469
8.3.2.4 Hubble 数据库如何既具有高可用性又具有强一致性? . . . 469
8.3.2.5 Hubble 支持分布式事务吗? . . . 469
8.3.2.6 为什么要用 Hubble 数据库 sql? . . . 470
8.3.2.7 Hubble 中的事务是否保证 ACID 语义? . . . 470
8.3.2.8 Hubble 数据库需要原子钟来同步时间吗? . . . 470
8.3.2.9 我可以使用哪些语言来连接数据库? . . . 470
8.3.2.10 为什么 Hubble 使用 PostgreSQL 协议而不是 MySQL 协议? . . . 470
8.3.2.11 Hubble 数据库的安全模型是什么? . . . 470
8.3.3 数据库的比较 . . . 471
8.3.3.1 Hubble 与 MySQL 或 PostgreSQL 相比如何? . . . 471
8.3.3.2 Hubble 数据库与 HBase、MongoDB 相比如何? . . . 471

9 版本发布 471
9.1 数据库版本 . . . 471
9.1.1 v3.17 . . . 471
9.1.1.1 BUG 修复 . . . 471
9.1.1.2 性能 . . . 471
9.1.1.3 函数 . . . 471
9.1.1.4 sql 变化 . . . 473

HUBBLE

1 快速上手

1.1 基础 SQL

成功部署 Hubble 集群之后，便可以在 Hubble 中执行 SQL 语句了。因 Hubble 支持 PostgreSQL 协议和大多数 PostgreSQL 语法，这意味着在 PostgreSQL 上构建的现有应用程序通常可以迁移到 Hubble，而无需更改应用程序代码。

SQL 是一门声明性语言，它是数据库用户与数据库交互的方式。它像是一种自然语言，好像在用英语与数据库进行对话，本文档介绍基本的 SQL 操作。

1.1.1 分类

SQL 语言通常按照功能划分成以下的 4 个部分：

- DDL (Data Definition Language)：数据定义语言，用来定义数据库对象，包括库、表、视图和索引等。
- DML (Data Manipulation Language)：数据操作语言，用来操作和业务相关的记录。
- DQL (Data Query Language)：数据查询语言，用来查询经过条件筛选的记录。
- DCL (Data Control Language)：数据控制语言，用来定义访问权限和安全级别。

常用的 DDL 功能是对象（如表、索引等）的创建、属性修改和删除，对应的命令分别是 `CREATE`、`ALTER`和 `DROP`。

1.1.2 用 dbeaver 工具连接数据库

参考dbeaver 连接数据库后，即可使用

1.1.3 用 ssh 工具连接数据库

> 需要注意 certs-dir 的地址和--host 的端口信息

- 连接数据库的 sh 语句

```
[root@hubble01 ~]# hubble sql --certs-dir=/var/lib/hubbletp/certs --host=
    ↪ hubble01:35432 --user hubble
```

```
# Welcome to the HubbleDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: HubbleDB v3.10.2-dirty (same version as client)
# Cluster ID: fb578fa1-b32c-4c71-bf5b-a4ee6f2ab28c
#
# Enter \? for a brief introduction.
#
hubble@hubble01:35432/defaultdb>
```

1.1.4 查看/创建/删除数据库

查看系统中所有数据库：

```
show databases;
```

要创建一个名为 hubble_db 的数据库，可使用以下语句：

```
create database  hubble_db;
```

使用SHOW TABLES语句查看当前数据库中的所有表：

```
show tables from hubble_db;
```

使用DROP DATABASE语句删除数据库：

```
drop database hubble_db;
```

切换到目标库

```
use hubble_db;
```

1.1.5 创建/查看/删除表

CREATE TABLE语句建表：

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );
```

> 主键的创建必须在 create table 语句中，如果建表不指定主键的话，hubble 数据库在建表时候自带 rowid 默认作为主键。

使用COMMENT给表和字段添加注释：

```
comment on table cust_info IS '客户信息表';
comment on column cust_info.cust_no is '客户号';
comment on column cust_info.cust_name is '客户姓名';
comment on column cust_info.cust_card_no is '客户身份证号';
comment on column cust_info.cust_phoneno is '客户手机号';
comment on column cust_info.cust_address is '客户所在地';
comment on column cust_info.cust_type is '客户授信类型';
```

查看表的注释

```
show tables with  comment;
```

查看字段的注释

```
show columns from cust_info with  comment;
```

使用SHOW CREATE语句查看建表语句:

```
show create table cust_info;
```

使用DROP TABLE语句删除表:

```
drop table cust_info;
```

1.1.6 创建/查看/删除索引

使用CREATE INDEX语句创建索引:

```
create index custbak_index on cust_info(cust_name);
```

使用SHOW INDEX语句查看表内所有索引:

```
show index from cust_info;
```

使用DROP INDEX语句来删除索引:

```
drop index custbak_index ;
```

1.1.7 数据的增删改查

使用INSERT INTO向表内插入数据:

```
insert into cust_info values('14435550','王吉','12022519960321531X',15122511874,
    ↪ '天津武清','抵押');
insert into cust_info values('14435551','张贺','431256197306265320',15534343555,
    ↪ '山西临汾','质押');
insert into cust_info values('14435552','刘明','371452199303034312',18967756743,
    ↪ '陕西延安','信用');
insert into cust_info values('14435553','李华','52112119860621421X',15833355455,
    ↪ '湖北武汉','抵押');
insert into cust_info values('14435554','郑青','213456199102275341',13054546567,
    ↪ '江西南昌','质押');
```

使用UPDATE语句修改表数据:

```
update cust_info set cust_phoneno='13565466894' where cust_no='14435554';
```

使用DELETE语句删除部分数据:

```
delete from cust_info where cust_no='14435554';
```

限制查询数据

```
select * from cust_info limit 3;
```

```
 cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
    ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------
    ↪
```

```
  14435550 | 王吉       | 12022519960321531X |  15122511874 | 天津武清      | 抵押
  14435551 | 张贺       | 431256197306265320 |  15534343555 | 山西临汾      | 质押
  14435552 | 刘明       | 371452199303034312 |  18967756743 | 陕西延安      | 信用
```

1.2 Hubble 常用命令

本页介绍用于配置、启动和管理 Hubble 集群的命令。

1.2.1 hubble cert

- 创建本机证书

```
hubble cert create-ca --certs-dir=certs --ca-key=mysafedirectory/ca.key
```

- 创建其他服务器证书

```
hubble cert create-node hubble02 192.168.1.12 --certs-dir=certs --ca-key=
    ↪ mysafedirectory/ca.key --overwrite
```

1.2.2 hubble start

- 启动一个节点服务

```
hubble start --locality=country=cn,region=ch-beijin,datacenter=tianyun,rack=1,
    ↪ node=1 --certs-dir=certs --listen-addr=hubble01:15432 --advertise-host=
    ↪ hubble01 --join=hubble01:15432,hubble01:15433,hubble01:15434 --cache=30GiB
    ↪  --max-sql-memory=10GiB --store=path=/data1/hubbledir/hubble1,size=350GiB
    ↪ --http-addr=0.0.0.0:48080  --external-io-dir=/data_shares
```

- hubble start查看参数帮助

```
hubble start --help
```

1.2.3 hubble init

- 集群初始化

```
hubble init --certs-dir=/var/lib/hubble/certs --host=hubble01:15432
```

1.2.4 hubble node

- 查看数据库节点状态

```
hubble node status  --all --certs-dir=certs  --host=hubble01:15432
```

```
  id | address   | is_available | is_live | is_decommissioning | membership|
     ↪ is_draining
+----+----------+ --------------+---------+
   1 | h3:26257 |  true         | true    |        true        |  active   |
      ↪ false
   2 | h2:26257 |  true         | true    |        true        |  active   |
      ↪ false
```

```
  3 | h3:26257 |  true          | true     |         true        |  active    |
     ↪ false
  4 | h4:26257 |  true          | true     |         true        |  active    |
     ↪ false
  5 | h5:26257 |  true          | true     |         true        |  active    |
     ↪ false
```

- 退役节点

```
hubble node --decommission    5     --certs-dir=certs  --host=hubble05:15432
```

```
 id | is_live | replicas | is_decommissioning | is_draining
+---+---------+----------+--------------------+-------------+
  5 |  true   |        0 |        true        |    false
(1 row)
```

1.2.5 hubble sql

- 连接数据库

```
hubble sql --certs-dir=/var/lib/hubble/certs --host=hubble01 --user hubble
```

```
# Welcome to the HubbleDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: HubbleDB v3.10.2-dirty (same version as client)
# Cluster ID: fb578fa1-b32c-4c71-bf5b-a4ee6f2ab28c
#
# Enter \? for a brief introduction.
#
hubble@hubble01:35432/defaultdb>
```

1.2.6 hubble version

- 查看数据库版本信息

```
hubble version
```

```
Build Tag:        v3.17-20-g40d6d40a-dirty
Build Time:       2023/01/13 09:52:28
Platform:         linux amd64 (x86_64-redhat-linux)
Go Version:       go1.17.8
C Compiler:       gcc 7.3.1 20180303 (Red Hat 7.3.1-5)
Build Commit ID:  40d6d40a8ac9cf8c8c94d8eba7b6dce4c9a329ea
Build Type:       development
```

2 部署

2.1 软硬件环境配置

Hubble 数据库，可以很好的部署和运行在 Intel、ARM 构架的服务器之上，支持主流的 Linux 操作系统环境。如有国产化系统版本建议要求的，请与天云公司 Hubble 售后部门确认继续情况

2.1.1 Linux 操作系统版本要求

操作系统平台	版本
CentOS	7.6 及以上的 7.x 版本
Red Hat Enterprise Linux	7.6 及以上的 7.x 版本

注意：
目前支持 Red Hat Enterprise Linux 8.2 ，其他 8.x 版本未测试
目前支持 CentOS Linux 8.2 ，其他 8.x 版本未测试

2.1.2 开发和测试环境-推荐配置

名称	要求
机器数量	3+
单台 CPU	2 路主频 2.4GHz 每路 16 核
单台内存	32GB*2
单台磁盘	480GB*2 SSD + 960GB*2 SSD
网络	千兆网卡
NFS 网盘	2T SSD

2.1.3 生产环境-推荐配置

名称	要求
机器数量	6+
单台 CPU	2 路主频 2.9GHz，每路至少 16 核
单台内存	32GB*8
单台磁盘	480GB*2 SSD + 960GB*6 SSD
网络	2 块万兆网卡
NFS 网盘	2T SSD

2.1.4 数据库管理系统一体机

名称	要求	备注
CPU	主频 32Ghz，320 核,320MB 缓存	处理能力
内存	1280GB	
容量	1PB	磁盘容量
显存	12GB GDDR6	

HUBBLE

名称	要求	备注
分辨率	1920*1080	支持 QHD 高分辨率

支持 Windows、Linux、Unix 和 ONIE 开放网络安装环境部署、支持第三方网络操作系统安装部署

支持数据导出、下载和备份

支持多副本存储，保障数据安全

支持网络端口光模块配置支持 25G 端口 128 个；支持 100G 端口可达 32 个

集中式管理，解放 IT 生产力

支持数据收集能力，支持多种协议接口接收，支持 2 个 100Gb 高速传输端口，支持有线、蓝牙、WIFI, 微波方式传输方式

支持超大容量存储

性能方面写入 >=6GB/s，读取 >=6GB/s

2.1.4.1 硬件建议

接下来所提到的 cpu 资源，指的是 vcore，也就是常说的超线程（hyperthreads）

硬件方面指导是平台无关的。可以是物理机、虚拟机、容器化的。

在正式部署环境前，对应用程序的工作负载模式进行测试，以及调优硬件设置是非常重要的一个环节。例如读多和写多的工作负载将对 cpu，内存，存储，IO 和网络的测试是有很大差异的。

指标项	建议
内存 /vcore	4GiB
存储 /vcore	150GiB
IOPS /vcore	500
MB/s /vcore	30

2.1.4.1.1 cpu 与内存

每个节点至少需要有 4 个 vcores。为了能有更好的性能，我们建议至少给每个节点提供 16 个 vcores，并给每个 vcore 提供 4GiB 的内存。

如果想获得更大的吞吐量，可以给每个节点配置更多的 vcore，最大建议为 32vcores。如还要进一步提高吞吐量，我们建议增加节点，而不是继续增加节点的物理硬件配置。

为了使得集群能有更好的弹性，我们建议使用更多的小节点，而不是使用更少的较大的节点。因为当数据分布在更多的小节点上时，故障节点的数据恢复的效率是更快的。

如果有大量的表，则需要更多的内存，由于元信息会在在每个节点的内存中，用于查询提速，当有巨量的表信息，无疑对内存的使用也将提升。

请避免使用超频或共享核心的虚拟机，这会限制 cpu 资源的负载。为了确保一致的性能，请确保所有节点具有统一的配置。

请关闭 linux 的 swap 功能，Hubble 有自己的内存与缓存的管理器。这是不依赖操作系统的缓存机制的。同时需要注意，当需要将页面读入内存时，在服务上过度分配内存可能会导致意料之外的性能问题。

2.1.4.1.2 存储

hubble 对于硬盘要求比较高，强烈推荐 SSD 类型，建议每块磁盘 960G 磁盘格式建议使用 EXT4 类型文件系统,xfs 能提供一定程度上的性能提升，但在长期的崩溃测试中，有出现过数据丢失的情况

我们建议给每个 vcore 提供对应 150GiB 的存储空间，如果您的工作负载对容量并没有很大的要求，那么每个 vcore 对应更少的存储空间，也是没问题的。无论这个节点的 vcore 是多少，最大的存储都不能超过 2.5TiB。

store 建议使用单独的卷，避免 store 的卷与其他 io 活跃的服务共用。建议将 log 路径与 store 区分开，这样可以避免相互影响。

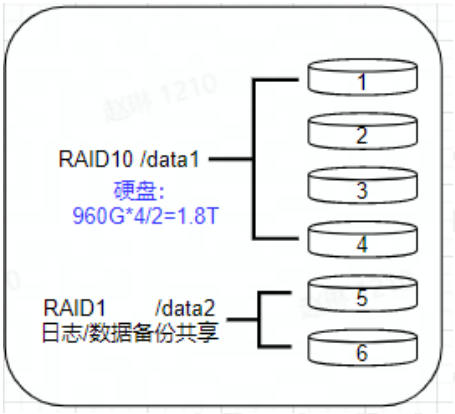


图 1: 图片

- 单台节点数据盘 6 块，每块 960GB

- 其中 /data1 RAID10 模式共占用 4 块硬盘用于存储数据文件
- 其中 /data2 作为 RAID1 模式共占用 2 块硬盘用于存放日志和数据备份文件

2.1.4.1.3 DISK I/O

磁盘必须能达到 500iops 和 30MB/s 对应每个 vcore。可以使用 sysbench 工具计算 IOPS，如果 iops 减少，建议增加节点，这样可以增加集群整体的 IOPS。

2.1.5 网络端口要求

默认端口	说明
48080	页面监听端口
15432	集群间通信端口

2.1.6 客户端 Web 浏览器要求

Hubble 数据库支持常用的 IE、Google Chrome、Mozilla Firefox 等较新版本访问

2.2 安装前服务配置

当我们准备好了软硬件环境后，就可以配置 Hubble 集群环境服务了

注意：
每台服务器都需要进行配置
使用 root 账户

2.2.1 系统级别配置

2.2.1.1 关闭交换分区

- 永久性关闭交换分区

```
sed -ri 's/.*swap.*/#&/' /etc/fstab
```

- 临时性关闭交换分区

```
swapoff -a   #关闭
swapon -a   #开启
```

注意：
建议永久性关闭交换分区 Hubble 数据库属于对 IO 敏感型基础架构，swapping 会将主内存交换到磁盘，从而对性能造成负面影响。内存中操作是需要快速执行的操作。如果内存交换到磁盘，100 微秒操作将花费 10 毫秒。建议系统上完全禁用 swapping

2.2.1.2 关闭服务器的防火墙

- 检查防火墙状态

```
systemctl status  firewalld.service
```

- 关闭防火墙

```
systemctl stop firewalld
```

- 关闭防火墙自启动

```
systemctl disable firewalld
```

- 检查防火墙状态

```
systemctl status  firewalld.service
```

2.2.1.3 关闭透明大页

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

2.2.1.4 禁用 selinux

- 永久性禁用

```
vi /etc/sysconfig/selinux
SELINUX=disabled
```

- 临时性禁用

```
setenforce 0
```

> **注意:**
> 建议永久性禁用 selinux

2.2.1.5 sysctl 配置

```
vi /etc/sysctl.conf
```

```
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
kernel.core_uses_pid = 1
kernel.pid_max = 4194303
kernel.randomize_va_space = 0
kernel.shmmax = 500000000
kernel.shmmni = 4096
kernel.shmall = 4000000000
kernel.sem = 250 51200 100 2048
kernel.sysrq = 1
kernel.msgmnb = 65536
kernel.msgmax = 65536
kernel.msgmni = 2048
fs.file-max = 1500000
net.ipv4.ip_forward = 0
net.ipv4.tcp_tw_recycle = 1
net.ipv4.conf.all.arp_filter = 1
net.ipv4.ip_local_port_range = 1025 65535
net.ipv4.icmp_echo_ignore_broadcasts = 1
net.ipv4.icmp_ignore_bogus_error_responses = 1
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn_backlog = 4096
net.ipv4.tcp_synack_retries = 3
net.ipv4.conf.all.log_martians = 1
net.ipv4.conf.default.log_martians = 1
net.ipv4.conf.all.accept_source_route = 0
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.tcp_no_metrics_save = 1
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_moderate_rcvbuf = 1
net.ipv4.tcp_retries2 = 3
```

HUBBLE

```
net.ipv4.tcp_keepalive_time = 600
net.ipv4.tcp_keepalive_probes = 3
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_fin_timeout = 10
net.core.netdev_max_backlog = 10000
net.core.rmem_max = 2097152
net.core.wmem_max = 2097152
net.core.somaxconn = 32768
vm.overcommit_memory = 0
vm.overcommit_ratio = 50
vm.swappiness = 0
vm.max_map_count = 6553000
```

```
sysctl -p
```

2.2.1.6 ulimit 配置

```
vi /etc/security/limits.d/hubble.conf
```

```
hubble - nofile  1000000
hubble - nproc   16000
```

```
echo 1500000 > /proc/sys/fs/file-max
```

2.2.1.7 主机与 IP 配置

```
vi /etc/hostname
```

```
hubble01
```

> **注意:**
> 主机名需要根据实际环境进行编排

```
vi /etc/sysconfig/network-scripts/ifcfg-xx
```

```
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
```

HUBBLE

```
NAME=p3p2
UUID=4f145ed6-e645-45c5-a91f-cbfb6df7213a
DEVICE=p3p2
ONBOOT=yes
IPADDR=192.168.1.11
PREFIX=24
GATEWAY=192.168.1.1
IPV6_PRIVACY=no
DNS1=8.8.8.8
DNS2=192.168.1.180
```

注意：
ifcfg-xx需要根据实际地址进行改动

2.2.1.8 hubble 账户与组配置

```
useradd -d /home/hubble -m hubble #添加用户与组
passwd  hubble   #设置密码
```

2.2.1.9 时间同步

- 安装 chrond

```
rpm -qa|grep chrony  # 确认是否安装了chrony
yum install chrony -y  #安装时间同步软件
```

- 配置

```
vi /etc/chrony.conf
```

```
server 192.168.1.11 iburst #选择集群中一台作为服务器
allow 192.168.1.0/24 # 设置运行同步的网段
local stratum 10 #同步时间周期10ms
```

- 自启动设置

```
systemctl start chronyd.service    #启动chrony服务
systemctl enable chronyd.service #开机启动
```

注意：
Chrony 是一个开源的自由软件，在 RHEL 7 操作系统，已经是默认服务,Chrony 作为 NTP 的替代，推荐使用 chrony
基于实际情况进行 chrony 配置
Hubble 数据库需要一定级别的时钟同步来保证数据的一致性。如果当一个节点检测到自身的时钟与集群中至少一半节点不同步，并超出了可接受的最大偏移量时（默认 500ms），这个节点将自动关闭服务，以保护数据库整体数据的一致性。所以 hubble 数据库集群的稳定性，需要建立在稳定的时钟同步前提下

2.2.1.10 NFS 配置（如果有网络共享盘跳过此步）

2.2.1.10.1 服务端配置

```
yum install nfs-utils rpcbind #安装nfs 和rpcbind服务
```

- 配置 nfs（防火墙关闭则无需配置）

```
vi /etc/sysconfig/nfs
#增加配置
LOCKD_TCPPORT=30001 #TCP锁使用端口
LOCKD_UDPPORT=30002 #UDP锁使用端口
MOUNTD_PORT=30003 #挂载使用端口
STATD_PORT=30004 #状态使用端口
```

- 防火墙访问开启（防火墙关闭则无需配置）

```
firewall-cmd --zone=public --add-port=111/tcp --permanent
firewall-cmd --zone=public --add-port=2049/tcp --permanent
firewall-cmd --zone=public --add-port=30001/tcp --permanent
firewall-cmd --zone=public --add-port=30002/tcp --permanent
firewall-cmd --zone=public --add-port=30003/tcp --permanent
firewall-cmd --zone=public --add-port=30004/tcp - permanent
```

- 增加服务端共享路径

```
mkdir  -p /data5/shares   #需要找一个磁盘空间大的文件目录
chmod  -R 777 /data5/shares/
```

- 配置服务端 nfs

```
vi /etc/exports  #默认不存在此文件新建即可
```

```
/data5/shares 192.168.1.0/24(rw,sync)
```

> **注意：**
> 指定集群中 1 台服务器中作为 NFS 挂载
> 设置的 NFS 盘一定要空间相对比较大

- 启动 nfs 服务

```
#先为rpcbind和nfs做开机启动：
systemctl enable rpcbind.service
systemctl enable nfs-server.service
#然后分别启动rpcbind和nfs服务：
systemctl start rpcbind.service
systemctl start nfs-server.service
```

- 配置服务生效

```
rpcinfo -p  #确认NFS服务器启动成功
exportfs -r #使exportfs生效
exportfs #可以查看到已经配置正确
```

2.2.1.10.2 客户端配置

- 创建共享目录

```
mkdir /data_shares
```

- 挂载共享目录

```
mount -t nfs 192.168.1.11:/data5/shares /data_shares
```

- 开机自动挂载

```
vi /etc/fstab
192.168.1.11:/data5/shares /data_shares nfs defaults 0 0
```

- 验证

```
df -h
```

```
192.168.1.11:/data5/shares  1.8T  507G  1.3T  29% /data_shares
```

> **注意:**
> 192.168.1.11 根据每台环境来设置
> 客户端配置集群中每台服务器都执行

2.2.2 服务器之间配置

2.2.2.1 主机之间主机名同步

```
vi /etc/hosts  #设置集群主机名称
```

```
192.168.1.11 hubble01
192.168.1.12 hubble02
```

> **注意:**
> 192.168.1.11 hubble01为每台服务器真实地址

```
scp -r /etc/hosts root@192.168.1.12:/etc/
```

> **注意:**
> 192.168.1.12 设置集群中每台节点主机名

2.2.2.2 Hubble 数据库用户免密配置

规划的 Hubble 启动用户做免密登录，使用 hubble 用户操作

```
ssh-keygen -t rsa     #一路回车 --【各个节点都执行】
cd ~/.ssh/
ssh-copy-id <目标节点地址>
```

```
ssh 192.168.1.12 #测试免密登录
```

2.3 Hubble 集群安装

当我们准备好了集群环境服务配置之后，就可以正式安装 Hubble 集群了。

2.3.1 安装 Hubble

使用 root 用户操作

- 解压安装包

```
tar -zxvf hubble.tar.gz
```

- 复制到/usr/local/bin目录下

```
cp -r hubble /usr/local/bin/
```

- 分发/usr/local/bin/hubble到集群所有节点的该目录下

```
scp -r /usr/local/bin/hubble node:/usr/local/bin/hubble
```

2.3.2 创建 Hubble 所需目录

在集群每个节点上，使用 root 用户操作

- 创建证书及数据存储目录并授权

```
mkdir /var/lib/hubble
mkdir /var/lib/hubble/certs
mkdir /var/lib/hubble/mysafedirectory
chown hubble:hubble -R /var/lib/hubble
chown -R hubble:hubble /usr/local/bin/hubble
chmod 711 /usr/local/bin/hubble
mkdir -p /data/hubbledir     ## 数据存储目录，按集群存储规划创建
chown -R hubble:hubble /data/hubbledir
```

注意：
`certs`生成 CA 证书以及所有节点和客户端证书
`mysafedirectory`用于创建签名证书 CA 密钥，用于整个集群创建和验证证书

2.3.3 添加 Hubble 数据库配置文件

在集群每个节点上，使用 root 用户操作

```
vi /etc/systemd/system/hubble.service
```

```
[Unit]
Description=Hubble Database cluster
Requires=network.target
[Service]
Type=notify
WorkingDirectory=/var/lib/hubble
ExecStart=/usr/local/bin/hubble start --locality=country=cn,region=ch-beijin,
    ↪ datacenter=tianyun,rack=1,node=1 --certs-dir=certs --listen-addr
    ↪ =0.0.0.0:15432 --advertise-host=hubble01 --join=hubble01:15432,hubble02
    ↪ :15432,hubble03:15432,hubble04:15432,hubble05:15432 --cache=30GiB --max-
    ↪ sql-memory=10GiB --store=path=/data3/hubbledir/hubble/logs,size=350GiB --
    ↪ store=path=/data4/hubbledir/hubble,size=350GiB --store=path=/data5/
    ↪ hubbledir/hubble,size=350GiB --http-addr=0.0.0.0:48080  --max-disk-temp-
    ↪ storage=10GiB
TimeoutStopSec=60
Restart=always
RestartSec=10
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=hubble
User=hubble
LimitNOFILE=1000000
[Install]
WantedBy=default.target
```

注意： + ExecStart=
`--locality`=country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=1 # 设置区域可以设置国家、地区、数据中心、机架、节点 ID
`--certs-dir`=certs # 设置证书
`--listen-addr`=hubble01:15432 # 监听地址和端口
`--advertise-host`=hubble01 # 当前节点的主机地址
`--join`=hubble01:15432,hubble02:15432,hubble03:15432,hubble04:15432,hubble05:15432 # 集群中的节点
`--cache`=30GiB # 缓存大小设置
`--max-sql-memory`=10GiB # 最大查询过程中所占用的内存大小，可以使用% 或者容量大小限制内存使用情况
`--store=path`=/data3/hubbledir/hubble,attrs=ssd,size=350GiB,rocksdb=write_buffer_size=134217728 # 设置数据存储目录，size 可以设置占用空间大小 (选填)，attrs 设置硬盘类型：转速，可以设置 rocksdb 方式写入同步
`--http-addr`=0.0.0.0:48080 # 页面访问端口

--max-disk-temp-storage=10GiB # 可用于存储基于磁盘的临时文件的最大存储容量
--external-io-dir=/data_shares # 指定外部目录用于数据导入导出，NFS 模式，推荐使用 NFS 模式
** 更多信息详情，可以使用 hubble start --help 查看 ** + RestartSec
RestartSec=10 10s 后重启 + User
User=hubble 设置用户

2.3.4 添加普通用户启停特权

在集群每个节点上，使用 root 用户操作

```
## 修改 hubble.service 文件的用户权限
chown hubble:hubble /etc/systemd/system/hubble.service
```

```
## 添加普通用户权限
vi /etc/sudoers

## Same thing without a password
hubble ALL=(root) NOPASSWD:/usr/bin/systemctl start hubble,/usr/bin/systemctl
    ↪ stop hubble,/usr/bin/systemctl restart hubble,/usr/bin/systemctl enable
    ↪ hubble,/usr/bin/systemctl disable hubble,/usr/bin/systemctl daemon-reload
```

- 保存并退出操作

```
:wq!
```

2.3.5 证书配置

之后的所有操作均在 license 执行注册的节点上使用普通用户操作

- 在第一台服务器生成证书，使用 hubble 账户

```
cd /var/lib/hubble/
hubble cert create-ca --certs-dir=certs --ca-key=mysafedirectory/ca.key    #生成
    ↪ 本机证书和密钥
```

- 生成其他服务器证书，使用 hubble 账户

```
cd /var/lib/hubble/
hubble cert create-node hubble02 192.168.1.12 --certs-dir=certs --ca-key=
    ↪ mysafedirectory/ca.key --overwrite
```

```
scp -r certs/ hubble@hubble02:/var/lib/hubble/
```

```
scp -r mysafedirectory/ hubble@hubble02:/var/lib/hubble/
```

注意：
`create-node hubble02 192.168.1.12` 创建其他节点的证书
基于 ca.key 生成其他节点证书
`scp` 分发到对应的节点上

- 生成本机服务器证书，使用 hubble 账户

```
cd /var/lib/hubble/
hubble cert create-node hubble01 192.168.1.11 --certs-dir=certs --ca-key=
    ↪ mysafedirectory/ca.key --overwrite
```

- 生成数据库客户端用户证书与密钥，使用 hubble 账户

```
cd /var/lib/hubble/
hubble cert create-client root --certs-dir=certs --ca-key=mysafedirectory/ca.key
    ↪  --overwrite
```

2.3.6 启动服务

使用 hubble 用户操作

```
sudo systemctl daemon-reload
sudo systemctl start hubble
```

2.3.7 初始化服务

使用 hubble 用户操作

```
hubble init --certs-dir=/var/lib/hubble/certs --host=hubble01:15432
```

注意：
`--host=hubble01:15432` 指定主机地址，集群中任意一台服务器

2.3.8 连接数据库，创建用户

使用 hubble 用户操作

```
hubble sql --certs-dir=/var/lib/hubble/certs --host=hubble01:15432
show databases;
create user hubble with password 'hubble';
grant admin to hubble with admin option;

set cluster setting kv.snapshot_rebalance.max_rate='32MiB';
set cluster setting kv.snapshot_recovery.max_rate='32MiB';
set cluster setting sql.distsql.temp_storage.workmem = '2GiB'
```

> **注意：**
> `--host=hubble01` 指定主机地址，集群中任意一台服务器

2.3.9 测试验证

- 页面访问，用户和密码为上一步创建的用户密码

```
https://192.168.1.11:48080/
```

- 查看进程

```
systemctl status hubble
```

2.3.10 设置开机自启动

```
systemctl enable hubble  ## root用户操作
sudo systemctl enable hubble  ## hubble用户操作
```

2.4 Hubble 单机安装

当我们准备好了单台服务器环境配置之后，就可以正式安装 Hubble 单机版本了

2.4.1 安装 Hubble

- 解压安装包

```
tar -zxvf hubble.tar.gz
```

- 复制到`/usr/local/bin`目录下

```
cp -r hubble /usr/local/bin/
```

2.4.2 创建 Hubble 所需目录

```
mkdir /var/lib/hubble
mkdir /var/lib/hubble/certs
mkdir /var/lib/hubble/mysafedirectory
chown hubble:hubble -R /var/lib/hubble
chown -R hubble:hubble /usr/local/bin/hubble
chmod 711 /usr/local/bin/hubble
```

> **注意：**
> `certs`生成 CA 证书以及所有节点和客户端证书
> `mysafedirectory`用于创建签名证书 CA 密钥，用于整个集群创建和验证证书

HUBBLE

2.4.3 添加 Hubble 数据库配置文件配置 _1

```
vi /etc/systemd/system/hubble1.service
```

```
[Unit]
Description=Hubble Database cluster
Requires=network.target
[Service]
Type=notify
WorkingDirectory=/var/lib/hubble
ExecStart=/usr/local/bin/hubble start --locality=country=cn,region=ch-beijin,
    datacenter=tianyun,rack=1,node=1 --certs-dir=certs --listen-addr=hubble01
    :15432 --advertise-host=hubble01 --join=hubble01:15432,hubble01:15433,
    hubble01:15434 --cache=30GiB --max-sql-memory=10GiB --store=path=/data1/
    hubbledir/hubble1,size=350GiB --http-addr=0.0.0.0:48080  --external-io-dir
    =/data_shares
TimeoutStopSec=60
Restart=always
RestartSec=10
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=hubble1
User=hubble
LimitNOFILE=1000000
[Install]
WantedBy=default.target
```

2.4.4 添加 Hubble 数据库配置文件配置 _2

```
vi /etc/systemd/system/hubble2.service
```

```
[Unit]
Description=Hubble Database cluster
Requires=network.target
[Service]
Type=notify
WorkingDirectory=/var/lib/hubble
ExecStart=/usr/local/bin/hubble start --locality=country=cn,region=ch-beijin,
    datacenter=tianyun,rack=1,node=2 --certs-dir=certs --listen-addr=hubble01
    :15433 --advertise-host=hubble01 --join=hubble01:15432,hubble01:15433,
    hubble01:15434 --cache=30GiB --max-sql-memory=10GiB --store=path=/data1/
    hubbledir/hubble2,size=350GiB --http-addr=0.0.0.0:48081  --external-io-
    dir=/data_shares
TimeoutStopSec=60
Restart=always
RestartSec=10
StandardOutput=syslog
StandardError=syslog
```

```
SyslogIdentifier=hubble2
User=hubble
LimitNOFILE=1000000
[Install]
WantedBy=default.target
```

2.4.5 添加 Hubble 数据库配置文件配置 _3

```
vi /etc/systemd/system/hubble3.service
```

```
[Unit]
Description=Hubble Database cluster
Requires=network.target
[Service]
Type=notify
WorkingDirectory=/var/lib/hubble
ExecStart=/usr/local/bin/hubble start --locality=country=cn,region=ch-beijin,
    ↪ datacenter=tianyun,rack=1,node=3 --certs-dir=certs --listen-addr=hubble01
    ↪ :15434 --advertise-host=hubble01 --join=hubble01:15432,hubble01:15433,
    ↪ hubble01:15434 --cache=30GiB --max-sql-memory=10GiB --store=path=/data1/
    ↪ hubbledir/hubble3,size=350GiB --http-addr=0.0.0.0:48082  --external-io-dir
    ↪ =/data_shares
TimeoutStopSec=60
Restart=always
RestartSec=10
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=hubble3
User=hubble
LimitNOFILE=1000000
[Install]
WantedBy=default.target
```

注意： + ExecStart=
`--locality`=country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=1 # 设置区域可以设置国家、地区、数据中心、机架、节点 ID
`--certs-dir`=certs # 设置证书
`--listen-addr`=hubble01:15432 # 监听地址和端口
`--advertise-host`=hubble01 # 当前节点的主机地址
`--join`=hubble01:15432,hubble02:15432,hubble03:15432,hubble04:15432,hubble05:15432 # 集群中的节点
`--cache`=30GiB # 缓存大小设置
`--max-sql-memory`=10GiB # 最大查询过程中所占用的内存大小，可以使用% 或者容量大小限制内存使用情况
`--store=path`=/data3/hubbledir/hubble,attrs=ssd,size=350GiB,rocksdb=write_buffer_size=134217728 # 设置数据存储目录，size 可以设置占用空间大小 (选填)，attrs 设置硬盘类型：转速，可以设

置 rocksdb 方式写入同步
`--http-addr=0.0.0.0:48080` # 页面访问端口
`--max-disk-temp-storage=10GiB` # 可用于存储基于磁盘的临时文件的最大存储容量
`--external-io-dir=/data_shares` # 指定外部目录用于数据导入导出，NFS 模式，推荐使用 NFS 模式
** 更多信息详情，可以使用 hubble start --help 查看 ** + RestartSec
`RestartSec=10` 10s 后重启 + User
`User=hubble` 设置用户

创建要用到的目录`/data1/hubbledir/hubble1`（`/data1/hubbledir/hubble2`，`/data1/hubbledir/hubble3`同理）

```
mkdir -p /data1/hubbledir/hubble1
chown -R hubble:hubble /data1/hubbledir/hubble1
chmod 755 /data1/hubbledir/hubble1
```

2.4.6 添加 Hubble 用户启停特权

使用 root 用户操作

- 修改 hubble1.service 文件的用户权限（hubble2.service，hubble3.service 同理）

```
chown hubble:hubble /etc/systemd/system/hubble1.service
chown hubble:hubble /etc/systemd/system/hubble2.service
chown hubble:hubble /etc/systemd/system/hubble3.service
```

- 添加普通用户权限

```
vi /etc/sudoers
##Allows people in group wheel to run all commands
hubble ALL=(root) NOPASSWD:/usr/bin/systemctl start hubble1,/usr/bin/systemctl
    ↪ stop hubble1,/usr/bin/systemctl restart hubble1,/usr/bin/systemctl enable
    ↪ hubble1,/usr/bin/systemctl disable hubble1,/usr/bin/systemctl daemon-
    ↪ reload,/usr/bin/systemctl start hubble2,/usr/bin/systemctl stop hubble2,/
    ↪ usr/bin/systemctl restart hubble2,/usr/bin/systemctl enable hubble2,/usr/
    ↪ bin/systemctl disable hubble2,/usr/bin/systemctl start hubble3,/usr/bin/
    ↪ systemctl stop hubble3,/usr/bin/systemctl restart hubble3,/usr/bin/
    ↪ systemctl enable hubble3,/usr/bin/systemctl disable hubble3
```

保存并退出操作:`wq!`

2.4.7 证书配置

- 服务器生成证书，使用 hubble 账户

```
cd /var/lib/hubble/
hubble cert create-ca --certs-dir=certs --ca-key=mysafedirectory/ca.key    #生成
    ↪ 本机证书和密钥
```

- 生成服务器证书，使用 hubble 账户

```
cd /var/lib/hubble/
hubble cert create-node hubble01 192.168.1.11 --certs-dir=certs --ca-key=
    ↪ mysafedirectory/ca.key --overwrite
```

- 生成数据库客户端用户证书与密钥，使用 hubble 账户

```
cd /var/lib/hubble/
hubble cert create-client root --certs-dir=certs --ca-key=mysafedirectory/ca.key
    ↪  --overwrite
```

2.4.8 启动服务

```
sudo systemctl daemon-reload
sudo systemctl start hubble1
sudo systemctl start hubble2
sudo systemctl start hubble3
```

> **注意：**
> 使用 hubble 用户执行

2.4.9 初始化服务

```
hubble init --certs-dir=/var/lib/hubble/certs --host=hubble01:15432
```

> **注意：**
> --host=hubble01:15432 指定主机地址，集群中任意一台服务器

2.4.10 连接数据库，创建用户

```
hubble sql --certs-dir=/var/lib/hubble/certs --host=hubble01    #连接
show databases;
create user hubble with password 'hubble';
grant admin to hubble with admin option;

set cluster setting kv.snapshot_rebalance.max_rate='32MiB';
set cluster setting kv.snapshot_recovery.max_rate='32MiB';
set cluster setting sql.distsql.temp_storage.workmem = '2GiB'
```

> **注意：**
> --host=hubble01 指定主机地址，集群中任意一台服务器

2.4.11 测试验证

- 页面访问

```
https://192.168.1.11:48080/
```

- 查看进程

```
systemctl status hubble1
systemctl status hubble2
systemctl status hubble3
```

2.4.12 设置开机自启动

> 使用 root 操作

```
systemctl enable hubble1
systemctl enable hubble2
systemctl enable hubble3
```

3 运维

3.1 数据备份与恢复

3.1.1 数据备份

BACKUP 语句用于备份整个数据库或者部分表进行完整备份或增量备份，同时支持指定时间戳备份。

3.1.1.1 备份语法

```
BACKUP <targets...> INTO <location...>
       [ LATEST IN <expr> ]
       [ AS OF SYSTEM TIME <expr> ]
       [ WITH <option> [= <value>] [, ...] ]
```

3.1.1.1.1 参数说明

- LOCATION 指定存储备份的目录。

- AS OF SYSTEM TIME `<TIMESTAMP>`可以指定备份截止日期时间。

> **注意：**
> 只有 admin 角色用户可以运行 BACKUP 备份命令
> 如表存在外键约束，序列，视图，交错表。必须同时备份其依赖的对象。
> 恢复的表从目标数据库继承特权授予；它们不保留来自备份表的特权授予，因为恢复集群可能有不同的用户。

3.1.1.2 备份数据说明

HUBBLE

类型	schema	host	参数	示例
NFS/Local	nodelocal	节点 ID 或为空	N/A	`nodelocal://n/path/mydatest`

3.1.1.3 全量备份示例

全量备份生成的备份数据，大概与集群存储占用容量相同。

- 备份集群

```
backup into 'nodelocal://1/backup';
```

- 备份整个数据库

```
backup database guar into  'nodelocal://1/backup_datadb/test';
```

- 备份某几张表

```
backup guar.trans,guar.tb_bs into 'nodelocal://1/backup_data'
```

3.1.1.4 增量备份

- 增量备份整个数据库

```
backup database guar into latest in  'nodelocal://1/backup_datadb/test';
```

3.1.1.5 指定时间戳备份

```
backup database guar into 'nodelocal://1/backup_datadb'  AS OF SYSTEM TIME '-20s
    ↪ ' WITH revision_history;
```

> **注意:**
> 推荐使用nfS共享存储模式
> 使用 nfs 模式必须在 hubble.service 文件中 ExecStart 启动中指定外部目录 `--external-io-dir`
> ↪ =/data_shares

3.1.2 数据恢复

Hubble 的设计具有很高的容错性，所以这些备份主要是为灾难恢复而设计的。如果集群丢失了大部分节点，可通过数据备份恢复。而少部分节点出现问题是不需要任何干预的。

您可以从备份中恢复整个表 (包括其索引) 或视图。此过程使用存储在备份中的数据在目标数据库中创建全新的表或视图。恢复数据库只是恢复属于数据库的所有表和视图，但不创建数据库。

因为这个过程是为灾难恢复而设计的，所以需要这些表当前不存在于目标数据库中。这意味着目标数据库必须没有需要恢复的表或视图同名的表或视图。可以做以下工作:

- 先执行`DROP TABLE`，`DROP VIEW`，`DROP SEQUENCE`，然后在使用数据恢复功能。需要注意序列在列的默认表达式中使用时不能被删除，因此必须在删除序列之前删除那些表达式，并在重新创建序列之后重新创建它们。可以使用 setval 函数将序列的值设置为以前的值。

- 将表或视图还原到另一个数据库中。

根据所包含的备份文件，您可以从完全备份进行恢复，也可以从带有增量备份的完全备份进行恢复。

全量恢复：只包含到完整备份的路径。

增量恢复：包括作为第一个参数的完整备份路径，以及作为以下参数的从旧备份到最新备份的后续增量备份。

在成功启动恢复后，它将成为一个 job 任务，您可以使用`SHOW JOBS`来查看。

启动恢复之后，您可以使用`PAUSE JOB`、`RESUME JOB`和`CANCEL JOB`对任务进行操作。

3.1.2.1 依赖对象

如表存在外键约束，序列，视图，交错表。必须同时备份其依赖的对象。

对象	依赖对象
有外键约束的表	所关联的表。同时，可以在数据恢复期间将其删除
有序列的表	所使用的序列。同时，可以在数据恢复期间将其删除
视图	视图定义使用到的表
交错表	交错关系中的父表

3.1.2.2 备份数据路径

每个备份的路径必须是唯一的，备份位置的 URL 必须使用以下格式:

```
[scheme]://[host]/[path]?[parameters]
```

类型	schema	host	参数	示例
http	http	主机地址	N/A	`http://localhost:8080/mydatest.sql`
NFS/Local	nodelocal	节点 ID 或为空	N/A	`nodelocal://n/path/mydatest/2023/03/30-010108.31`

> 恢复数据的时候要找`BACKUP_MANIFEST`文件，将地址写到与`BACKUP_MANIFEST`一致，请看以下示例。

3.1.2.3 参数说明

可以将以下选项，用以控制恢复进程的行为。

参数	描述	示例
`skip_missing_foreign_keys` ↪	如果希望恢复具有外键的表，但不希望恢复它引用的表，则可以从表中删除外键约束，然后恢复它。	WITH skip_missing_foreign_keys
`skip_missing_sequences` ↪	如果希望还原依赖于序列的表，但不希望还原它引用的序列，则可以从表中删除序列依赖项	WITH skip_missing_sequences
`skip_missing_views`	如果您想要还原具有视图的表，但又不想还原视图的依赖项，您可以删除视图，然后恢复表。	WITH skip_missing_views

3.1.2.4 解决命名冲突

在恢复表和数据库的时候，会遇见命名冲突的问题，有两种解决形式

- 第一种：删除表或者数据库
- 第二种：更改表的名称或者数据库名

> 建议选用第 2 种方式

- 更改数据库名

```
alter database database_name rename to database_bak;
```

- 更改表名

```
alter table target_database.example_table rename to target_database.
    ↪ archived_example_table;
```

3.1.2.5 示例

完成恢复命令后，找到BACKUP_MANIFEST文件所在位置

```
[hubble@ty-bj03-test01 30-010108.31]$ pwd
/data/hubble317cyh/extern/backup_tt/2023/03/30-010108.31
```

执行恢复语句

```
root@ty-bj03-test01:14432/test> restore  test.tt from 'nodelocal://1/backup_tt
    ↪ /2023/03/30-010108.31';
        job_id        |  status   | fraction_completed | rows | index_entries |
            ↪ bytes
--
    ↪ -------------------+-----------+--------------------+------+---------------+---------
    ↪
  852193907899072513 | succeeded |                  1 |    1 |             0 |
      ↪     16
```

3.1.2.5.1 恢复完整集群

完整集群还原只能在从未有用户创建的数据库或表的目标集群上运行

```
restore from 'nodelocal://1/backup/2023/03/30-055845.78';
```

3.1.2.5.2 恢复单个表

```
restore guar.trans from 'nodelocal://1/backup_data/2023/03/30-010108.31';
```

3.1.2.5.3 恢复多个表

```
restore guar.trans,guar.tb_bs from 'nodelocal://1/backup_data
    ↪ /2023/03/30-010108.31';
```

3.1.2.5.4 恢复整个数据库

带有new_db_name参数，恢复数据时候会生成一个新的数据库

> 库中不能包含视图，若有视图，则无法指定新的库名

```
restore database guar from latest in 'nodelocal://1/backup_db
    ↪ /2023/03/30-022407.66' with new_db_name ='guar_db';
```

3.1.2.5.5 指定恢复时间点

```
restore guar.trans from 'nodelocal://1/backup_datatime/2023/03/30-022457.26' AS
    ↪ OF SYSTEM TIME '-30s';
```

3.1.2.5.6 恢复到另一个数据库

默认情况下，表和视图被恢复到它们原来所属的数据库。但是，使用 into_db 选项，您可以控制目标数据库。

```
restore guar.trans,guar.tb_bs from 'nodelocal://1/backup_data
    ↪ /2023/03/30-021207.04' WITH into_db = 'ora';
```

3.1.2.5.7 在恢复之前删除外键

默认情况下，具有外键约束的表必须与它们引用的表同时恢复。但是，使用skip_missing_foreign_keys选项可以从表中删除外键约束，然后再恢复。

```
restore guar.emp from 'nodelocal://1/backup_data/2023/03/30-324121.03' with
    ↪ skip_missing_foreign_keys;
```

3.1.2.5.8 从系统中恢复用户，用户备份

system.users表存储集群的用户名及其散列密码。要恢复它们，需要将system.users表恢复到新数据库中，因为您不能删除现有的system.users表。

在将其恢复到新数据库之后，可以将恢复的用户表数据写入集群的现有system.users表。

```
restore system.users from 'nodelocal://1/backup_users/2023/03/30-104421.14' WITH
    ↪  into_db = 'newdb';
```

```
insert into system.users select * from newdb.users;
```

```
drop table newdb.users;
```

3.2 集群扩容缩容

3.2.1 集群扩容

以下过程参考标准生产环境，使用的是安全模式。如未采用安全模式，可省略生成证书的步骤。

3.2.1.1 新节点准备

参考集群环境服务配置，做免密及用户配置

3.2.1.2 目录准备

在新节点上，使用 root 用户操作

```
mkdir /var/lib/hubble
mkdir /var/lib/hubble/certs
mkdir /var/lib/hubble/mysafedirectory
chown hubble:hubble -R /var/lib/hubble
mkdir -p /data/hubbledir    ## 数据存储目录，按集群存储规划创建
chown -R hubble:hubble /data/hubbledir
```

3.2.1.3 证书配置

在现役节点上，使用 hubble 用户操作

- 传输现有服役节点的程序及节点证书和 key 传输到新节点对应得目录下：

```
scp /usr/local/bin/hubble <new node>:/usr/local/bin/
scp /var/lib/hubble/certs/ca.crt <new node>:/var/lib/hubble/certs/
scp /var/lib/hubble/mysafedirectory/ca.key <new node>:/var/lib/hubble/
    ↪ mysafedirectory/
```

- 登录到新节点，生成该节点证书，证书目录分配与现有一致

```
cd /var/lib/hubble
hubble cert create-node <new node hostname> <new node ip> --certs-dir=certs --ca
    ↪ -key=mysafedirectory/ca.key --overwrite
```

3.2.1.4 添加 hubble 数据库配置文件

在新节点上，使用 root 用户操作

```
vi /etc/systemd/system/hubble.service
```

```
[Unit]
Description=Hubble Database cluster
Requires=network.target
[Service]
Type=notify
WorkingDirectory=/var/lib/hubble
ExecStart=/usr/local/bin/hubble start --locality=country=cn,region=ch-beijin,
    ↪ datacenter=tianyun,rack=1,node=<new node id> --certs-dir=certs --listen-
    ↪ addr=0.0.0.0:15432 --advertise-host=<new node> --join=hubble01:15432,
    ↪ hubble02:15432,hubble03:15432,hubble04:15432,hubble05:15432,<new node
    ↪ >:15432 --cache=30GiB --max-sql-memory=10GiB --store=path=/data3/hubbledir
    ↪ /hubble,attrs=ssd,size=350GiB,rocksdb=write_buffer_size=134217728 --store=
    ↪ path=/data4/hubbledir/hubble,attrs=ssd,size=350GiB,rocksdb=
```

```
    ↪ write_buffer_size=134217728 --store=path=/data5/hubbledir/hubble,attrs=ssd
    ↪ ,size=350GiB,rocksdb=write_buffer_size=134217728 --http-addr=0.0.0.0:48080
    ↪   --max-disk-temp-storage=10GiB
TimeoutStopSec=60
Restart=always
RestartSec=10
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=hubble
User=hubble
LimitNOFILE=1000000
[Install]
WantedBy=default.target
```

3.2.1.5 添加普通用户启停特权

在新节点上，使用 root 用户操作

修改hubble.service文件的用户权限

```
chown hubble:hubble /etc/systemd/system/hubble.service
```

添加普通用户权限

```
vi /etc/sudoers
```

```
## Same thing without a password
hubble ALL=(root) NOPASSWD:/usr/bin/systemctl start hubble,/usr/bin/systemctl
    ↪ stop hubble,/usr/bin/systemctl restart hubble,/usr/bin/systemctl enable
    ↪ hubble,/usr/bin/systemctl disable hubble,/usr/bin/systemctl daemon-reload
```

添加完成后，输入:wq!保存退出

```
:wq!
```

3.2.1.6 启动服务

使用 hubble 用户操作

```
sudo systemctl daemon-reload
sudo systemctl start hubble
```

3.2.1.7 测试验证

- 页面访问，查看集群状态

```
https://192.168.1.11:48080/
```

- 查看进程

```
systemctl status hubble
```

- 命令行连接新节点

```
hubble sql --certs-dir=/var/lib/hubble/certs --host=<new node>
```

3.2.1.8 设置开机自启动

```
systemctl enable hubble  ## root用户操作
sudo systemctl enable hubble  ## hubble用户操作
```

3.2.2 集群缩容

停用节点时，Hubble 允许节点完成运行中的请求，拒绝任何新请求，并将所有范围副本和范围租约转移到该节点之外，以便可以安全地将其关闭。

> 注意事项在停用某个节点之前，请确保其他节点可用于从该节点接管分片副本。如果没有其他可用节点，则停用过程将无限期地挂起。当前版本节点停用后将永久删除，不可以重新启用。当退役多个节点时候，建议单独退役节点，防止分片副本接管过程中出现问题。

3.2.2.1 退役单个活动节点 (安全模式)

3.2.2.1.1 前提条件

为确保您的群集可以充分处理停用节点：

- 一次仅停用一个节点，并且在停用每个节点之前，请确认没有重复不足或不可用的分片。
- 如果您有一个正在挂起的停用节点，则可以重新启用该节点。
- 如果可能，请保持节点运行而不是杀死它，因为挂起的停用过程可能导致问题，甚至数据丢失。
- 确认有足够的节点来接管您要删除的节点的副本。请参阅上面的一些示例方案。

3.2.2.1.2 步骤 1: 检查节点

展示所有节点状态：

```
hubble node status  --all  --certs-dir=certs --host=<any address of nodes>
```

参考以上语句

```
hubble node status  --all --certs-dir=certs  --host=hubble01:15432
```

```
  id | address   | is_available | is_live | is_decommissioning | membership|
     ↪ is_draining
+----+----------+ --------------+---------+
   1 | h3:26257 |  true         | true    |        true        |  active   |
     ↪ false
   2 | h2:26257 |  true         | true    |        true        |  active   |
     ↪ false
```

```
  3 | h3:26257 |  true          | true     |         true        |  active    |
    ↪ false
  4 | h4:26257 |  true          | true     |         true        |  active    |
    ↪ false
  5 | h5:26257 |  true          | true     |         true        |  active    |
    ↪ false
```

3.2.2.1.3 步骤 2: 关闭需要停用的节点:

```
systemctl stop hubble.service
```

3.2.2.1.4 步骤 3: 退役节点

SSH 到运行该节点的计算机，并使用--decommission命令：(<NODEID>换成数字 5 即可)

```
hubble node --decommission <NODEID> --certs-dir=certs --host=<address of node to
    ↪ remove>
```

参考以上语句

```
hubble node --decommission    5     --certs-dir=certs  --host=hubble05:15432
```

然后，您将在退役状态更改时将信息打印：

```
 id | is_live | replicas | is_decommissioning | is_draining
+---+---------+----------+--------------------+-------------+
  5 |  true   |       73 |        true        |    false
(1 row)
```

节点完全退役并停止后，您将看到一个确认：

```
 id | is_live | replicas | is_decommissioning | is_draining
+---+---------+----------+--------------------+-------------+
  5 |  true   |        0 |        true        |    false
(1 row)

No more data reported on target nodes. Please verify cluster health before
    ↪ removing the nodes.
ok
```

3.2.2.1.5 步骤 4: 停用后检查群集

检查集群状态

```
hubble node status --all --certs-dir=certs --host=<any address of nodes>
```

参考以上语句

```
hubble node status  --all --certs-dir=certs  --host=hubble01:15432
```

```
  id | address   | is_available| is_live | is_decommissioning|  membership  |
     ↪ is_draining
+----+----------+
   ↪ ------------+---------+-------------------+--------------+-------------
   1 | h3:26257 |  true       | true    |        false      |  active      |
      ↪ false
   2 | h2:26257 |  true       | true    |        false      |  active      |
      ↪ false
   3 | h3:26257 |  true       | true    |        false      |  active      |
      ↪ false
   4 | h4:26257 |  true       | true    |        false      |  active      |
      ↪ false
(4 rows)
```

检查集群decommission状态:

```
hubble node status --decommission  --certs-dir=certs --host=<any address of
   ↪ nodes>
```

参考以上语句

```
hubble node status  --decommission --certs-dir=certs  --host=hubble01:15432
```

```
id | address   | is_available| is_live | is_decommissioning|  membership  |
   ↪ is_draining
+----+----------+
   ↪ ------------+---------+-------------------+--------------+-------------
   1 | h3:26257 |  true       | true    |        false      |  active      |
      ↪ false
   2 | h2:26257 |  true       | true    |        false      |  active      |
      ↪ false
   3 | h3:26257 |  true       | true    |        false      |  active      |
      ↪ false
   4 | h4:26257 |  true       | true    |        false      |  active      |
      ↪ false
   5 | h5:26257 |  false      | false   |        true       |decommissioned|
      ↪ false
(5 rows)
```

> 注意: id=5 的节点is_available，is_live，is_decommissioning和membership值与其他节点的不同

3.3 监控告警

管理员界面提供有关集群和数据库配置的详细信息，并帮助您优化集群性能。

3.3.1 管理员界面区域

导航栏	区域	描述
概况		查看集群节点的详细信息。
监控信息	概况	查看重要的 SQL 性能，复制和存储指标。监控信息-> 仪表盘：概况
	硬件	查看有关 CPU 使用率，磁盘吞吐量，网络负载，存储容量和内存的指标。监控信息-> 仪表盘：硬件
	运行	查看有关节点数，CPU 时间和内存使用情况的指标。监控信息-> 仪表盘：运行
	SQL	查看有关 SQL 连接、字节流量、查询、事务和服务延迟的指标。监控信息-> 仪表盘：SQL
	存储	查看有关存储容量和文件描述符的指标。监控信息-> 仪表盘：存储
	复制	查看有关如何跨集群复制数据的度量标准，例如分片，每个存储的副本数和副本静态数。监控信息-> 仪表盘：复制
	分布式	查看 batches，RPCs，kv 事务等相关指标。监控信息-> 仪表盘：分布式
	队列	查看队列相关指标，例如队列处理失败数，队列执行时间，副本GC队列等。监控信息-> 仪表盘：队列
	慢请求	查看相关慢 raft，慢租赁采集，慢锁采集的指标。监控信息-> 仪表盘：慢请求
	数据变动	查看有关数据变动的指标，例如最大数据变动延时，接收字节流量等。监控信息-> 仪表盘：数据变动
	节点信息	查看活动，停止和停用节点的详细信息。监控信息-> 汇总
	事件	查看最近的集群事件列表。监控信息-> 事件
数据库列表		查看有关集群中系统和用户数据库的详细信息。
事务列表		事务列表识别任务的执行情况。
语句列表		识别频繁执行或高延迟的 SQL 语句
网络延迟		识别集群节点的网络运行情况
任务列表		查看集群中正在运行的作业的详细信息。

3.3.2 管理员界面访问

在不安全的集群上，所有用户均可访问管理界面的所有区域。

在安全集群上，`admin`用户只能访问管理员界面的某些区域。

出于安全原因，非管理员用户只能访问其具有特权的数据和不需要特权的数据（例如，集群运行状况，节点状态，指标）。

安全区域	权限信息
节点映射	数据库和表名
数据库细节	数据量
状态细节	SQL 语句
作业细节	SQL 状态和操作细节
进阶调试页	存储的表数据，操作详细信息，内部 IP 地址，名称，凭据，应用程序数据（取决于报告）

3.3.3 仪表盘

集群概况页面提供了集群节点的详细信息，及其活动状态，分片状态，正常运行时间和关键硬件指标的详细信息。

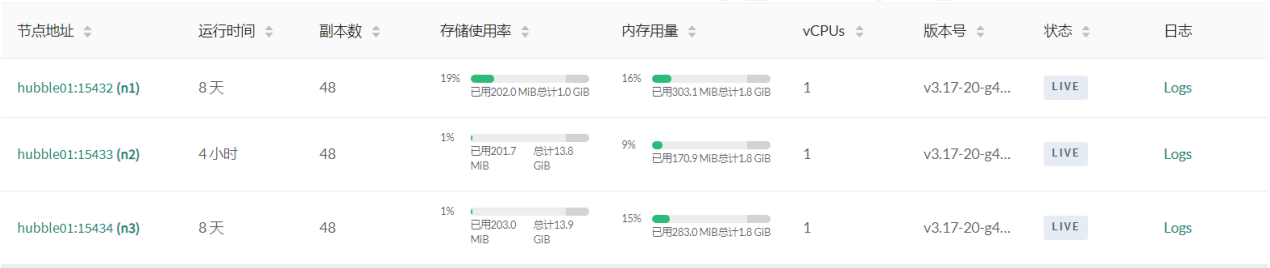
3.3.3.1 整体预览

集群概况面板提供以下指标：

指标	描述
存储情况	已用容量：已使用的存储容量（表示为在所有节点之间分配的总存储容量的百分比）。可用容量：Hubble 数据库能够使用的容量（不包括 Hubble 程序，操作系统和其他系统文件使用的容量）。
节点状态	集群中活动节点的数量。集群中可疑节点的数量。如果节点无法获取其活动状态或该节点正在退役，则认为该节点是可疑节点。集群中宕机节点的数量。
副本状态	集群中 shard 的总数。集群中正在复制的 shard 数。非零值表示群集不稳定。集群中不可用的 shard 数量。非零值表示群集不稳定。

3.3.3.2 节点列表

节点列表是概况页面上的默认视图。



3.3.3.3 LIVE 节点列表

LIVE 节点列表是集群中在线且能够响应请求的节点，它们用绿点标记。如果节点被移除或死亡，则该绿点变为黄点，表示该节点没有响应。如果该节点在一定时间（默认情况下为 5 分钟）内一直无响应，则该黄点将变为红点，并移至 DEAD 节点列表，表明不再期待该节点回归集群。

部分信息说明：

字段	描述
节点 ID	节点 ID
节点地址	节点的地址。可以单击该地址以查看有关该节点的更多详细信息。
运行时间	查看节点运行时间
副本	节点副本的数量
CPUS	节点所在机器的 CPU 核数
存储使用率	节点上已使用的存储容量占总可用容量的百分比。该值以数字和条形图表示。
内存使用率	节点上已使用的内存占总内存的百分比。该值以数字和条形图表示。
日志	单击日志以查看该节点的详细日志。在安全群集上需要管理员权限。

3.3.4 监控信息

3.3.4.1 概况

集群指标仪表板显示关键指标的概况面板。要查看概况面板，请单击左侧的监控信息。

概况面板提供以下指标（不一定全部显示）：

指标	描述
Total Nodes	集群中的节点总数。
可疑节点	集群中的可疑节点数。
宕机节点	集群中的宕机节点数。
存储使用	表示为在所有节点已用容量与的总存储容量的百分比。
不可用分片	集群中不可用分片的数量。非零数字表示群集不稳定。
每秒查询数	每秒在整个集群中执行的SELECT，UPDATE，INSERT和DELETE查询的总数。
99% 的延时	最后一分钟内执行了 99% 的查询。

3.3.4.2 事件列表

集群指标仪表板显示事件面板，该面板列出了集群中所有节点记录的 10 个最新事件。要查看事件面板，请单击左侧导航栏上的监控信息。要查看所有事件的列表，请在事件面板中单击查看所有事件。

列出了以下类型的事件：

- `Database Created`
- `Database Dropped`
- `Table Created`
- `Table Dropped`

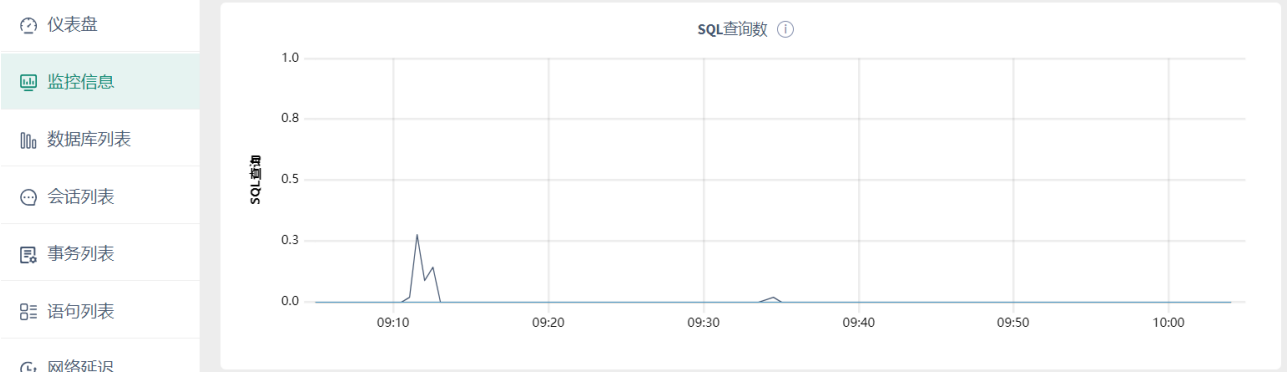
- Table Altered
- Index Created
- Index Dropped
- View Created
- View Dropped
- Schema Change
- Sequence Created
- Node Joined
- Node Rejoined
- Node Decommissioned
- Node Restarted
- Cluster Setting Changed
- Zone Config Changed

3.3.4.3 监控信息-概况仪表板

概况仪表板使您可以监视重要的 SQL 性能和存储情况等指标。要查看此信息中心，请访问管理界面，然后点击左侧导航栏上监控信息。默认情况下会显示概况仪表板。

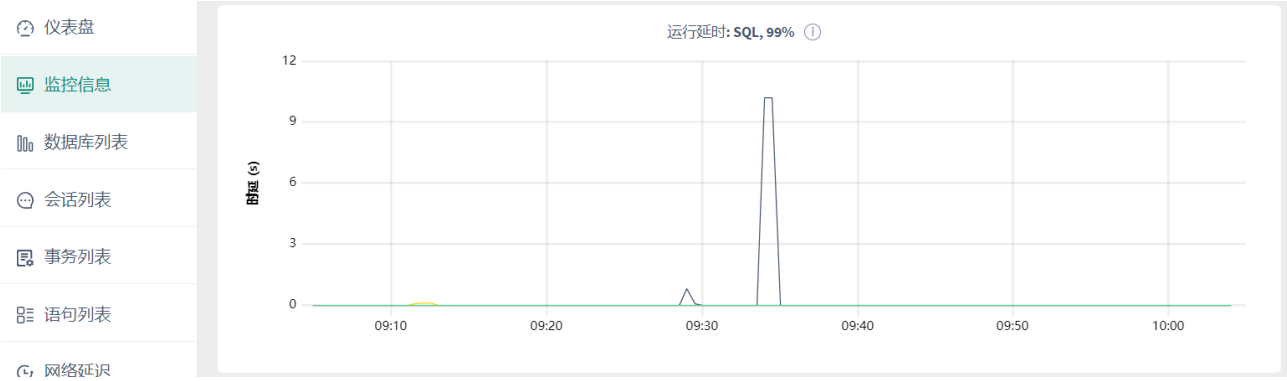
概述仪表板显示以下时间序列图：

3.3.4.3.1 SQL 查询数



- 在节点视图中，该时间序列图显示的是指定节点处理客户端请求的 SQL 查询数，查询包括SELECT/INSERT/UPDATE/DELETE语句。采样值为采样周期 10 秒内的平均值。
- 在集群视图中，该时间序列图整合了每个节点统计信息，将每个节点最近 10 秒的活动情况进行汇总，结果视作当前集群查询负载的估计值。

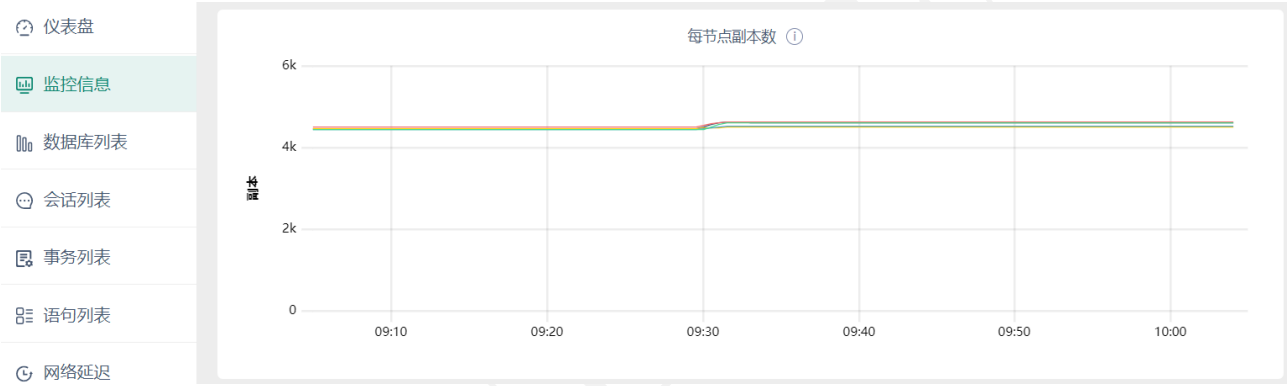
3.3.4.3.2 运行延迟



运行延时是集群从接收到查询请求到查询执行结束之间的时间，不包含将查询结果传输给客户端的时延。

- 在节点视图，在最后一分钟内，此节点在此时间内执行了 99% 的查询。此时间不包括节点和客户端之间的网络延迟。
- 在集群视图中，该图显示了集群中所有节点在此时间内执行了 99% 的查询。

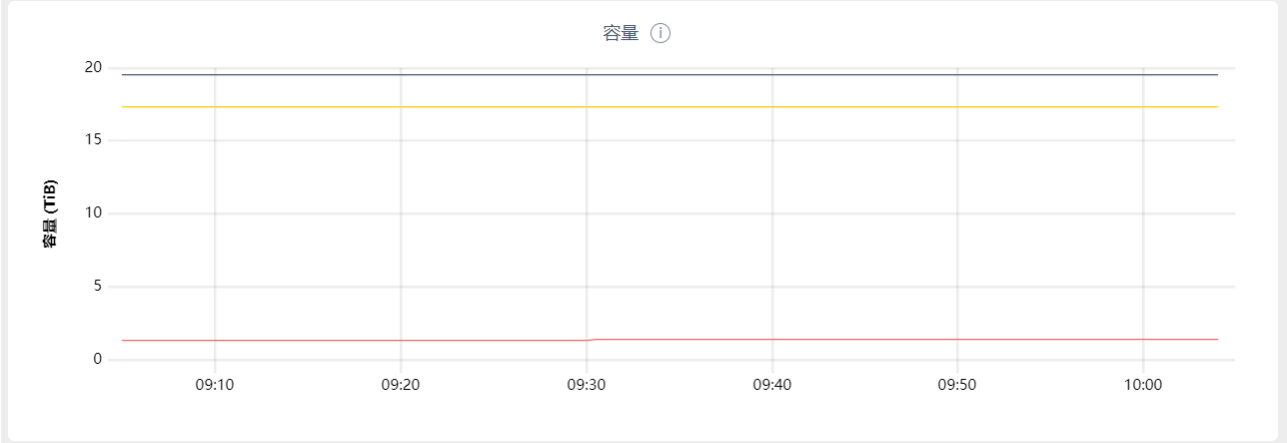
3.3.4.3.3 每节点的副本数



- 在节点视图中，该图显示了存储在此节点上的副本数。分片是这个的子集，复制分片生成副本是为了保障可用性。
- 在集群视图中，该图显示了集群中每个节点上的副本数。

有关如何控制副本的数量和位置的详细信息，请参阅配置复制区域。

3.3.4.3.4 容量



可以通过监控容量图来判断什么时候需要为集群添加新的存储空间。

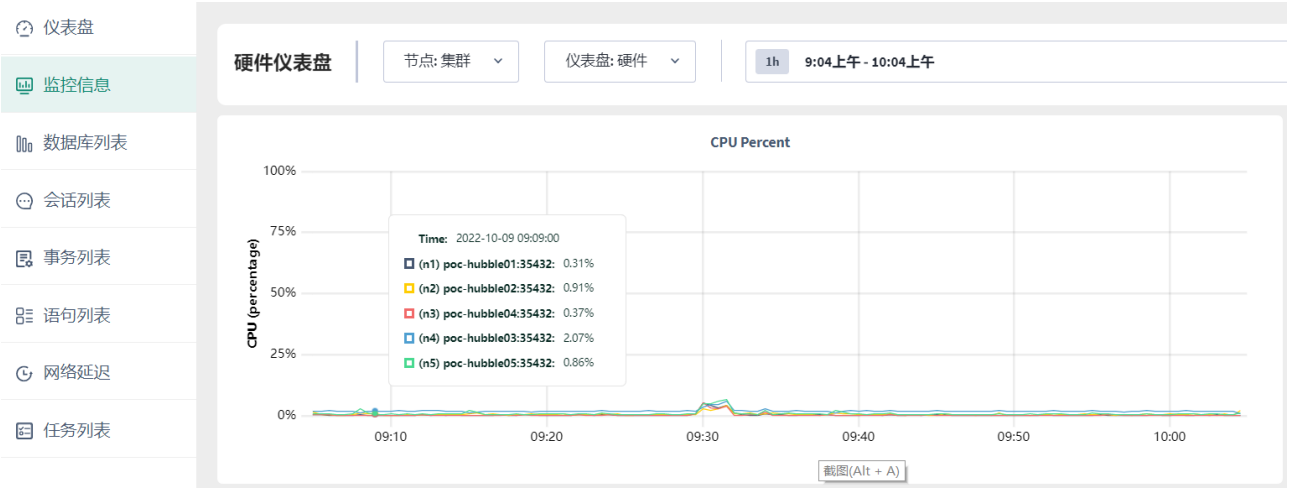
- 在节点视图中，该时间序列图展示了在集群中指定节点的最大分配容量、可用容量和已使用容量的情况。

- 在集群视图中，该时间序列图展示了集群中所有节点的最大分配容量总和、可用容量总和和已使用容量总和

将鼠标悬停在图形上时，将显示以下指标的值：

指标	描述
容量	分配给数据库的存储容量。您可以为指定节点通过`--store`配置可用最大存储容量。
可用	可用的空闲存储容量。
已用	已使用的磁盘空间。请注意，此值小于容量-可用，因为容量和可用指标统计的是整个硬盘，受硬盘上的所有程序的影响，已用指标只统计已存储的磁盘使用情况。

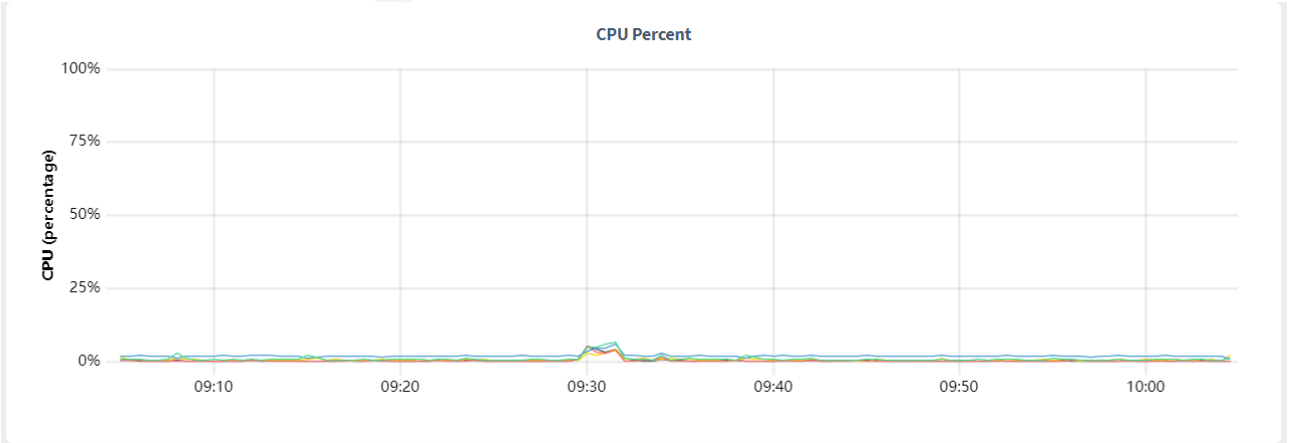
3.3.4.4 硬件仪表盘



硬件仪表板能够帮助用户监控集群节点的 CPU 使用率、硬盘吞吐量、网络负载、存储容量和内存的使用情况。要查看此仪表板，请访问管理界面，单击左侧的监控信息，然后选择仪表板 > 硬件。

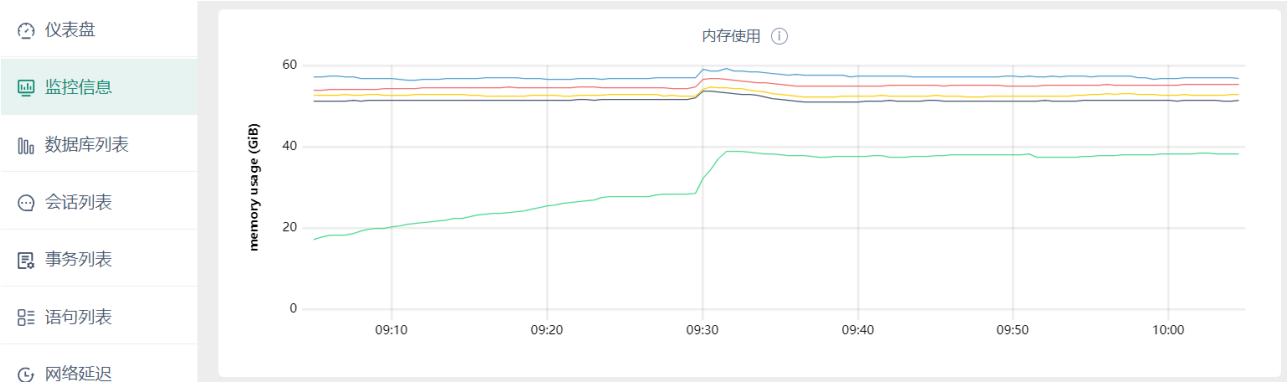
硬件仪表板显示以下时间序列图：

3.3.4.4.1 CPU 使用率



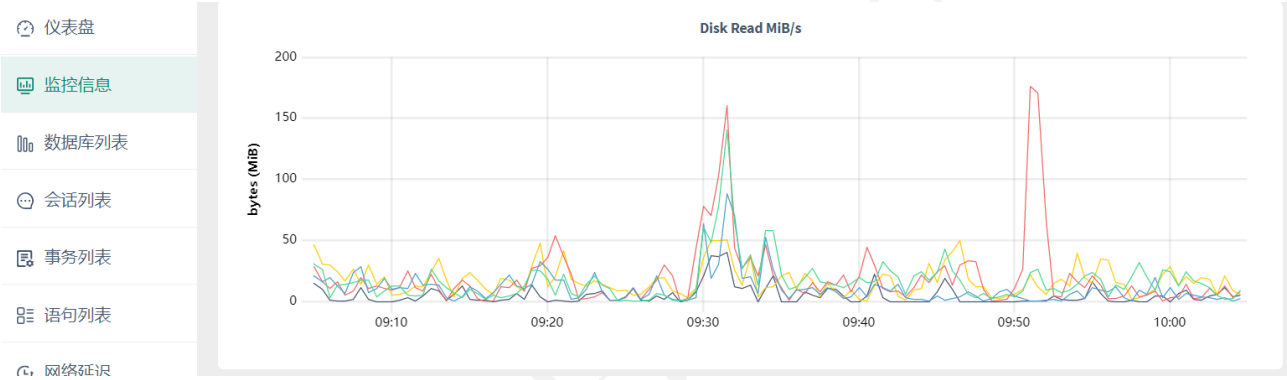
- 在节点视图中，该时间序列图显示了指定节点上运行的 CPU 使用率情况。
- 在集群视图中，该时间序列图显示了集群所有节点上运行的 CPU 使用率情况。

3.3.4.4.2 内存使用情况



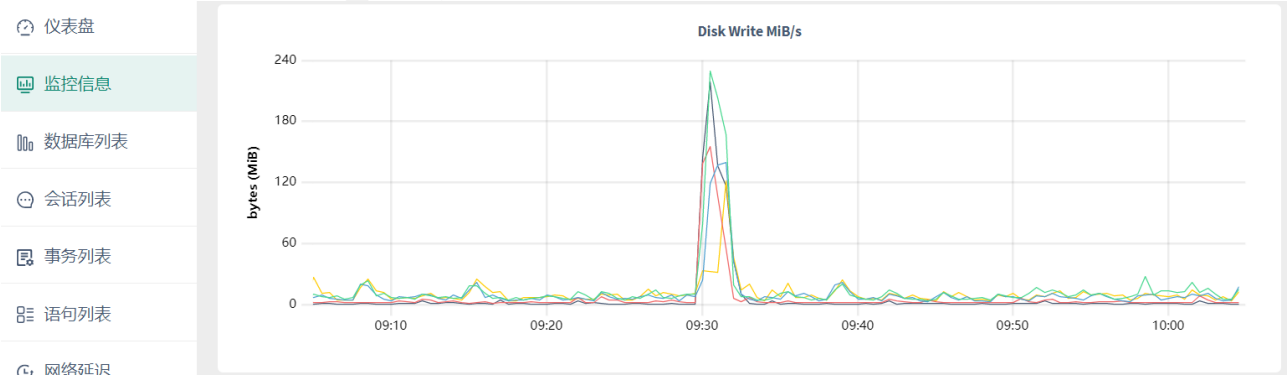
- 在节点视图中，该时间序列图显示了指定节点上内存使用情况
- 在集群视图中，该时间序列图显示了集群中所有节点上内存使用情况。

3.3.4.4.3 磁盘读取字节



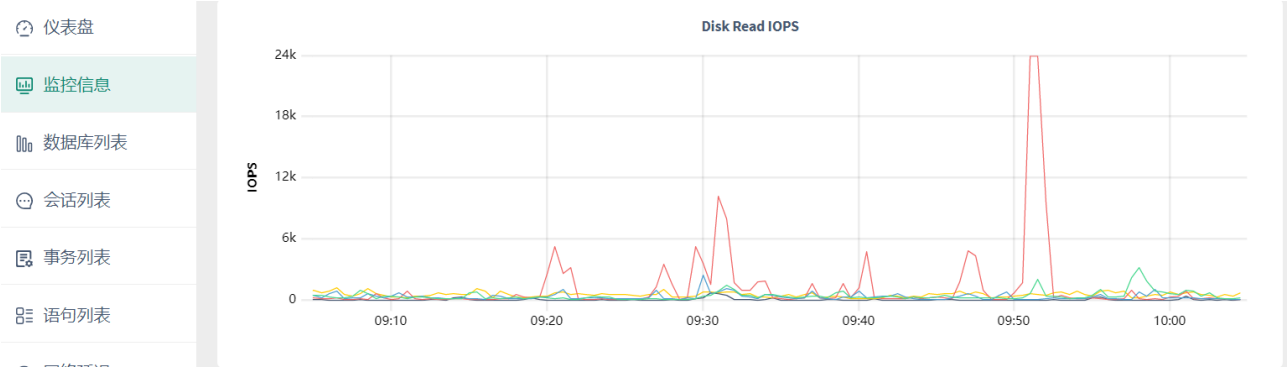
- 在节点视图中，该时间序列图展示的是指定节点上运行的所有进程硬盘读取的速率，采样值为采样周期 10 秒内 rps 的平均值。
- 在集群视图中，该时间序列图展示的是集群中所有节点上运行的所有进程硬盘读取的速率，采样值为采样周期 10 秒内 rps 的平均值。

3.3.4.4.4 磁盘写入字节



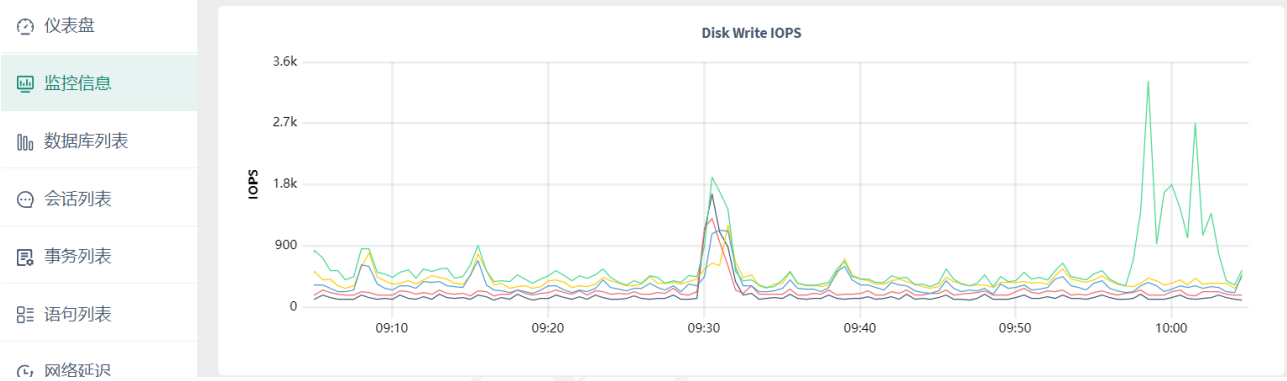
- 在节点视图中，该时间序列图展示的是指定节点上运行的所有进程硬盘写入的速率，采样值为采样周期 10 秒内 rps 的平均值。
- 在集群视图中，该时间序列图展示的是集群中所有节点上运行的所有进程硬盘写入的速率，采样值为采样周期 10 秒内 rps 的平均值

3.3.4.4.5 磁盘读取操作



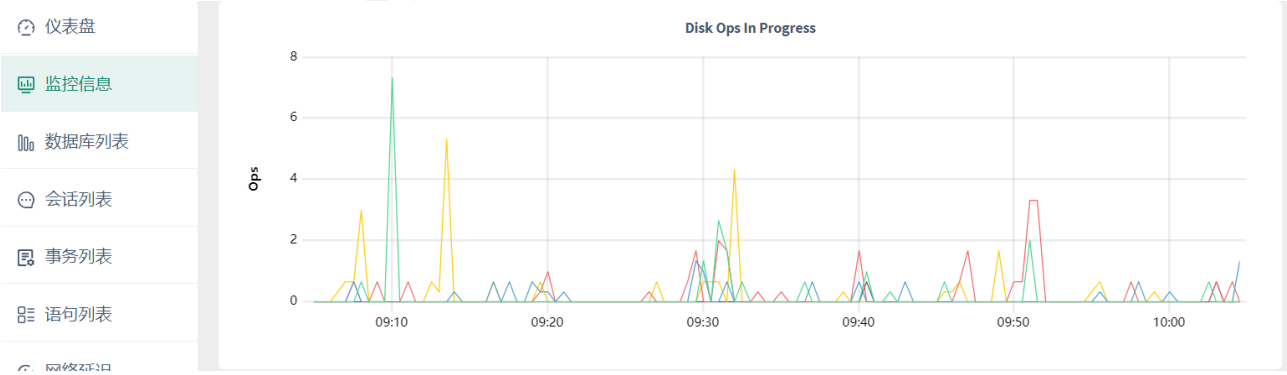
- 在节点视图中，该时间序列图展示的是指定节点上运行的所有进程硬盘读取操作数，采样值为采样周期 10 秒内的平均值。
- 在集群视图中，该时间序列图展示的是集群中所有节点上运行的所有进程硬盘读取操作数，采样值为采样周期 10 秒内的平均值

3.3.4.4.6 磁盘写操作



- 在节点视图中，该时间序列图展示的是指定节点上运行的所有进程硬盘写入操作数，采样值为采样周期 10 秒内的平均值。
- 在集群视图中，该时间序列图展示的是集群中所有节点上运行的所有进程硬盘写入操作数，采样值为采样周期 10 秒内的平均值

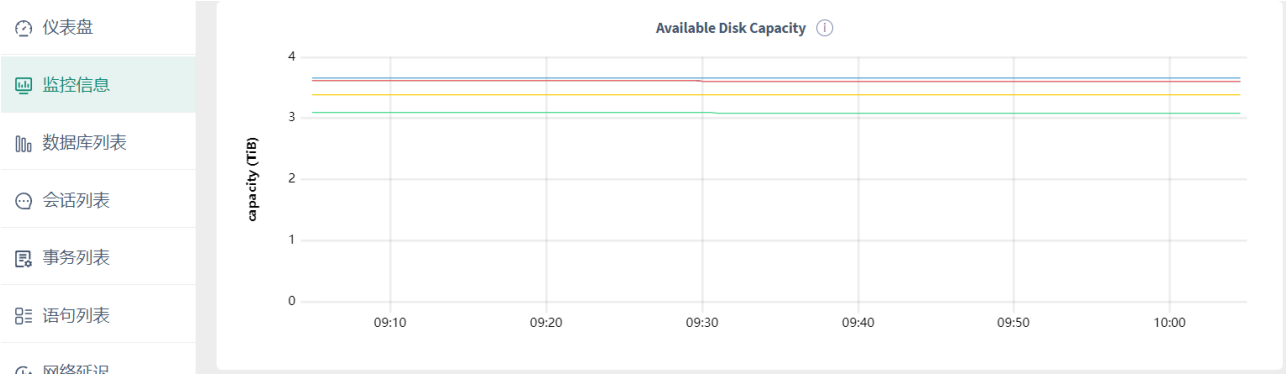
3.3.4.4.7 磁盘 IOPS 进行中



- 在节点视图中，该时间序列图展示的是指定节点上运行的所有进程硬盘读写队列中请求的数量。

在集群视图中，该时间序列图展示的是集群中所有节点上运行的所有进程硬盘读写队列中请求的数量。

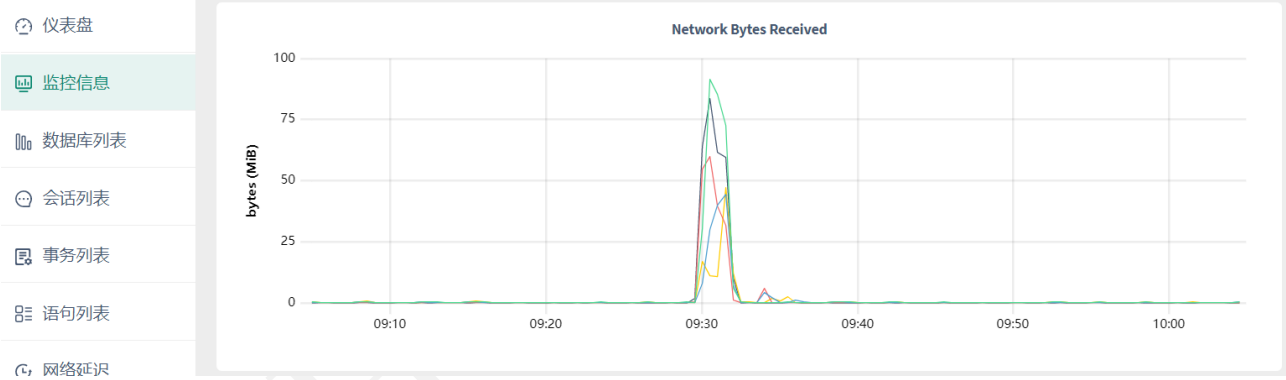
3.3.4.4.8 可用磁盘容量



- 在节点视图上，该时间序列图展示了指定节点可用的存储容量。
- 在集群视图上，该时间序列图展示了集群中所有节点可用的存储容量。

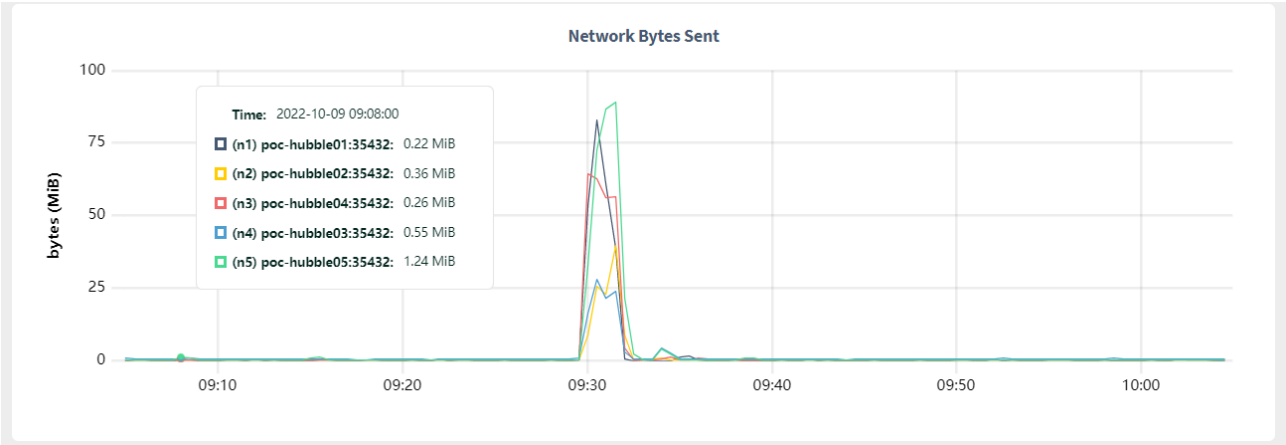
> 注意：如果用户在一台机器上运行多个节点（在生产环境下不推荐这样做）且没有通过`--store`指定每个节点最大分配的存储容量，则 Admin 界面显示的 Capacity 指标数值是不正确的。这是因为当多个节点运行在同一台机器上的时候，运行的每个节点都会把该机器的硬盘整个视作一个可用的存储空间，所有节点的硬盘可用容量的总和等于节点数量乘以硬盘的可用容量。但实际上只有一个物理硬盘。

3.3.4.4.9 收到的网络字节



- 在节点视图上，该时间序列图显示的是指定节点上所有进程的每秒钟接收的网络字节数的总和。
- 在集群视图上，该时间序列图显示的是集群中所有节点上运行的所有进程的每秒钟接收的网络字节数的总和。

3.3.4.4.10 发送的网络字节



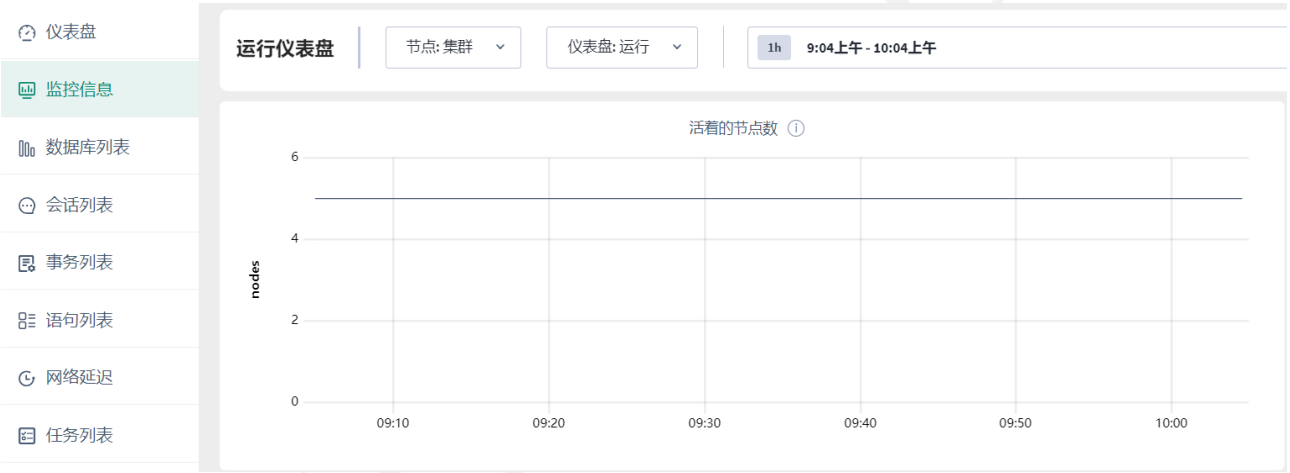
- 在节点视图上，该时间序列图显示的是指定节点上所有进程的每秒钟发送的网络字节数的总和。
- 在集群视图上，该时间序列图显示的是集群中所有节点上运行的所有进程的每秒钟发送的网络字节数的总和。

3.3.4.5 运行时仪表板

通过运行仪表板可以获取集群运行时的指标，例如节点数，内存使用情况和 CPU 时间。要查看此仪表板，请访问管理界面，单击左侧导航栏上的监控信息，然后选择仪表板 > 运行。

运行时仪表板显示以下时间序列图：

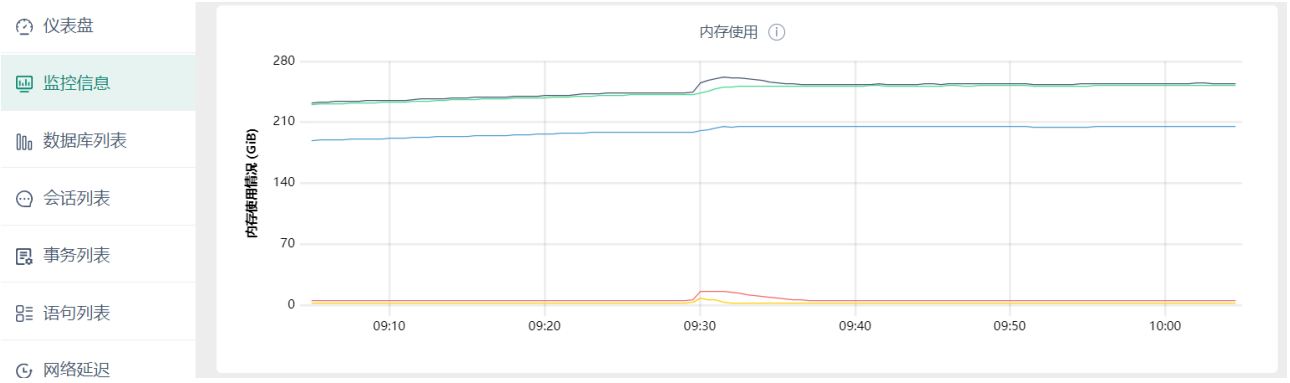
3.3.4.5.1 活动节点数



在节点视图和集群视图中，该图显示了集群中活动节点的数量。

该图中的曲线的下降表示存在退役节点或宕机节点。

3.3.4.5.2 内存使用情况



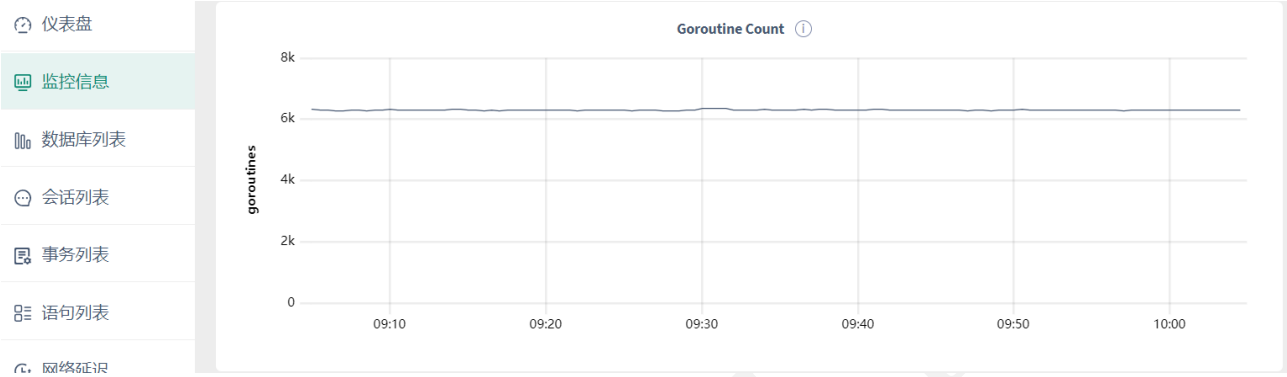
- 在节点视图中，该时间序列图显示的是指定节点的内存使用量。
- 在集群视图中，该时间序列图显示的是集群中所有节点的内存使用量。

将鼠标悬停在图形上时，将显示以下指标的值：

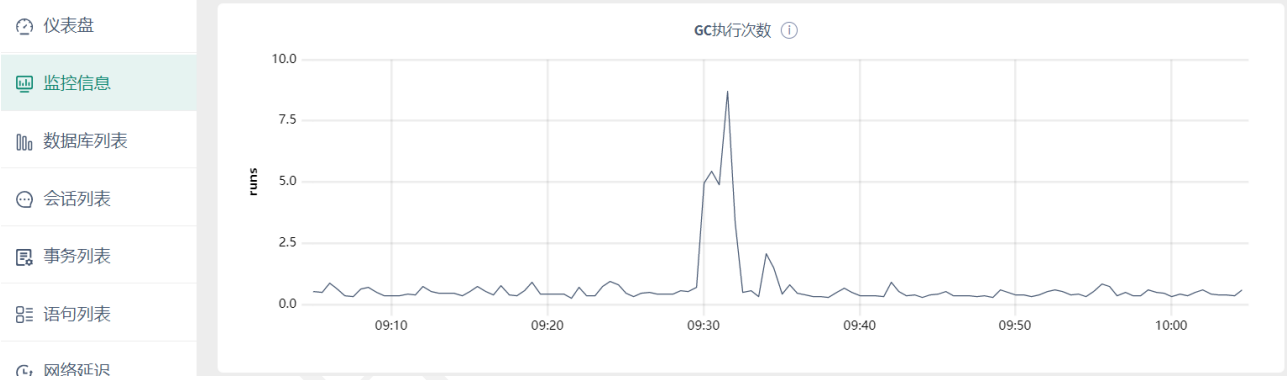
HUBBLE

指标	描述
RSS	整体内存使用量
Go Allocated	Go 分配的内存。
Go Total	Go 管理的内存。
CGo Allocated	C 分配的内存。
CGo Total	C 管理的内存。

3.3.4.5.3 并发数

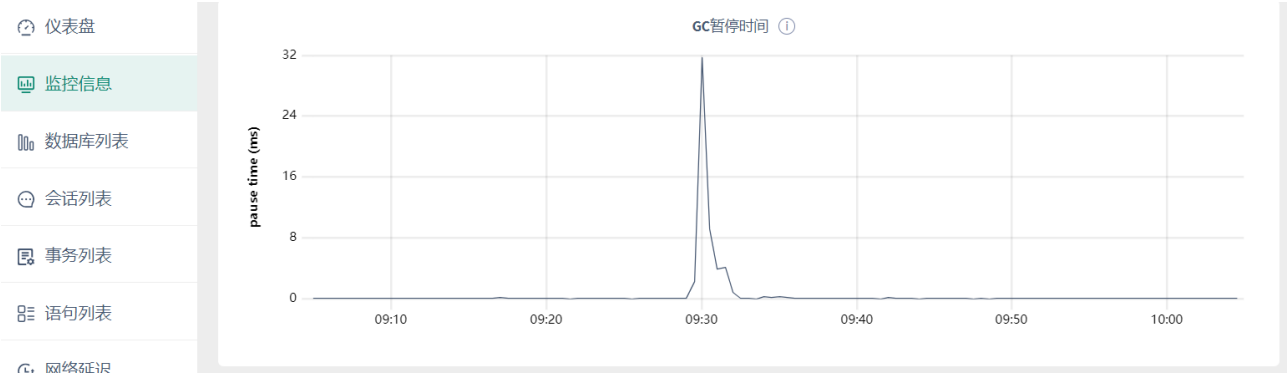


3.3.4.5.4 GC 运行数



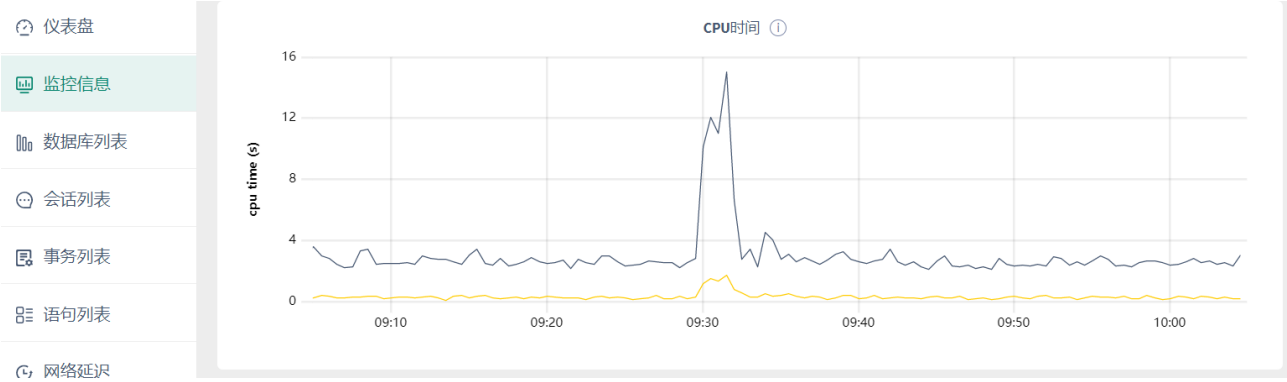
GC运行数可以理解为每秒调用垃圾收集器的次数。

3.3.4.5.5 GC 暂停时间



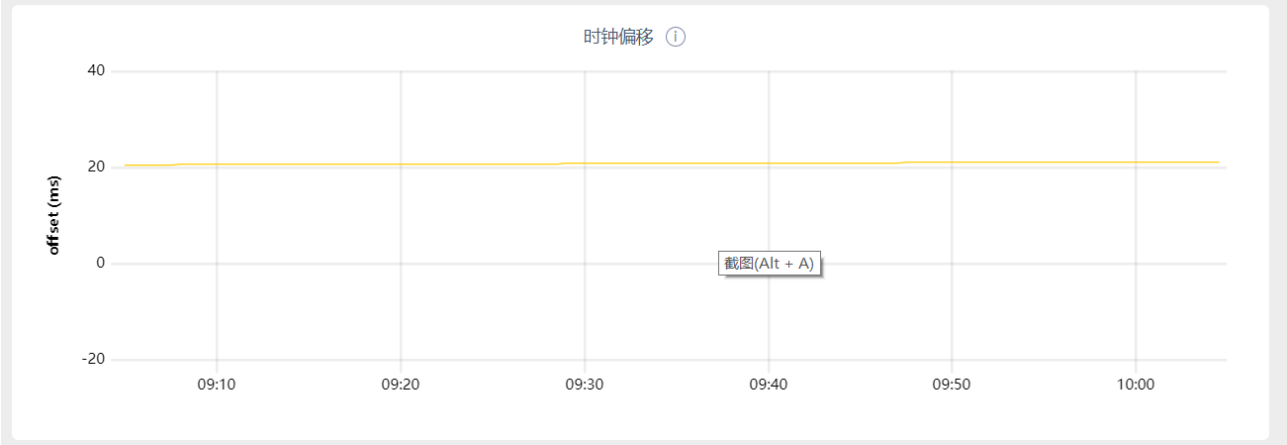
GC暂停时间可以理解为垃圾收集器每秒使用的处理器时间，执行垃圾回收时，数据库执行将暂停。

3.3.4.5.6 CPU 时间



- 在节点视图上，该时间序列图显示的是指定节点上所有进程和操作相关系统级的 CPU 时间。
- 在集群视图上，该时间序列图显示的是集群中所有节点上运行的所有进程和操作相关系统级的 CPU 时间。

3.3.4.5.7 时钟偏移



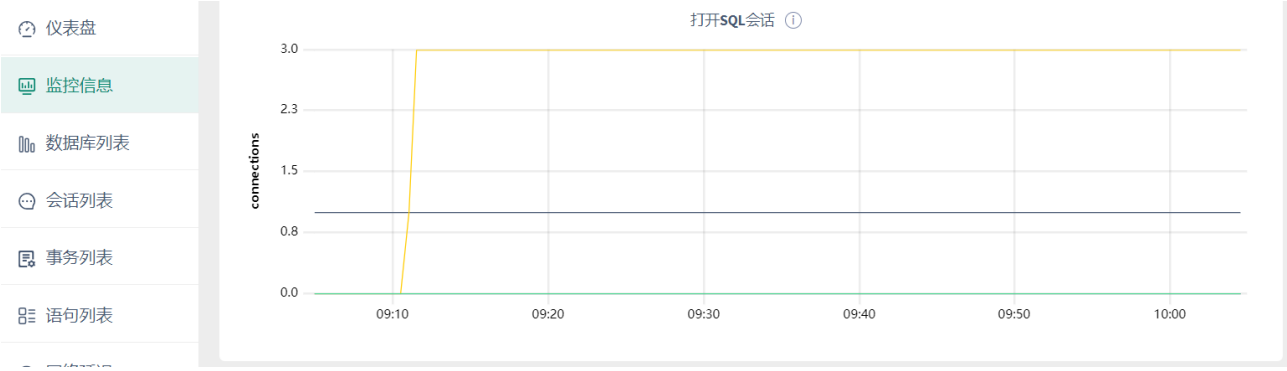
- 在节点视图中，该时间序列图显示的是指定节点与集群其他节点的时钟偏差值的平均值。
- 在集群视图中，该时间序列图显示的是集群中每个节点与集群其他节点的时钟偏差值的平均值。

3.3.4.6 SQL 仪表板

通过管理界面中的 SQL 仪表板，可以帮助用户监控 SQL 查询的性能。要查看此仪表板，请访问管理界面，单击左侧导航栏上的监控信息，然后选择仪表板 >SQL。

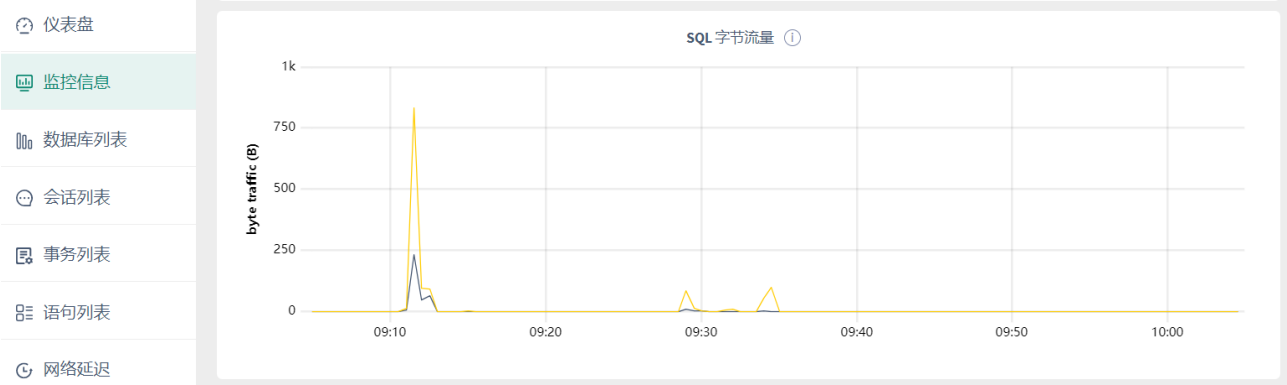
SQL 仪表板显示以下时间序列图：

3.3.4.6.1 SQL 连接数



- 在节点视图中，该时间序列图显示的是指定节点和客户端之间打开的 SQL 连接数量。
- 在集群视图中，该时间序列图显示的是所有节点和客户端之间打开的 SQL 连接数量的总和。

3.3.4.6.2 SQL 字节流量



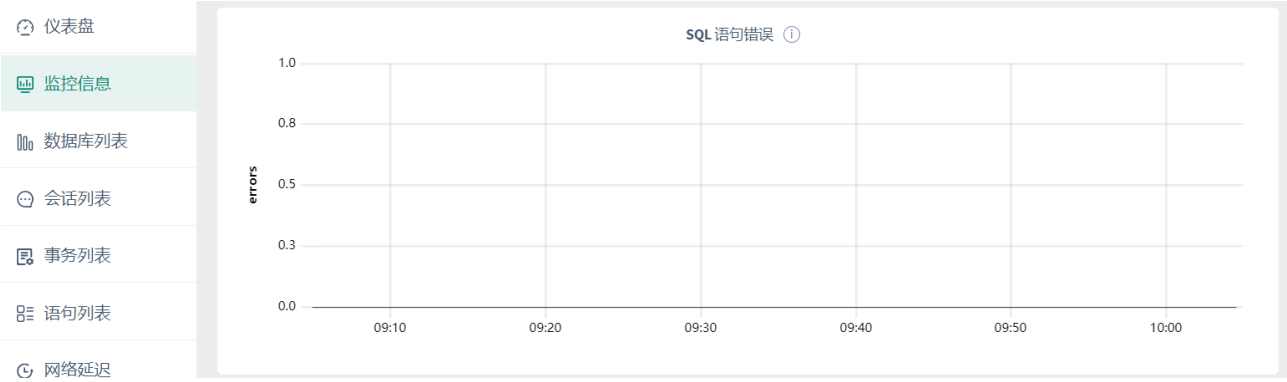
SQL 字节流量图能够帮助用户关联 SQL 查询数量和字节流量，特别适合监控批量数据插入或是返回大量数据的分析型查询。

- 在节点视图中，该时间序列图显示的是指定节点与所有连接的 SQL 客户端之间的字节吞吐量。
- 在集群视图中，该时间序列图显示的是集群中所有节点与连接的 SQL 客户端之间的字节吞吐量的总和。

3.3.4.6.3 SQL 查询数

- 在节点视图中，该时间序列图显示的是指定节点处理客户端请求的 QPS，查询包括SELECT/INSERT/UPDATE/DELETE语句。
- 在集群视图中，该时间序列图整合了每个节点统计信息。具体来说，将每个节点最近 10 秒的活动情况进行汇总，结果视作当前集群查询负载的估计值。

3.3.4.6.4 SQL 查询失败数



- 在节点视图中，该时间序列图显示的是指定节点返回错误的 SQL 查询数量情况。
- 在集群视图中，该时间序列图显示的是所有节点返回错误的 SQL 查询数量情况。

3.3.4.6.5 事务数

- 在节点视图中，该图显示了节点上 SQL 客户端每秒发出的已打开，已提交，中止和回滚的事务数的 10 秒平均值。
- 在集群视图中，该图显示了每个节点平均值的总和，即集群中当前事务负载的汇总估计，假设每个节点的活动的最后 10 秒代表此负载。

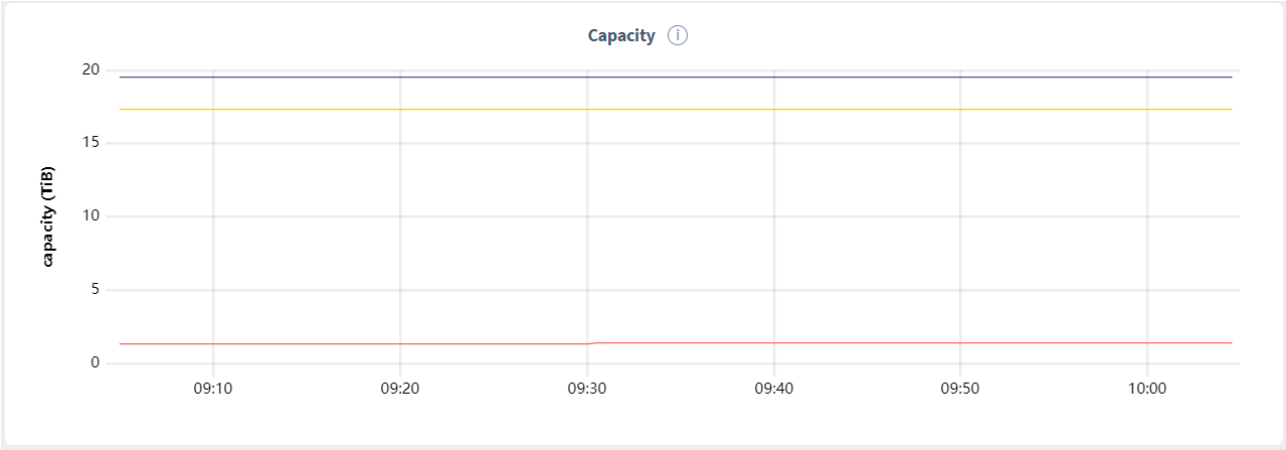
如果图形显示过多的异常终止或回滚，则可能表明 SQL 查询存在问题。在这种情况下，请重新检查查询以降低争用。

3.3.4.7 存储仪表板

通过存储仪表板，您可以监视集群的存储空间的使用率。要查看此仪表板，请访问管理界面，单击左侧导航栏上监控信息，然后选择仪表板 > 存储。

存储仪表板显示以下时间序列图：

3.3.4.7.1 容量



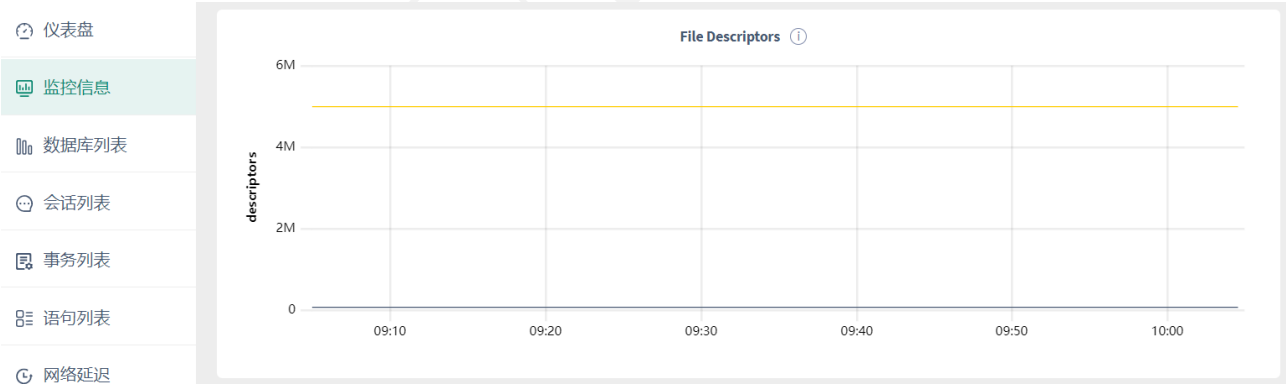
您可以监视容量图以判断何时需要添加新的存储空间。

- 在节点视图中，该时间序列图展示了在集群中指定节点的最大分配容量、可用容量和已使用容量的情况。
- 在集群视图中，该时间序列图展示了集群中所有节点的最大分配容量总和、可用容量总和和已使用容量总和

将鼠标悬停在图形上时，将显示以下指标的值：

指标	描述
容量	分配给数据库的存储容量。您可以为指定节点通过`--store`配置可用最大存储容量。
可用	可用的空闲存储容量。
已用	已使用的磁盘空间。请注意，此值小于容量-可用，因为容量和可用指标统计的是整个硬盘，受硬盘上的所有程序的影响，已用指标只统计已存储的磁盘使用情况。

3.3.4.7.2 文件描述符



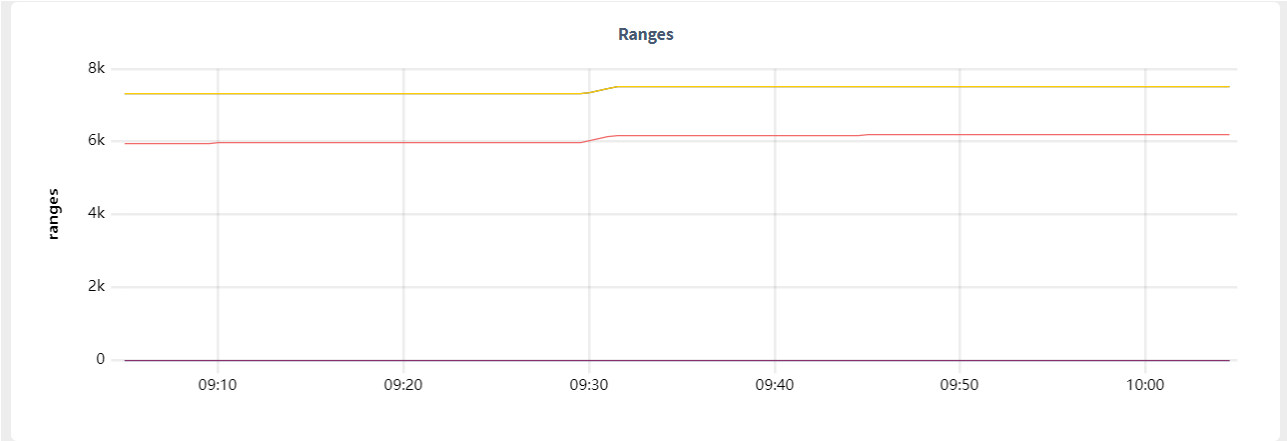
- 在节点视图中，该时间序列图显示了指定节点上已经打开的文件描述符数量，以及系统允许的文件描述符数量的上限值。
- 在集群视图中，该时间序列图显示了集群中每个节点已经打开的文件描述符数量，以及系统允许的文件描述符数量的上限值。

如果打开的文件描述符数量等于系统允许的上限值，则用户需要增加文件描述符上限。

3.3.4.8 复制仪表板

通过管理界面中的复制仪表板，您可以监视集群的复制指标。要查看此仪表板，请访问管理界面，单击左侧导航栏上的监控信息，然后选择仪表板 > 复制。

3.3.4.8.1 分片



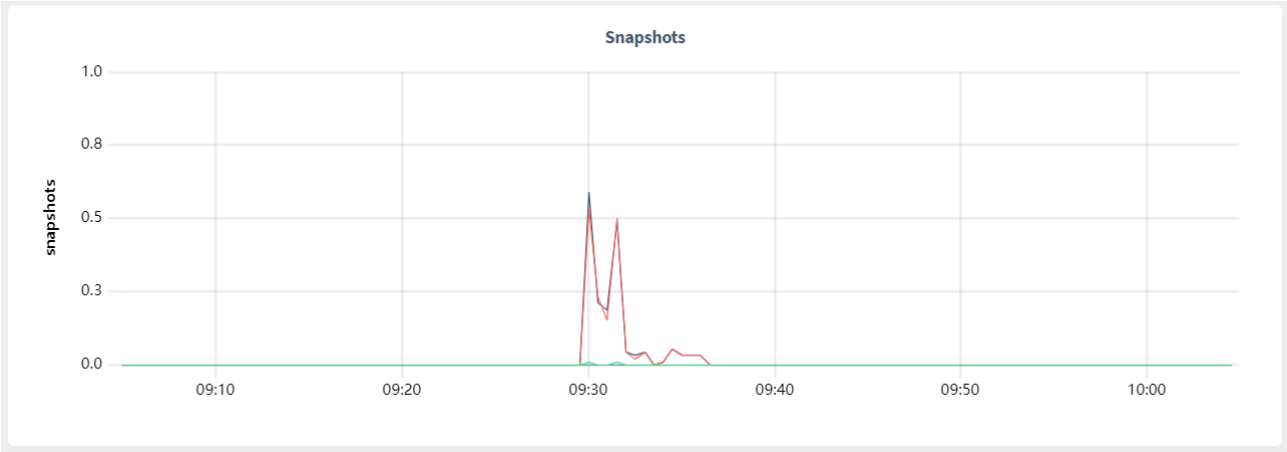
分片图显示有关范围状态的各种详细信息。

- 在节点视图中，该时间序列图显示的是分片副本数量。
- 在集群视图中，该时间序列图显示的是分片副本数量。

将鼠标悬停在图形上时，将显示以下指标的值：

指标	简介
分片	该节点持有的分片数量
Leaders	拥有 Leader 的分片数量。如果一个节点上拥有 Leader 的分片数量与节点持有的分片数量在很长时间内都不匹配，则需要进行故障定位追踪。
Lease Holders	持有租约的分片数量。
Leaders w/o Leases	没有租约的 Raft leaders 数量。如果该指标的数值在很长时间内非零，则需要进行故障定位追踪。
Unavailable	不可用的分片数量，如果该指标的数值在很长时间内非零，则需要进行故障定位追踪。
Under-replicated	正在复制的分片数量。

3.3.4.8.2 快照数



通常情况下 Raft 组里的节点会通过相互之间传递 Raft 日志消息的方式来保持同步。然而当一个节点需要同步的日志落后太多，相比起发送所有引起分片变更的消息，集群会直接发送分片的快照，落后节点在应用快照后重新开始同步。大多数情况下这是一个主动推的过程，集群会在认为某个节点需要跟上同步进度的时候推送快照。而在少数情况下，节点会根据 Raft 协议请求快照同步。

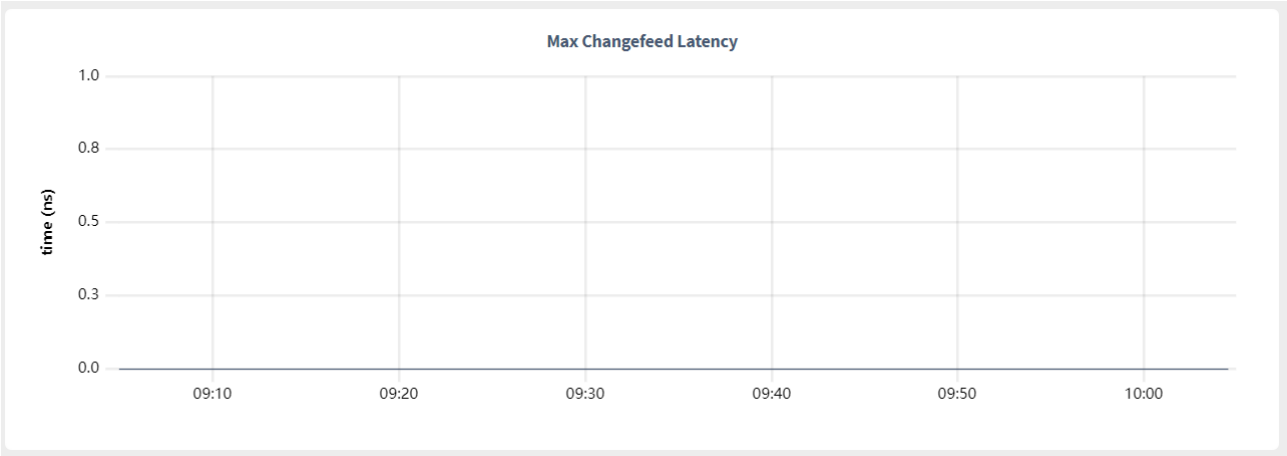
指标	描述
Generated	每秒创建的快照数量
Applied (Raft-initiated)	Raft 中每秒应用于节点的快照数量。
Applied (Learner)	每秒提前应用于节点的快照数量。
Applied (Preemptive)	每秒提前应用于节点的快照数量
Reserved	每秒为将要发送到节点的传入快照保留的插槽数。

3.3.4.9 数据变动仪表板

通过数据变动仪表板，您可以监视在整个集群中创建的数据变动。要查看此仪表板，请访问管理界面，单击左侧导航栏上的监控信息，然后选择仪表板 > 数据变动。

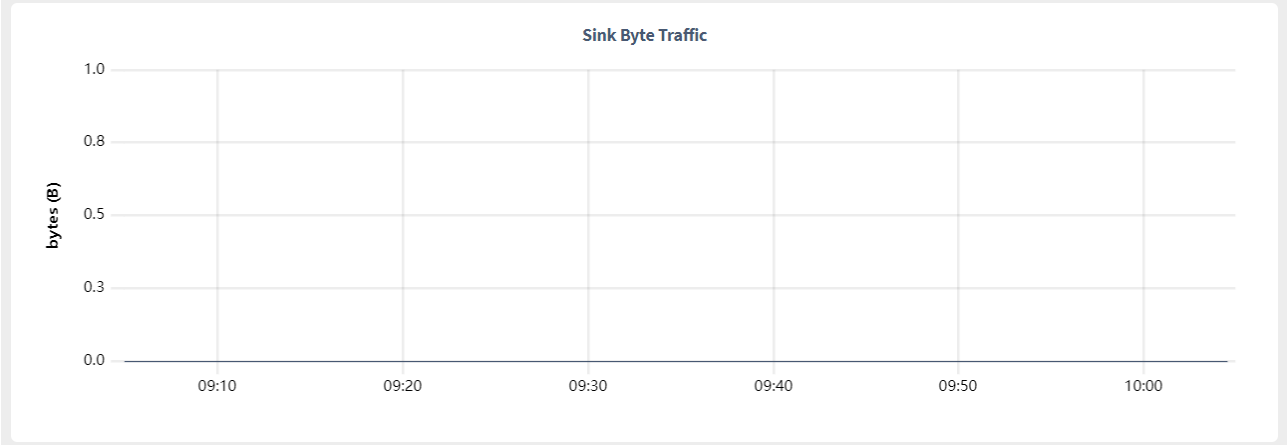
数据变动仪表板显示以下时间序列图：

3.3.4.9.1 最大数据变动延迟



- 在节点视图中，该时间序列图显示了该节点任何正在运行的变更源的已解决时间戳的最大延迟。
- 在集群视图中，该时间序列图显示了所有节点上任何正在运行的变更源的已解决时间戳的最大延迟。

3.3.4.9.2 接收字节流量

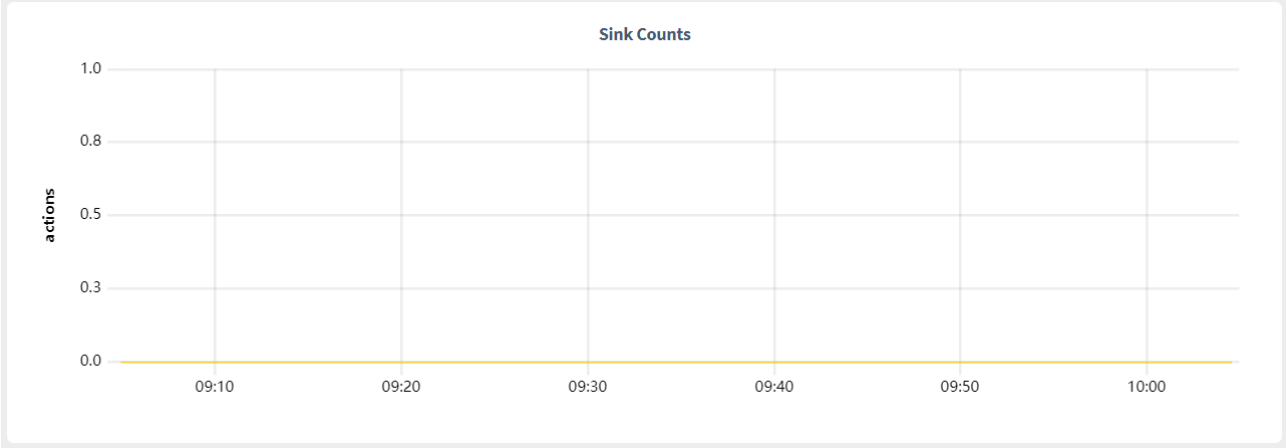


- 在节点视图中，该图显示了在选定节点的所有数据变动中发射到接收器中的字节数。
- 在集群视图中，该图显示了在集群中的所有数据变动和所有节点上发射到接收器中的字节数。

将鼠标悬停在图形上时，将显示以下指标的值：

指标	描述
发射字节	数据库发出的所有数据变动接收器中发送的字节数。

3.3.4.9.3 接收数

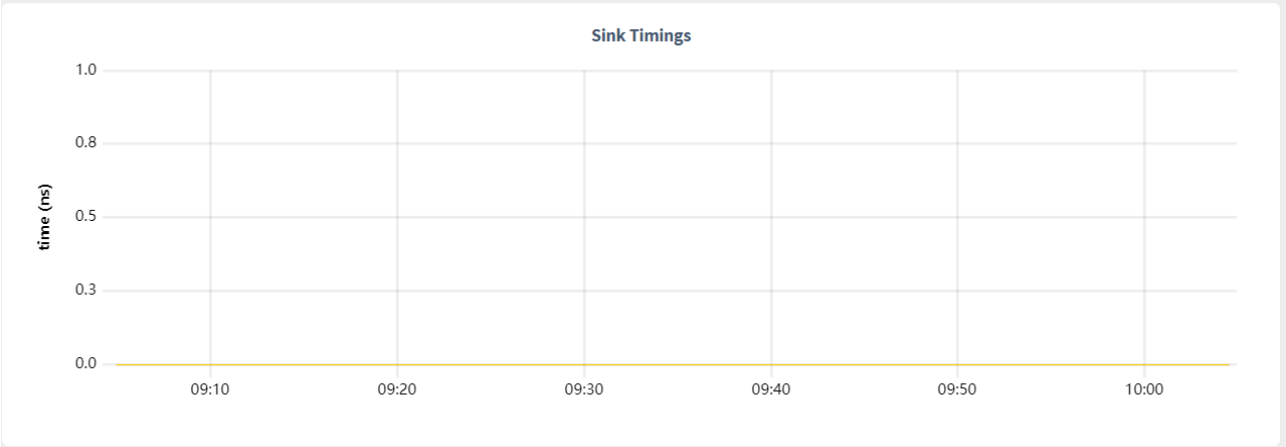


- 在节点视图中，该时间序列图显示数据库发送到接收器的消息数，以及接收器针对所有数据变动执行的刷新次数
- 在集群视图中，该时间序列图显示数据库发送到接收器的消息数，以及接收器对集群中所有变更源执行的刷新次数。

将鼠标悬停在图形上时，将显示以下指标的值：

指标	Description
Messages	发送到所有数据变动的接收器的消息数。
Flushes	接收对所有数据变动执行的刷新次数。

3.3.4.9.4 接收计时



- 在节点视图中，该时间序列图显示了数据库将消息发送到接收器所需的时间（以毫秒为单位），以及等待接收器刷新所有数据变动的消息所花费的时间。
- 在集群视图中，该时间序列图显示了数据库将消息发送到接收器所需的时间（以毫秒为单位），这是等待接收器刷新集群中所有数据变动的消息所花费的时间。

将鼠标悬停在图形上时，将显示以下指标的值：

指标	描述
信息发射时间	将所有数据变动发送到接收器所需的时间。
刷新时间	等待接收器刷新所有数据变动消息所花费的时间。

3.3.5 数据库列表

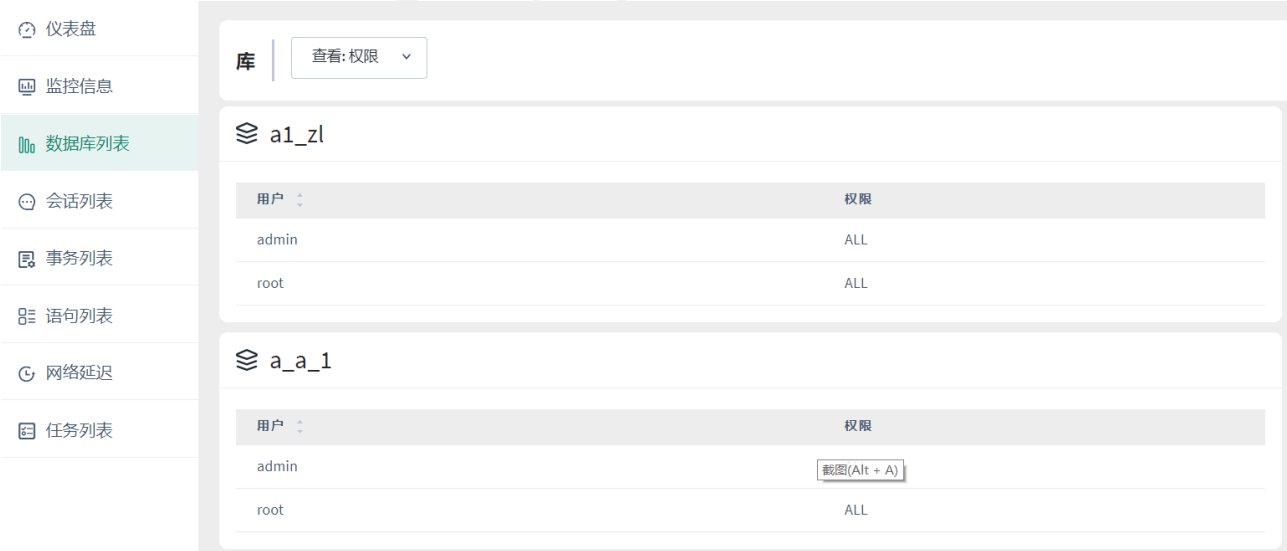
管理员界面的数据库列表提供了配置的数据库信息，每个数据库中的表以及分配给每个用户的授权的详细信息。要查看这些详细信息，请访问管理界面，然后单击左侧导航栏上的数据库列表。

3.3.5.1 表视图

表视图显示系统表以及数据库中的表的详细信息。要查看这些详细信息，请访问管理界面，然后从左侧导航栏中选择数据库列表，然后从VIEW菜单中选择'表'。

3.3.5.2 权限视图

权限视图显示用户在每个数据库当中被授予的权限。要查看这些详细信息，请访问管理界面，从左侧导航栏中选择数据库列表，然后从VIEW菜单中选择'权限'。



3.3.6 任务列表

展示了备份恢复任务和Schema变更作业的细节信息，包括 ID、描述信息、相关用户、创建时间以及状态。用户可以点击每行第一列的下拉按钮，查看更加丰富的信息内容.

3.3.6.1 任务细节

任务列表页面显示集群中所有节点的每个备份和还原作业的 ID，描述，用户，创建时间和状态，架构更改，用户创建的表统计信息和自动表统计信息作业以及执行的数据更改。要查看作业的完整说明，请单击第一列中的下拉箭头。

3.3.6.2 过滤结果

您可以根据作业的状态或作业的类型（备份，还原，导入，变更Schema，数据变动，创建统计或自动统计）过滤结果。您也可以选择查看最新的 50 个作业或所有节点上的所有作业。

过滤条件	描述
任务状态	从状态菜单中，选择所需的状态过滤器。
任务类型	从类型菜单中，选择备份，还原，导入，变更Schema，数据变动，创建统计或自动统计。
任务展示	在显示菜单上，选择前 50 个或全部。

3.3.7 语句列表

语句列表页面能够帮助用户查看热点查询以及高时延的 SQL 语句。用户还能够点击一个单独的 SQL 语句，在语句详情页中查看到该语句的详细信息。

要查看语句列表页面，请访问管理界面，然后单击左侧的语句列表。

3.3.7.1 有效期

语句列表页面显示的是在指定的时间窗口内执行的 SQL 语句的细节信息，显示的内容将周期性地被擦除。擦除后，在执行下一组语句之前，用户将不会在页面上查看到任何语句。默认情况下，时间间隔设置为一小时。用户可以通过修改集群配置项diagnostics.reporting.interval来自定义时间间隔。

3.3.7.2 按应用过滤

如果用户在集群上运行了多个应用程序，语句列表页在默认情况下能够显示来自所有应用程序的语句。如果用户需要查询某个应用程序有关的所有语句，可以点击下拉式菜单应用，选择指定的应用程序。

3.3.7.3 参数

语句列表页面显示的是每个 SQL 语句的总执行时间、执行计数、重试次数、受影响行、延时等信息。默认情况下，语句指纹按时间排序；但是，您可以按执行计数，重试，受影响的行和延迟对表进行排序。

为每个语句提供以下详细信息：

参数	描述
语句	SQL 语句。
事务类型	事务类型（显式或者隐式）。显式事务是指客户端由BEGIN和COMMIT语句包装的语句。对于不在显式事务中的语句，数据库将每个语句包装在单独的隐式事务中。显式事务使用事务流水线，因此报告的延迟不考虑副本。
时间	在最后一个小时或自定义时间间隔内，一个 SQL 语句执行时间的总和。
执行次数	最近 1 小时或是自定义时间间隔内，一个 SQL 语句执行计数。执行计数将以数值和水平条形图的形式显示。条形图用颜色标记，表示执行计数中执行成功（用蓝色表示）与执行失败（用红色表示）的比率。
重试次数	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句重试的累计计数。
受影响行	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句返回行数的平均值。该指标以数值和水平条形图的形式显示。条形图用颜色标记，蓝色代表返回行数的平均值，黄色代表返回行数的标准差。
延迟	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句服务时延的平均值。该指标以数值和水平条形图的形式显示。条形图用颜色标记，蓝色代表服务时延的平均值，黄色代表服务时延的标准差。

3.3.7.4 语句明细页面

语句详细显示的是指定语句指纹具体的执行时间、执行次数、返回行数、各个阶段的时延以及网关节点的时延等详细信息。选中想要查看的 sql 点击进去。

3.3.7.4.1 逻辑计划

概况　诊断　逻辑计划　执行状态

逻辑计划

```
Filter
  filter = (claim_session_id = _) AND (claim_instance_id = _)
Index join
  table = jobs@primary
Scan
  table = jobs@jobs_status_created_idx
  missing stats
  spans = 2 spans
```

3.3.7.4.2　执行状态（按阶段执行延迟）

阶段	均值 Latency	标准差
解析	157.6 µs	469.3 µs
计划	110.6 µs	196.0 µs
运行	91.4 µs	110.7 µs
额外开销	5.1 µs	1.7 µs
全部时间	364.7 µs	527.1 µs

阶段图表提供了的 SQL 语句服务时延的平均值和标准差，同时也提供了各个执行阶段（解析、计划、执行）时延的平均值和标准差。该指标以数值和水平条形图的形式显示。条形图用颜色标记，蓝色代表服务时延的平均值，黄色代表服务时延的标准差。

3.3.7.4.3　执行状态（按节点统计）

	执行次数	读取行数	读取字节数	语句 时间	连接	最大内存	网络使用	重试次数
(n2) poc-hubble02:35432	315	0	0 B	228.8 µs	0.0 ns	0 B	0 B	0
(n4) poc-hubble03:35432	11	0	0 B	332.4 µs	0.0 ns	0 B	0 B	0

提供了指定语句在每个网关节点上详细的数值指标。关键的参数如下:

参数	描述
节点	网关节点 ID
时间	在最近 1 小时或是自定义时间间隔内，SQL 语句的累计执行时间
执行次数	一个 SQL 语句执行的总次数
重试次数	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句重试次数的累计值。

参数	描述
受影响行	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句返回行数的平均值。该指标以数值和水平条形图的形式显示。条形图用颜色标记，蓝色代表返回行数的平均值，黄色代表返回行数的标准差。
延时	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句服务时延的平均值。该指标以数值和水平条形图的形式显示。条形图用颜色标记，蓝色代表服务时延的平均值，黄色代表服务时延的标准差。

3.3.7.4.4 概况（执行次数）

执行次数

第一次尝试	26
总执行次数	26
重试	0
最大重试次数	0

'执行计数'表提供有关以下参数的信息，包括数值和条形图：

Parameter	Description
First Attempts	在最后一个小时或指定的时间间隔内执行 SQL 语句的首次尝试的累积次数。
Retries	在最后一个小时或指定的时间间隔内执行 SQL 语句的重试累积次数。
Max Retries	在最后一个小时或指定的时间间隔内，使用此指纹的单个 SQL 语句的最大重试次数。
Total	具有此结构的语句的执行总数。它计算为首次尝试和累积重试的总和。

3.3.7.4.5 概况（语句详细信息）

统计信息框下方的表格提供以下详细信息：

HUBBLE

语句详细信息

应用	$ internal-select-running/get-claimed-jobs
失败了?	No
使用基于成本的优化器?	Yes
分布式执行?	No
向量化执行?	No
事务类型	Implicit

参数	简介
应用	应用程序通过session配置指定的名称。
事务类型	事务类型（显性或者隐性）。
是否分布式执行	指示语句执行是否是分布式的。
向量化执行	表明该语句是否向量化执行
使用基于成本的优化器	表明该语句是否使用了基于代价的优化器。
失败	指示该语句是否已成功执行。

3.3.7.4.6 概况（资源使用）

资源使用

读取的平均行数/字节数	0.00 / 0 B
最大内存使用量	0 B
网络使用	0 B
最大暂存盘使用量	0 B

参数	简介
读取的平均行数字节数	读取字节占用的情况
最大内存使用量	执行语句用的最大内存
网络使用	指示语句执行网络使用情况
最大暂存盘使用量	表明该语句最大暂存盘使用量

3.4 日常巡检

3.4.1 巡检意义

持续保证数据库稳定运行，在第一时间发现问题，为后续的持续改进提供依据。

3.4.2 巡检范围

巡检适用于 hubble 数据库的监控，对数据库的配置和性能进行了分析，在一定程度上检验了数据库的安全问题及其相关细节的分析。总体可以分为配置信息管理，故障监控，性能检测等。

3.4.3 登录的网址

每个项目有所不同，以 ip 加端口的形式登录，例如`https://172.16.2.161:58080`

3.4.4 页面服务

3.4.4.1 概况

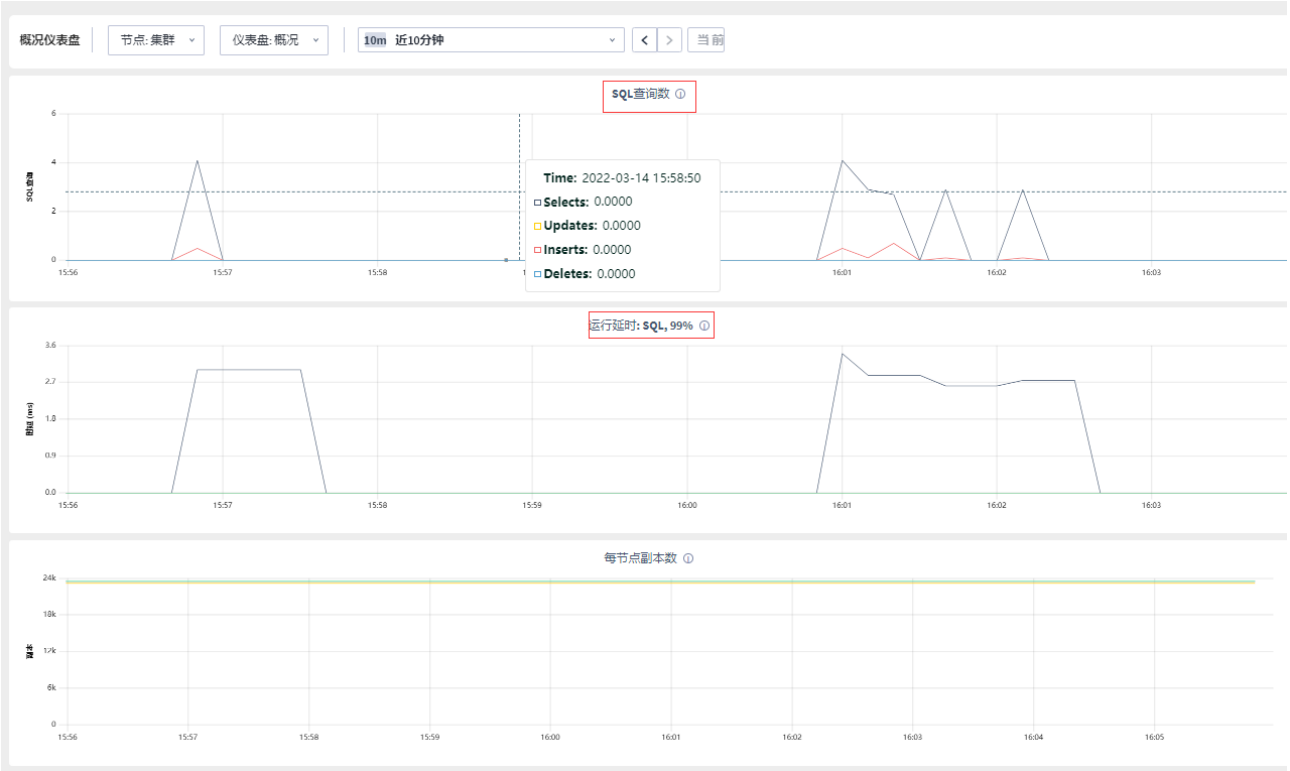
查看仪表盘，如下图：

上图展示了节点状态/分片状态/存储/内存使用率

> 当出现节点状态可疑或者宕机，以及分片状态出现未复制或者不可用则需要重点引起关注

3.4.4.2 监控

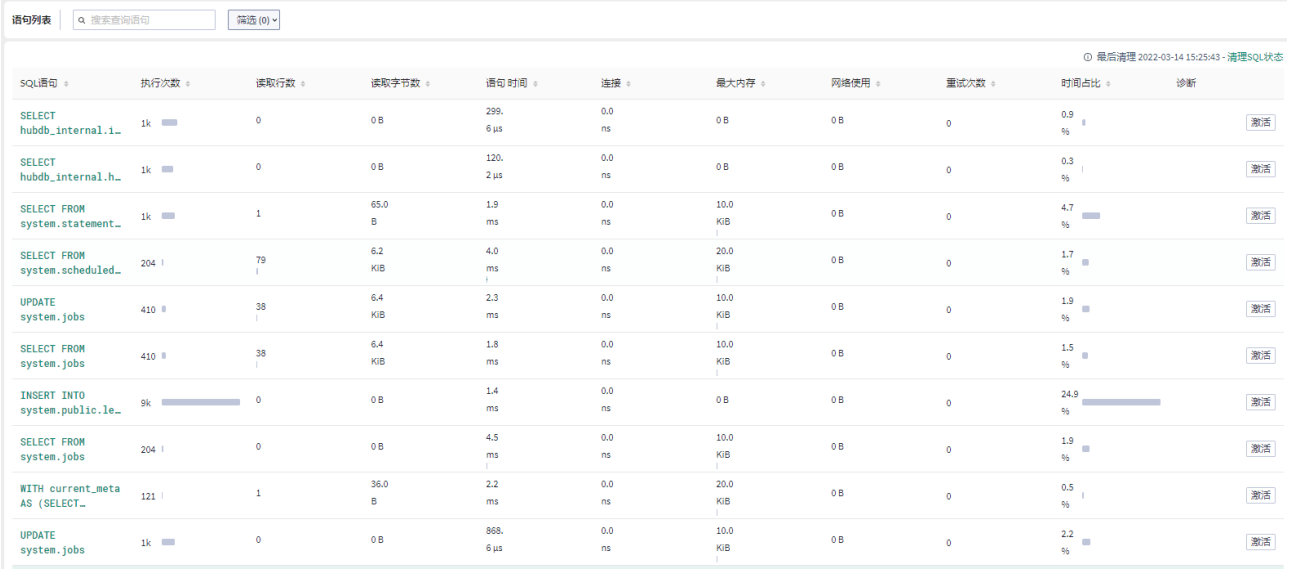
查看监控信息



关注 SQL 运行延时，如果延时时间都是大于 3 秒的需要进行关注

3.4.4.3 语句

查看语句列表



SQL语句	执行次数	读取行数	读取字节数	语句 时间	连接	最大内存	网络使用	重试次数	时间占比	诊断	
SELECT hubdb_internal.i…	1k	0	0 B	299.6 µs	0.0 ns	0 B	0 B	0	0.9 %		激活
SELECT hubdb_internal.h…	1k	0	0 B	120.2 µs	0.0 ns	0 B	0 B	0	0.3 %		激活
SELECT FROM system.statement…	1k	1	65.0 B	1.9 ms	0.0 ns	10.0 KiB	0 B	0	4.7 %		激活
SELECT FROM system.scheduled…	204	79	6.2 KiB	4.0 ms	0.0 ns	20.0 KiB	0 B	0	1.7 %		激活
UPDATE system.jobs	410	38	6.4 KiB	2.3 ms	0.0 ns	10.0 KiB	0 B	0	1.9 %		激活
SELECT FROM system.jobs	410	38	6.4 KiB	1.8 ms	0.0 ns	10.0 KiB	0 B	0	1.5 %		激活
INSERT INTO system.public.le…	9k	0	0 B	1.4 ms	0.0 ns	0 B	0 B	0	24.9 %		激活
SELECT FROM system.jobs	204	0	0 B	4.5 ms	0.0 ns	10.0 KiB	0 B	0	1.9 %		激活
WITH current_meta AS (SELECT…	121	1	36.0 B	2.2 ms	0.0 ns	20.0 KiB	0 B	0	0.5 %		激活
UPDATE system.jobs	1k	0	0 B	868.6 µs	0.0 ns	10.0 KiB	0 B	0	2.2 %		激活

关注执行的平均时间是否有大于 3 秒的语句，如果有重点关注

3.4.4.4 任务

查看任务列表

描述	任务ID	用户	创建时间	状态
BACKUP DATABASE cdc_demo INTO '/2022/03/15-0000...	744605693688446981	root	2022-03-15 09:00:24	SUCCEEDED 耗时: 00:00:00
GC for DROP TABLE xuejs.public.tmp_hubble_0623e31f6...	744604774892044289	hubble	2022-03-15 08:55:43	Waiting For GC TTL 0.0 %
DROP TABLE xuejs.public.tmp_hubble_0623e31f6efd4d0...	744604774740590593	hubble	2022-03-15 08:55:43	SUCCEEDED 耗时: 00:00:00
CREATE TABLE xuejs.public.tmp_hubble_0623e31f6efd4...	744604652691718145	hubble	2022-03-15 08:55:06	SUCCEEDED 耗时: 00:00:00
BACKUP DATABASE cdc_demo INTO '/2022/03/15-0000...	744593895003127813	root	2022-03-15 08:00:23	SUCCEEDED 耗时: 00:00:00
BACKUP DATABASE cdc_demo INTO '/2022/03/14-0000...	744593895145177093	root	2022-03-15 08:00:23	SUCCEEDED 耗时: 00:00:00
BACKUP DATABASE cdc_demo INTO '/2022/03/14-0000...	744582096509140997	root	2022-03-15 07:00:22	SUCCEEDED 耗时: 00:00:00
BACKUP DATABASE cdc_demo INTO '/2022/03/14-0000...	744570298065125381	root	2022-03-15 06:00:22	SUCCEEDED 耗时: 00:00:00
BACKUP DATABASE cdc_demo INTO '/2022/03/14-0000...	744558499550134277	root	2022-03-15 05:00:21	SUCCEEDED 耗时: 00:00:00
BACKUP DATABASE cdc_demo INTO '/2022/03/14-0000...	744546701090488325	root	2022-03-15 04:00:21	SUCCEEDED 耗时: 00:00:00

任务列表 状态: 全部 类型: 全部 显示: 最新 50

> 关注任务是否有较多的进行中的

3.4.5 异常处理检测

3.4.5.1 节点实例查看

```
hubble node status --certs-dir=/var/lib/hubbletp/certs --host=poc-hubble01:35432
```

```
[hubble@poc-hubble01 ~]$ hubble310 node status --certs-dir=/var/lib/hubbletp310/certs --host=poc-hubble01:35432
  id |       address       |     sql_address     |     build     |         started_at         |         updated_at         |                          locality                          | is_available
| is_live
-----+---------------------+---------------------+---------------+----------------------------+----------------------------+-------------------------------------------------------------+-------------
+----------
   1 | poc-hubble01:35432 | poc-hubble01:35432 | v3.10.2-dirty | 2022-03-10 10:25:08.028497 | 2022-03-15 01:29:33.099485 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=1 | true
| true
   2 | poc-hubble02:35432 | poc-hubble02:35432 | v3.10.2-dirty | 2022-03-10 10:12:56.868096 | 2022-03-15 01:29:35.201628 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=2 | true
| true
   3 | poc-hubble04:35432 | poc-hubble04:35432 | v3.10.2-dirty | 2022-03-10 10:25:49.462678 | 2022-03-15 01:29:34.370426 | country=us,region=us-east1,datacenter=tianyun,rack=1,node=4  | true
| true
   4 | poc-hubble03:35432 | poc-hubble03:35432 | v3.10.2-dirty | 2022-03-12 02:35:47.990198 | 2022-03-15 01:29:33.655949 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=3 | true
| true
   5 | poc-hubble05:35432 | poc-hubble05:35432 | v3.10.2-dirty | 2022-03-10 10:26:14.969284 | 2022-03-15 01:29:37.194499 | country=us,region=us-east1,datacenter=tianyun,rack=1,node=5  | true
| true
(5 rows)
```

3.4.5.2 运行任务查看

```
show jobs;
```

```
root@poc-hubble01:35432/guohq> show jobs;
       job_id       |     job_type     |                                          description
                    | statement | user_name |  status   |              running_status              |          created           |          started           |          finished          |          modi
fied          | fraction_completed | error | coordinator_id
--------------------+------------------+-----------------------------------------------------------------------------------------------------------
--------------------+-----------+-----------+-----------+------------------------------------------+----------------------------+----------------------------+----------------------------+-------------
--------------+--------------------+-------+----------------
  744604774892044289 | SCHEMA CHANGE GC | GC for DROP TABLE xuejs.public.tmp_hubble_0623e31f6efd4d0b8088163d624a3d19
                    |           | hubble    | running   | waiting for GC TTL                       | 2022-03-15 00:55:43.900169 | 2022-03-15 00:55:43.920513 | NULL                       | 2022-03-15 00
:55:43.926989 |                  0 |       |           NULL
  744432793826230273 | SCHEMA CHANGE GC | GC for DROP TABLE xuejs.public.tmp_hubble_f1c992ebbcab42b0bb4dda024d0adcf8
                    |           | hubble    | running   | waiting for GC TTL                       | 2022-03-14 10:20:59.444043 | 2022-03-14 10:20:59.470416 | NULL                       | 2022-03-14 10
:20:59.47601  |                  0 |       |           NULL
  744431129029181441 | SCHEMA CHANGE GC | GC for DROP TABLE xuejs.public.tmp_hubble_ed36cdbbc9e840448cb4c06e4058d30a
                    |           | hubble    | running   | waiting for GC TTL                       | 2022-03-14 10:12:31.388304 | 2022-03-14 10:12:31.406302 | NULL                       | 2022-03-14 10
```

只有确认此任务影响到整个集群才后进行取消任务操作，job_id 为上述查询出来的。

```
cancel job 744604774892044289;
```

3.4.5.3 运行 sql 查看

当您看到一个查询需要很长时间才能完成时，您可以使用该`CANCEL QUERY`语句来结束它。

假设用于查找已运行超过 3s 的查询：

```
select query_id,start,client_address,query from [show all queries] t  WHERE
    start < (now() - INTERVAL '3 second');
```

```
            query_id            |          start           |    client_address    |
-----------------------------------+---------------------------+----------------------+-----------------------------------------
--
  16ffb7a67b8d18e70000000000000001 | 2022-07-08 01:51:53.560847 | 192.168.80.151:17681 | SELECT query_id, start, client_address, que
(1 row)
```

要取消这个长时间运行的查询，并阻止它消耗资源，请注意query_id并将其与CANCEL QUERY语句一起使用：

```
cancel query  '16ffb7a67b8d18e70000000000000001';
```

3.4.6 服务启停

> 注意使用 hubble 用户

数据库状态

```
systemctl status hubbletp
```

关闭数据库

```
sudo systemctl stop hubbletp
```

启动数据库

```
sudo systemctl start hubbletp
```

3.4.7 服务器端

主要检查 cpu、内存、网络、硬盘占用情况，其中 CPU 可能发生瞬时变大，这种情况需要根据预警的提示进行处理。

3.5 时区调整

> Hubble 时区设置为客户端 session 级别设置

3.5.1 SQL 中设置时区

- 设置语句

```
set timezone='Asia/Shanghai';
```

```
root@poc-hubble01:35432/defaultdb> set timezone='Asia/Shanghai';
SET

Time: 0ms total (execution 0ms / network 0ms)
```

- 展示时区

```
show time zone;
```

```
root@poc-hubble01:35432/defaultdb> show time zone;
    timezone
-----------------
```

```
  Asia/Shanghai
(1 row)
```

- 查看时间

```
select now();
```

```
root@poc-hubble01:35432/defaultdb> select now();
               now
---------------------------------
  2022-03-21 13:50:22.843663+08
(1 row)
```

3.5.2 JAVA 中设置时区

```
TimeZone.setDefault(TimeZone.getTimeZone("Asia/Shanghai"));
```

```
public static void main(String[] args) {
        // TODO Auto-generated method stub
        TimeZone.setDefault(TimeZone.getTimeZone("Asia/Shanghai"));
    try {

        Class.forName('org.postgresql.Driver');

        Connection con = DriverManager.getConnection(url, user, password);

    } catch (Exception e) {
        // TODO: handle exception
    }
    }
```

- 执行 JAVA 类

Plain Text -Duser.timezone=Asia/Shanghai

3.5.3 在 Dbeaver 设置时区

打开 dbeaver.ini，添加下列行

```
-Duser.timezone=Asia/Shanghai
```

```
-Dosgi.requiredJavaVersion=11
-Xms64m
-Xmx1024m
-Duser.timezone=Asia/Shanghai
-Ddbeaver.distribution.type=exe
```

3.6 租户管理

3.6.1 定义

多租户简单来说是指一个单独的实例可以为多个组织服务，多租户为共用的数据如何以单一系统架构与服务提供多数客户端相同甚至可定制化的服务，并且仍然可以保障客户的数据隔离。一个支持多租户技术的系统需要在设计上对它的数据和配置进行虚拟分区，从而使系统的每个租户或称组织都能够使用一个单独的系统实例，并且每个租户都可以根据自己的需求对租用的系统实例进行个性化配置。

3.6.2 租户的作用

租户的特点：既可以实现多个租户之间共享系统实例，同时又可以实现租户的系统实例的个性化定制；可以保证系统共性的部分被共享，个性的部分被单独隔离。通过在多个租户之间的资源复用，运营管理维护资源，有效节省开发应用的成本。并且数据库升级的时候，只需要少量修改相应的配即可。

3.6.3 Hubble 租户分类

Hubble 数据库也提供了租户的功能，主要分为按照节点和按照目录的两种方式来分租户。

3.6.4 复制区域

ALTER 语句用于重置或控制这些对象的复制区域。要查看有关现有复制区域的详细信息，您可以使用复制区域来控制特定数据的对应服务数量和位置，最终达到平衡集群的目的。

3.6.5 参数

- `range_name`: 要更改其复制区域配置的系统范围的名称。
- `database_name`: 要更改其复制区域配置的数据库的名称。
- `table_name`: 要更改其复制区域配置的表的名称。
- `partition_name`: 要更改其复制区域配置的分区的名称。
- `index_name`: 要更改其复制区域配置的索引的名称。

3.6.6 变量

- `range_min_bytes`: 区域中数据范围的最小大小（以字节为单位）。当一个范围小于这个大小时，hubble 会将它与相邻的范围合并。
- `range_max_bytes`: 区域中数据范围的最大大小（以字节为单位）。当一个范围达到这个大小时，hubble 会将它分成两个范围。
- `gc.ttlseconds`: 在垃圾收集之前将保留被覆盖值的秒数。如果值经常被覆盖，较小的值可以节省磁盘空间。
- `num_replicas`: 区域中的副本数。
- `constraints`: 影响副本位置的约束的数组。

3.6.7 示例

- `num_replicas`的最小值是 1。
- `constraints`约束的总数不能超过`num_replicas`
- 约束中出现的`key=value`或 attr 至少匹配一个 node/store。

3.6.7.1 按目录分租户

一个集群中有不同的目录，我们可以按照不同的属性进行数据的归类，能够让数据通过文件配置的方式进入特定的目录，具体配置方式在集群目录位置节点的启动属性。

- 节点启动属性的介绍

```
--attrs 举例: --attrs=ram:64gb、ssd、hhd
--store attrs 举例: --store=path=/data3/hubbledir/hubble,attrs=ssd --store=
    ↪ path=/data3/hubbledir/hubble,attrs=hdd:7200rpm
```

- 指定数据存储在 store 属性中

```
ALTER table kld.dept configure zone using constraints='[+ssd]',num_replicas = 3;
```

3.6.7.2 按节点分租户

3.6.7.2.1 索引层级

- 查看索引节点配置

```
root@hubble01:35432/kld> show zone configuration for index emp_index;
     target      |              raw_config_sql
----------------+-------------------------------------------
  RANGE default | ALTER RANGE default CONFIGURE ZONE USING
                |     range_min_bytes = 134217728,
                |     range_max_bytes = 536870912,
                |     gc.ttlseconds = 90000,
                |     num_replicas = 3,
                |     constraints = '[]',
                |     lease_preferences = '[]'
(1 row)

Time: 33ms total (execution 33ms / network 0ms)
```

- 更新索引的配置

```
root@hubble01:35432/kld> ALTER index emp_index configure zone using constraints
    ↪ ='{"+node=3":1,"+node=4":1,"+node=5":1}',num_replicas = 3;
CONFIGURE ZONE 1

Time: 198ms total (execution 197ms / network 0ms)
```

configure zone 为区域配置，constraints 为约束条件，num_replicas 为约束配置数量，以上述语句为例子，有 3,4,5 三个节点

- 再次查看索引节点配置

```
root@hubble01:35432/kld> show zone configuration for index emp_index;
              target              |                      raw_config_sql
----------------------------------+-----------------------------------------------------------
    ↪
```

```
  INDEX kld.public.emp@emp_index | ALTER INDEX kld.public.emp@emp_index
     ↪ CONFIGURE ZONE USING
                                 |     range_min_bytes = 134217728,
                                 |     range_max_bytes = 536870912,
                                 |     gc.ttlseconds = 90000,
                                 |     num_replicas = 3,
                                 |     constraints = '{+node=3: 1, +node=4: 1, +
                                    ↪ node=5: 1}',
                                 |     lease_preferences = '[]'
(1 row)

Time: 26ms total (execution 25ms / network 0ms)
```

此时，我们可以看到索引的复制区域在 3,4,5 三个节点。

3.6.7.2.2 表层级

- 查看表的区域节点

```
root@hubble01:35432/kld>  show zone configuration for table dept;
     target     |              raw_config_sql
----------------+-------------------------------------------
  RANGE default | ALTER RANGE default CONFIGURE ZONE USING
                |     range_min_bytes = 134217728,
                |     range_max_bytes = 536870912,
                |     gc.ttlseconds = 90000,
                |     num_replicas = 3,
                |     constraints = '[]',
                |     lease_preferences = '[]'
(1 row)

Time: 96ms total (execution 95ms / network 0ms)
```

- 更新表的配置

```
root@hubble01:35432/kld> ALTER table dept configure zone using constraints='{"+
   ↪ node=3":1,"+node=4":1,"+node=5":1}',num_replicas = 3;
CONFIGURE ZONE 1

Time: 44ms total (execution 44ms / network 0ms)
```

- 查看表的配置

```
root@hubble01:35432/kld> show zone configuration for table dept;
    target    |                     raw_config_sql
-------------+-------------------------------------------------------------
  TABLE dept | ALTER TABLE dept CONFIGURE ZONE USING
             |     range_min_bytes = 134217728,
             |     range_max_bytes = 536870912,
```

```
              |      gc.ttlseconds = 90000,
              |      num_replicas = 3,
              |      constraints = '{+node=3: 1, +node=4: 1, +node=5: 1}',
              |      lease_preferences = '[]'
(1 row)

Time: 12ms total (execution 12ms / network 0ms)
```

- 重置约束

```
root@hubble01:35432/kld> ALTER TABLE dept CONFIGURE ZONE USING DEFAULT;
CONFIGURE ZONE 1

Time: 138ms total (execution 138ms / network 0ms)
```

3.6.7.2.3 数据库层级

- 查看数据库配置节点

```
root@hubble01:35432/test> show zone configuration for database kld;
     target     |              raw_config_sql
----------------+-------------------------------------------
  RANGE default | ALTER RANGE default CONFIGURE ZONE USING
                |      range_min_bytes = 134217728,
                |      range_max_bytes = 536870912,
                |      gc.ttlseconds = 90000,
                |      num_replicas = 3,
                |      constraints = '[]',
                |      lease_preferences = '[]'
(1 row)

Time: 5ms total (execution 5ms / network 0ms)
```

- 更改数据库的配置

```
ALTER database kld configure zone using constraints='{"+node=3":1,"+node=4":1,"+
    ↪ node=5":1}',num_replicas = 3;
```

- 查看更改后的配置

```
root@hubble01:35432/defaultdb> show zone configuration for database kld;
     target    |                      raw_config_sql
---------------+-------------------------------------------------------------
  DATABASE kld | ALTER DATABASE kld CONFIGURE ZONE USING
               |      range_min_bytes = 134217728,
               |      range_max_bytes = 536870912,
               |      gc.ttlseconds = 90000,
               |      num_replicas = 3,
               |      constraints = '{+node=3: 1, +node=4: 1, +node=5: 1}',
```

```
                |     lease_preferences = '[]'
(1 row)

Time: 4ms total (execution 4ms / network 0ms)
```

3.6.7.2.4 RANGE 层级

- range 更改与查看

```
root@hubble01:35432/kld> SHOW ZONE CONFIGURATION FROM RANGE default;
     target     |              raw_config_sql
----------------+-------------------------------------------
  RANGE default | ALTER RANGE default CONFIGURE ZONE USING
                |     range_min_bytes = 134217728,
                |     range_max_bytes = 536870912,
                |     gc.ttlseconds = 90000,
                |     num_replicas = 3,
                |     constraints = '[]',
                |     lease_preferences = '[]'
(1 row)

Time: 2ms total (execution 2ms / network 0ms)
```

```
root@hubble01:35432/kld> ALTER RANGE default CONFIGURE ZONE USING num_replicas =
   ↪  5, gc.ttlseconds = 100000;
CONFIGURE ZONE 1

Time: 128ms total (execution 128ms / network 0ms)
```

```
root@hubble01:35432/kld> SHOW ZONE CONFIGURATION FROM RANGE default;
     target     |              raw_config_sql
----------------+-------------------------------------------
  RANGE default | ALTER RANGE default CONFIGURE ZONE USING
                |     range_min_bytes = 134217728,
                |     range_max_bytes = 536870912,
                |     gc.ttlseconds = 100000,
                |     num_replicas = 5,
                |     constraints = '[]',
                |     lease_preferences = '[]'
(1 row)

Time: 3ms total (execution 2ms / network 0ms)
```

> gc.ttlsecond 的值不建议低于 600，太低可能会导致查询太慢；也不建议太高，超过 90000 可能引起服务器性能下降。

- 说明

租户其实是各个服务中的一些可以访问的资源集合。这些资源集合可供多个用户使用，这也是为什么用户默认的总是绑定到某些租户上；用户通过租户访问计算管理资源，也就是说必须指定一个相应的租户才进行资源的使用。

4 数据迁移

4.1 迁移概述

hubble 支持从以下数据库导入数据：

- MySQL
- ORACLE(使用 CSV)
- PostgreSQL

并来自以下数据格式：

- CSV
- AVRO

本页列出了在计划迁移到 hubble 时需要注意的一般注意事项。

除了下面列出的信息之外，请参阅以下页面，了解适用于您要从中迁移的数据库或数据格式的具体说明和注意事项：

- 从 Oracle 迁移
- 从 PostgreSQL 迁移
- 从 MySQL 迁移
- 从 CSV 迁移
- 从 DB2 迁移

有关在 hubble 中优化导入性能的最佳实践，请参阅导入性能最佳实践。

4.1.1 导入文件的存储

在迁移期间，`IMPORT`与外部文件存储交互的所有功能都假定每个节点都具有该存储的完全相同的视图。换句话说，为了从文件导入，每个节点都需要对该文件具有相同的访问权限。

4.1.2 架构和应用程序更改

可能必须更改架构以及应用程序与数据库的交互方式。我们强烈建议您针对 hubble 测试您的应用程序，以确保满足下面的要求：

- 您的数据状态是您在迁移后所期望的。
- 性能符合您的应用程序工作负载的预期。您可能需要应用一些最佳实践来优化 hubble 中的 SQL 性能。

4.1.3 数据类型大小

超过一定大小，比如`STRING`, `DECIMAL`, `ARRAY`, `BYTES`等很多数据类型可能会因为写放大 JSONB 而遇到性能问题。请参阅每种数据类型的文档以了解其建议的大小限制。

4.2 CSV 迁移

4.2.1 步骤一 CSV 格式数据准备

您将需要为每个表导出一个 CSV 文件，并具有以下要求：

1. 文件必须为有效的 CSV 格式，但请注意，分隔符必须为单个字符。要使用逗号以外的其他字符（例如制表符），请使用`delimiter`设置自定义分隔符。
2. 文件必须为 UTF-8 编码。
3. 如果字段中出现以下字符之一，则该字段必须用双引号引起来：

- 分隔符（`,`）为默认的情况
- 双引号（`"`）
- 换行符（`\n`）
- 回车（`\r`）

1. 如果使用双引号将字段括起来，则必须在字段内部出现双引号，然后在其前面加上另一个双引号，以对其进行转义。例如：”aaa”,”b””bb”,”ccc”。
2. 如果列是类型`BYTES`，则它可以是有效的 UTF-8 字符串，也可以是以开头的十六进制编码的字节常量`\x`。例如，一个字段，其值应是字节 1，2 将被写为\x0102。

4.2.2 步骤二：将数据文件放置在集群可访问到的位置

URL 必须使用以下格式:

```
[scheme]://[host]/[path]?[parameters]
```

类型	schema	host	参数	示例
NFS/Local	nodelocal	节点 ID 或为空	`N/A`	`nodelocal://n/path/mydatest,nodelocal://n/path/mydatest2`

其中 n 代表在哪个 node 节点，见以下示例（以下举例在第一个节点进行的操作）。

4.2.3 步骤三：将数据迁移到 Hubble 数据库中

在导入数据前一定要先创建表

4.2.3.1 hubble 库中表的数据迁移

```
import into employees (emp_no, birth_date,first_name,last_name,gender,hire_date)
    ↪ CSV DATA ('nodelocal://1/customers/export176656930ae17564000000000000001
    ↪ -n871825080457560067.0.csv')
```

4.2.3.2 多文件导入

```
import into employees csv data(
'nodelocal://1/customers/export176656930ae17564000000000000001-
    ↪ n871825080457560067.0.csv',
'nodelocal://1/customers/export176656930ae17564000000000000001-
    ↪ n871825087878788759.0.csv'
) delimiter = e'|';
```

4.2.4 配置参数

以下选项可用于IMPORT ... CSV。

4.2.4.1 列定界符

该delimiter选项用于设置Unicode字符，该字符标记每列的结尾。默认值：，。

用法示例：

```
import into employees
  CSV DATA ('nodelocal://1/customers/*.csv') WITH delimiter = e'\t';
```

```
        job_id        |  status   | fraction_completed |  rows  | index_entries |
          ↪  system_records |  bytes
+--------------------+-----------+--------------------+--------+---------------+----------
  ↪
  535428382456872961 | succeeded |                  1 | 300024 |             0 |
    ↪                0 | 12134341
(1 row)

Time: 5.768633441s
```

4.2.4.2 跳过注释行

comment选项确定哪个 Unicode 字符标记数据中要跳过的行。

用法示例：

```
import into employees
  CSV DATA ('nodelocal://1/customers/*.csv') WITH  comment = '#';
```

4.2.4.3 跳过标题行

skip选项确定导入文件时要跳过的标题行数。

用法示例：

```
import into employees
  CSV DATA   ('nodelocal://1/customers/*.csv') WITH skip = '2';
```

4.2.4.4 空字符串

nullif选项定义应将哪个字符串转换为NULL。

用法示例：

```
import into employees
  CSV DATA ('nodelocal://1/customers/*.csv') WITH nullif = '';
```

HUBBLE

4.2.4.5 文件压缩格式

decompress选项用于指定需导入的csv文件压缩格式，默认不使用压缩格式

可选的压缩格式包括：gzip，bzip，none

```
import into employees
  CSV DATA   ('nodelocal://1/customers/*.csv.gz') WITH decompress = 'gzip';
```

4.2.4.6 行限制数

row_limit选项确定要从表中导入的行数。它有助于在执行更耗时和更耗费资源的导入之前快速查找错误。

用法示例：

```
import into employees
  CSV DATA   ('nodelocal://1/customers/*.csv')
  WITH row_limit = '10';
```

4.3 Mysql 迁移

本文档适用于单表形式的数据迁移。

表导出导入建议

表的操作顺序：

- 首先对字典表进行操作，比如：国家表、职业表等基本不更新或者很少进行更新的表。
- 对有主外键依赖的表进行优先操作。
- 对视图、序列等进行操作。

在进行 MySQL 向 Hubble 的数据迁移时，需要关注以下几点：

- MySQL 和 Hubble 数据类型不完全一致，需要进行数据类型转换。
- MySQL 和 Hubble 的语法也有所不同，需要进行语法转换。
- 在进行数据导入时，需要注意数据的完整性和一致性。
- 在导入数据之前，最好先备份原有数据，以防数据丢失。

4.3.1 MySQL 迁移至 Hubble

4.3.1.1 步骤一：导出需要迁移的表的数据

推荐用 dbeaver 工具

- 1. 选中表 (右键)
- 2. 点击'导出数据'
- 3. 选中 SQL(导出 sql 到insert语句)
- 4. 一直'下一步'即可

例如：导出的文件命名为 emp.sql

4.3.1.2 步骤二：导出需要迁移的表的建表语句

- 1. 选中表 (右键)
- 2. 点击'生成 SQL'

- 3. 选 DDL，复制出建表语句，对建表语句进行修改

进行表结构修改: 主要进行数据类型映射

数据类型映射

- 使用下表进行数据类型映射:

MySQL 数据类型	Hubble 数据类型
CHAR(n)	CHAR(n)
CLOB	STRING
DATE	DATE
FLOAT(n)	DECIMAL(n)
DECIMAL(m,n)	DECIMAL(m,n)
DOUBLE	DECIMAL(m,n)
JSON	JSON
BLOB	BYTES
VARCHAR(n)	VARCHAR(n)
int(n)	int
bigint64	int64
bigint	INT8
TINYINT	INT
SMALLINT	INT
MEDIUMINT	INT
TIMESTAMP(p)	TIMESTAMP
TIME	TIME
date	date
datetime	datetime
BOOLEAN	BOOL
ENUM	ENUM

4.3.1.3 步骤三：在 Hubble 数据库中建表

默认情况下 Hubble 创建的数据库名与 mysql 当中一致，本示例 mysql 中的库名为 guar

```
create database if not exists guar;

use guar;

create table guar.emp (
  id int NOT NULL ,
  name varchar(255) ,
  sex varchar(255) ,
  age int ,
  zzms varchar(255) ,
  eid int ,
  PRIMARY KEY (id)
)
```

HUBBLE

4.3.1.4 步骤四：把数据导入 Hubble 中

将文件 (emp.sql) 放置到可访问的位置 (与 certs 同级别目录)，如：`/var/lib/hubble`下

```
hubble sql   --certs-dir=/var/lib/hubble/certs --host=hubble01:15432 < emp.sql
```

4.3.2 部分常见函数的转换

mysql	hubble
ifnull()	coalesce()
str1+str2	str1
`date_format`	`cast()`字符串转日期
`date_format`	`experimental_strftime()`日期转字符串
`FROM_UNIXTIME`	1668338530::timestamp(时间戳转日期)
`UNIX_TIMESTAMP`	extract(epoch from cast('2022-11-13 11:22:10' as timestamp))::int(日期转时间戳)
substring()	substring()
replace()	replace()
trim()	trim()
ltrim()	ltrim()
rtrim()	rtrim()
lower()	lower()
upper()	upper()
now()	now()
curdate()	current_date()
date_add()	now() + interval '1 day'
datediff()	age()
sysdate()	now()

4.3.3 Mysql 中有自增字段的表的导入

用 Hubble 的序列替换 Mysql 的自增字段

Mysql 中形式：

```
CREATE TABLE test_col (
id INT(8) NOT null auto_increment primary key,
name VARCHAR(100) NOT null,
age  int(1)
);
```

Hubble 中形式：

```
create sequence userid_seq;

create table test_col(
  id int PRIMARY KEY DEFAULT nextval('userid_seq'),
  name string,
```

```
  age  int
);
```

4.3.4 MySQL 问题总结

- 1.bit 类型迁移问题

bit 类型或者在 mysql 的迁移脚本中处理为`bool`，或者处理为`int64`。这里有个注意的点，即使处理为`bool`，`insert`语句中也不可以处理为`true`或者`false`。需要处理为 0 和 1，0 来代表`false`，1 来代表`true`。另外 sql 中注意对 bit 的语句的`where`条件`='f'`或者`='t'`，需要改为`='0'`和`='1'`。

- 2.`find_in_set()`函数问题

`find_in_set()`(查询字段 (strlist) 中是否包含 (str) 的结果) 在 Hubble 中需要用`string_to_array()`和`ANY()`这两个函数来代替，也可以用`string_to_array()`和和`array_position()`来代替

- 3. 类型转换问题

MySQL 中支持隐式类型转换，但是 Hubble 需要强转语法是: 字段:: 数据类型，例如：`id::varchar`，项目中常遇见的时间戳转日期

```
select 1668338530::timestamp;
```

- 4. 分页问题

MySQL 中分页配置是`pagehelper.helperDialect=mysql`，在 Hubble 中用的话需要先把这个配置注释掉，在加上`pagehelper.auto-dialect=true`和`pagehelper.auto-runtime-dialect=true`这两个配置

- 5. 如果遇到关键字当字段名，MySQL 是加`''`单引号转，Hubble 是加`""`双引号转。

`order`字段为关键字的 mysql 用单引号形式：

```
CREATE TABLE test_col (
id INT(8) NOT NULL,
name VARCHAR(100) NOT NULL,
`order` INT(8) NOT NULL,
PRIMARY KEY (`id`)
);
```

`order`字段为关键字的 Hubble 用双引号形式：

```
CREATE TABLE test_col (
id INT8 NOT NULL,
name VARCHAR(100) NOT NULL,
"order" INT8 NOT NULL,
PRIMARY KEY (id)
);
```

- 6.MySQL 的`database()`函数要用 Hubble 中`current_database()`函数来代替。

mysql:`select database()`等同于 Hubble:`select current_database()`

- 7.`split failed while applying backpressure to Put`错误

split failed while applying backpressure to Put错误引起的原因是 Hubble 使用了 MVCC，而如果单表中数据频繁更新，导致键值过大就会进行拆分，单个键值拆分时会导致这个错误发生。常用的解决方法可以是修改表的 gc ttl(默认是 24 小时)，alter table test_a CONFIGURE ZONE using gc.ttlseconds = 1800; 将表的 ttl 时间改为 1800 秒，这存在的问题是会导致数据库回滚时，恢复时只能恢复 1800 秒之内的。另外一种方式是在配置中加大分片的大小。还有一种是将频繁更新的字段作为单独的表列族来使用，这样会减少更新时 mvcc 值的大小。

4.4 Oracle 迁移

4.4.1 步骤 1: 导出 dmp 文件

- 假设一个用户为 hubble，密码也为 hubble，并且用户拥有读写 (read/write) 权限。
- 导出 dmp 文件 (主要是表结构和视图)

```
expdp hubble/hubble directory=directory_name dumpfile=oracle_example.dmp content
    ↪ =metadata_only logfile=example.log
```

4.4.2 步骤 2: 将 dmp 文件转换为 sql 格式

```
impdp hubble/hubble directory=directory_name dumpfile=oracle_example.dmp sqlfile
    ↪ =example_sql.sql TRANSFORM=SEGMENT_ATTRIBUTES:N:table PARTITION_OPTIONS=
    ↪ MERGE
```

导出后的 sql 文件，包括了用户所拥有的的权限，建表语句、主键、索引、表之间的主外键关系，视图等一些信息。但是对应到 hubble 上的话仍然需要修改，注意事项包括：

- 1. 表的主键和表之间的主外键关系在create table语句中创建。
- 2. 视图创建去掉FORCE关键字。
- 3.oracle 数据类型转换成 hubble 的数据类型。
- 4. 注释语句，索引语句可以继续使用。

4.4.3 步骤 3: 导出表数据

- 需要将每个表的数据提取到数据列表文件 (.lst) 中，编写 SQL 脚本 (spool.sql) 来执行此操作：

```
cat spool.sql
```

```
SET ECHO OFF
SET TERMOUT OFF
SET FEEDBACK OFF
SET PAGESIZE 0
SET TRIMSPOOL ON
SET WRAP OFF
set linesize 30000
SET RECSEP OFF
SET VERIFY OFF
SET ARRAYSIZE 10000
SET COLSEP '|'
```

```
SPOOL '&1'
ALTER SESSION SET nls_date_format = 'YYYY-MM-DD HH24:MI:SS';
SELECT * from &1;
SPOOL OFF
SET PAGESIZE 24
SET FEEDBACK ON
SET TERMOUT ON
```

- 用两张表举例 cust_info 和 trans

```
create table cust_info(cust_no varchar2(30) primary key,cust_name varchar2(30) ,
    ↪ cust_card_no  varchar(18),age  number);
create table trans (cust_id varchar2(20) primary key,cust_no varchar2(30),
    ↪ trans_date date,trans_aml number(10,2));
```

- 要提取数据，登录 sqlplus

```
cd /home/oracle/dump
```

```
sqlplus hubble/hubble
```

- 传入参数为表名，且表中必须包含数据，不能为空

```
SQL> @spool cust_info
SQL> @spool trans
```

退出 SQL * Plus:

```
EXIT
```

- 查看是否存在.lst 文件

```
[oracle@zhangdb dump]$ ls *.lst
cust_onfo.lst  trans.lst
```

4.4.4 步骤 4: 配置表数据并将其转换为 CSV

每个表的数据列表文件都需要转换为 CSV 并针对 hubble 进行格式化。我们编写了一个简单的 Python 脚本 (fix-example.py) 来执行此操作:

```
cat fix-example.py
```

```
import csv
import string
import sys

for lstfile in sys.argv[1:]:
  filename = lstfile.split(".")[0]

  with open(sys.argv[1]) as f:
    reader = csv.reader(f, delimiter="|")
```

HUBBLE

```
    with open(filename+".csv", "w") as fo:
      writer = csv.writer(fo)
      for rec in reader:
        writer.writerow(map(string.strip, rec))
```

- 执行 python 脚本，转换 csv 文件

```
python fix-example.py cust_info.lst trans.lst
```

- 查看是否存在.csv 文件

```
[oracle@zhangdb dump]$ ls *.csv
cust_info.csv  trans.csv
```

4.4.5 步骤 5:Oracle 映射到 Hubble 数据类型

- 使用之前**步骤**二生成的 SQL 文件，编写`IMPORT TABLE`与要导入的表数据的模式匹配的语句。
- 删除所有特定于 Oracle 的属性，重新映射所有 Oracle 数据类型，重构所有`CREATE TABLE`语句。

数据类型映射

- 使用下表进行数据类型映射:

Oracle 数据类型	Hubble 数据类型
BLOB	BYTES
CHAR(n)，CHARACTER(n)n <256	CHAR(n)，CHARACTER(n)
CLOB	STRING
DATE	DATE
FLOAT(n)	DECIMAL(n)
INTERVAL YEAR(p) TO MONTH	VARCHAR，INTERVAL
INTERVAL DAY(p) TO SECOND(s)	VARCHAR，INTERVAL
DOUBLE	DECIMAL(m,n)
JSON	JSON
LONG	STRING
LONG RAW	BYTES
NCHAR(n)n <256	CHAR(n)
NCHAR(n)n>255	VARCHAR，STRING
NCLOB	STRING
NUMBER(p,0)，NUMBER(p)1 <= p <5	INT2
NUMBER(p,0)，NUMBER(p)5 <= p <9	INT4
NUMBER(p,0)，NUMBER(p)9 <= p <19	INT8
NUMBER(p,0)，NUMBER(p)19 <= p <= 38	DECIMAL(p)
NUMBER(p,s) s> 0	DECIMAL(p,s)
NUMBER	DECIMAL
NVARCHAR2(n)	VARCHAR(n)
RAW(n)	BYTES
TIMESTAMP(p)	TIMESTAMP
TIMESTAMP(p) WITH TIME ZONE	TIMESTAMP WITH TIMEZONE
VARCHAR(n)，VARCHAR2(n)	VARCHAR(n)

HUBBLE

Oracle 数据类型	Hubble 数据类型
XML	JSON

- BLOBS，CLOBS应将其转换为BYTES，或者STRING大小可变的位置，但建议将值保持在 1MB 以下，以确保性能。1MB 以上的任何内容都将需要重构到对象存储中，并在表中嵌入一个指针来代替对象。
- JSON，XML类型可以转换为JSONB使用任何XML到JSON的转换。在导入 hubble 之前XML必须将其转换为JSONB ↪ 。
- 转换时NUMBER(p,0)，请考虑NUMBER将 Base-10 限制的INT类型映射到 hubble 类型的 Base-10 限制，NUMBERS可以转换为DECIMAL。

4.4.6 步骤 6: 数据导入

- 以上述 oracle 两张表 cust_info 和 trans 为例在 hubble 数据库中建表，对应字段类型参考步骤五的数据类型映射表

```
create table  cust_info(cust_no varchar(40) primary key,cust_name varchar(40) ,
    ↪ cust_card_no  varchar(20),age  DECIMAL);
create table trans (cust_id varchar(20) primary key,cust_no varchar(30),
    ↪ trans_date timestamp,trans_aml DECIMAL(10,2));
```

- 将 csv 文件放置集群在可访问的位置

```
[hubble@poc-hubble01 ~]$ cd /data_shares/cus
[hubble@poc-hubble01 cus]$ ls
cust_info.csv  trans.csv
```

- 导入 cust_info

```
IMPORT into cust_info ( cust_no ,cust_name, cust_card_no, age
) CSV DATA ( 'nodelocal://1/cus/cust_info.csv' )
WITH
    DELIMITER = ','
;
```

```
        job_id        |  status   | fraction_completed | rows | index_entries |
            ↪ bytes
---------------------+-----------+--------------------+------+---------------+---------
    ↪
  748594745725714433 | succeeded |                  1 |    2 |             0 |
      ↪    114
```

```
root@poc-hubble01:35432/oracle> select * from cust_info;
    cust_no    | cust_name |     cust_card_no     | age
--------------+-----------+---------------------+------
  a2327818434 | 张三      | 130224195405256540 |  19
  a8723518461 | 马超      | 110224199905216541 |  15
(2 rows)
```

- 导入 trans

```
IMPORT into trans ( cust_id ,cust_no,trans_date, trans_aml
) CSV DATA ( 'nodelocal://1/cus/trans.csv' )
WITH
    DELIMITER = ','
;
```

```
        job_id        |  status   | fraction_completed | rows | index_entries |
          ↪ bytes
---------------------+-----------+--------------------+------+---------------+--------
  ↪
  748595201171456001 | succeeded |                  1 |    2 |             0 |
    ↪   102
```

```
root@poc-hubble01:35432/oracle> select * from trans;
    cust_id     |    cust_no    |      trans_date       | trans_aml
---------------+-------------+---------------------+-----------
  saae22324434 | a2343355545 | 2013-02-26 11:07:25 |    123.40
  saae22324489 | a2343355545 | 2013-02-26 12:07:00 |    523.40
(2 rows)
```

步骤六可参考 CSV 数据迁移

4.5 DB2 迁移

4.5.1 步骤一: 导出建表语句

- 假设数据库为 HGQW1108，用户名 db2inst1，密码 HUBBLE
- DB2 迁移到 hubble，整库从 DB2 导出的话也要单表导入 Hubble，所以只考虑单表导出，然后导入到 Hubble
- 进入/home/db2inst1/data 目录 (**根据具体项目确定目录**)，执行以下语句，导出建表 sql 脚本

```
db2start #开启数据库
```

```
db2look -d HGQW1108 -a -e -i db2inst1 -w HUBBLE -o db2.sql
```

- 查看目录

```
[db2inst1@hilbert02 data]$ ls
db2.sql
```

4.5.2 步骤二: 导出 csv 文件

- 连接数据库 (/home/db2inst1/data 目录下)

```
db2
```

```
connect to HGQW1108
```

- 导出单表

```
export to /home/db2inst1/data/tb_BBS.csv of del select * from  tb_BBS
```

```
db2 => export to /home/db2inst1/data/tb_BBS.csv of del select * from  tb_BBS
SQL3104N  The Export utility is beginning to export data to file
"/home/db2inst1/data/tb_BBS.csv".

SQL3105N  The Export utility has finished exporting "3" rows.


Number of rows exported: 3
```

- 退出数据库连接，并查看 csv 生成情况

```
quit
```

```
[db2inst1@hilbert02 data]$ ls *.csv
tb_BBS.csv
```

4.5.3 步骤三:DB2 映射 Hubble 数据类型

- 使用之前**步骤**一生成的 SQL 文件，编写`IMPORT TABLE`与要导入的表数据的模式匹配的语句。
- 删除所有特定于 DB2 的属性，重新映射所有据类型，重构所有`CREATE TABLE`语句

数据类型映射表

DB2 数据类型	Hubble 数据类型
BLOB	BYTES 1 个
CHAR(n)	CHAR(n)
CLOB	STRING 1 个
DATE	DATE
FLOAT(n)	DECIMAL(n)
VARCHAR2(n)	VARCHAR(n)
TIMESTAMP	TIMESTAMP
Integer	TNT4
FLOAT(n)	DECIMAL(n)
Double	DECIMAL
Smallint	TNT4
Bigint	TNT8
Numeric(p,s)	DECIMAL(p,s)
Decimal(p,s)	DECIMAL(p,s)
Vargraphic(n)	varchar(n)
Graphicn)	varchar(n)
Real	DECIMAL(n)

- 确定好数据类型后修改对应的建表语句
- 调整主键的建表格式，以上述 tb_bs 表举例

HUBBLE

- db2 语法

```
create table TB_BS(id int not null,title varchar(10),content varchar(40),primary
    ↪ key(id))
```

- hubble 语法

```
create table TB_BS(id int not null primary key,title varchar(10),content varchar
    ↪ (40));
```

4.5.4 步骤四: 将导出的数据文件放置于集群可访问到的位置

- Hubble 集群中的每个节点都需要访问到需导入的数据文件。
- URL 必须使用以下格式:

```
[scheme]://[host]/[path]?[parameters]
```

当前支持的类型如下:

类型	schema	host	参数	示例
NFS/Local	nodelocal	节点 ID 或为空	N/A	nodelocal:///path/mydatest,nodelocal://n/path/mydatest

4.5.5 步骤五: 导入数据

- 在完成建表以及文件放置后，执行导入语句

```
IMPORT into TB_BS( id ,title,content
) CSV DATA ( 'nodelocal://1/cus/tb_BBS.csv' )
WITH
    DELIMITER = ','
;
```

```
        job_id        |  status   | fraction_completed | rows | index_entries |
            ↪ bytes
----------------------+-----------+--------------------+------+---------------+---------
    ↪
  749715636016250881 | succeeded |                  1 |    3 |             0 |
      ↪     93
(1 row)

Time: 365ms total (execution 365ms / network 0ms)
```

- 查看数据

```
select * from tb_bs;
  id  | title | content
------+-------+-----------
  111 | TXT   | 文章描述
  123 | SQL   | 文本内容
```

```
  222 | CSV    | 文本导入
(3 rows)

Time: 2ms total (execution 2ms / network 0ms)
```

可参考 CSV 数据迁移

4.6 导入性能最佳实践

本页提供了在 hubble 中优化导入性能的最佳实践。

导入速度主要取决于您要导入的数据量。但是，有两个主要因素会对运行导入所需的时间产生很大影响：

- 拆分数据
- 导入格式

> 如果导入文件大小很小，那么您不需要做任何事情来优化性能。在这种情况下，无论设置如何，导入都应该快速运行。

4.6.1 将数据拆分为多个文件

将导入数据拆分为多个文件会对导入性能产生很大影响。以下格式支持使用`IMPORT INTO`多文件导入：

- `CSV`
- `DELIMITED DATA`
- `AVRO`

对于这些格式，建议将数据拆分为与节点数量一样多的文件。

例如，如果您有一个 3 节点集群，请将数据拆分为至少 3 个文件，创建表并导入该表：

```
create table emp (id int PRIMARY KEY, name TEXT, INDEX name_idx(name));
```

```
import into emp (id, name)
    CSV DATA (
      'nodelocal://1/emp.csv',
      'nodelocal://1/emp1.csv',
      'nodelocal://1/emp2.csv'
    );
```

hubble 会导入您提供给它的文件，并且不会进一步拆分它们。例如，如果您为所有数据导入一个大文件，hubble 将在一个节点上处理该文件，即使您有更多可用节点。但是，如果您导入两个文件（并且您的集群至少有两个节点），每个节点将并行处理一个文件。这就是为什么将数据拆分为与节点一样多的文件将大大减少导入数据所需的时间。

> 如果将数据拆分为比节点更多的文件，则不会对性能产生很大影响。

4.6.2 选择一种高性能的导入格式

由于处理方式不同，导入格式的性能不同。下面，导入格式从最快到最慢列出：

- CSV或DELIMITED DATA（两者的导入性能大致相同）
- AVRO
- MYSQLDUMP
- PGDUMP

我们建议将您的导入文件格式化为CSV。这些格式可以由多个线程并行处理，从而提高性能。要以这些格式导入，请使用IMPORT INTO。

MYSQLDUMP和PGDUMP运行单个线程来解析它们的数据，因此性能会大大降低。

4.6.2.1 将模式与数据分开导入

对于单表MYSQLDUMP或PGDUMP导入，将转储数据拆分为两个文件：

- 包含表模式的 SQL 文件
- 包含表数据的 CSV 文件

然后，导入仅模式文件：

```
import table emp
from pgdump
    'nodelocal://1/emp.sql' WITH ignore_unsupported_statements
;
```

并使用该IMPORT INTO语句将 CSV 数据导入新创建的表中：

```
import into emp (id, name)
CSV DATA
(
    'nodelocal://1/emp.csv'
)
;
```

这种方法的另一个好处是可以更快地警告导入的潜在问题；也就是说，您不必等待文件加载模式和数据就可以在模式中找到错误。

5 开发文档

5.1 数据库设计

5.1.1 概述

本页概述了 Hubble 中的数据库模式。

5.1.1.1 数据库架构对象

以下部分简要概述了构成 Hubble 中数据库模式的逻辑对象。

5.1.1.1.1 数据库

数据库对象构成了 Hubble 命名层次结构的第一层。

所有 Hubble 集群都包含一个名为defaultdb的默认数据库，但建议创建自己的数据库。

有关创建数据库的指导，请参阅创建数据库。

5.1.1.1.2 模式

模式构成命名层次结构的第二层。每个模式都属于一个数据库。模式包含表和其他对象，如视图和序列。

有关创建用户定义模式的指南，请参阅创建用户定义模式。

5.1.1.1.3 表

表属于命名层次结构的第三层和最低层。每个表都可以属于一个模式。

表包含数据行。一行数据中的每个值都属于一个特定的列。每列允许单一数据类型的数据值。可以使用列级约束进一步限定列，或使用表达式进行计算。

有关定义表格的指南，请参阅表的创建。

5.1.1.1.4 索引

索引是单个表中行的副本，按列或列集排序。给定特定列的值，Hubble 查询使用索引更有效地在表中查找数据。每个索引都属于一个表。

有关定义二级索引的指南，请参阅二级索引。

专门索引

Hubble 支持一些特殊类型的索引，旨在提高特定用例中的查询性能。包括：

- 索引行的子集
- 索引JSON和数组数据
- 索引表达式

5.1.1.1.5 其他对象

Hubble 在命名层次结构的第三层支持其他几个对象，包括可重用视图、序列和临时对象。

视图

视图是存储的和命名的选择查询。

序列

序列创建并存储顺序数据。

临时对象

临时对象是未存储到持久内存中的对象，例如表、视图或序列。

5.1.1.2 控制对对象的访问

Hubble 支持基于用户和基于角色的访问控制。通过角色或直接分配，可以授予用户查看、修改和删除数据库架构对象所需的权限。

默认情况下，创建对象的用户是该对象的所有者。对象所有者拥有查看、修改或删除该对象及其中存储的数据所需的所有权限。

有关所有权、特权和授权的更多信息，请参阅授权。

5.1.1.3 对象大小和缩放

Hubble 不会对大多数数据库对象施加硬性限制。

HUBBLE

5.1.1.3.1 硬限制

下表列出了 Hubble 施加的特定限制。

目的	范围	说明
角色名称	63 字节	
用户名	63 字节	这些等同于角色名称。
标识符长度	128 字节	此限制并未强制执行，但建议保持在这个限制内

5.1.1.3.2 行数

Hubble 可以通过添加额外的节点和存储来支持任意数量的行。

5.1.1.3.3 表和其他模式对象的数量

当扩展到大量表时，要了解：

- 增加 RAM 可能会对集群可以支持的这些对象的数量产生最大的影响，而增加节点的数量不会产生实质性影响。
- 集群上的数据库或模式数量对其可支持的表总数的影响最小。

5.1.2 创建数据库

此页面提供有关创建数据库的实践指导。

5.1.2.1 准备工作

在阅读本页之前，请执行以下操作：

- 创建 Hubble 集群或启动本地集群。
- 确认数据库架构对象。

5.1.2.2 创建数据库

数据库对象构成了 Hubble 命名层次结构的第一层。

要创建数据库，请使用一条`CREATE DATABASE`语句，遵循数据库最佳实践，并参阅以下示例。

5.1.2.2.1 数据库最佳实践

以下是创建和使用数据库时要遵循的一些最佳实践：

- 不使用预加载的`defaultdb`数据库。相反，使用语句创建用户自己的数据库。
- 以角色成员的身份（作为用户）创建数据库。
- 限制创建的数据库的数量。如果需要在集群中创建多个具有相同名称的表，请在同一数据库中的不同用户定义模式中执行此操作。

5.1.2.2.2 示例

`CREATE DATABASE`语句创建 test 库：

```
create database if not exists test;
```

查看集群中的数据库，`SHOW DATABASES`语句：

```
show databases;
```

```
  database_name | owner | primary_region | regions | survival_goal
----------------+-------+----------------+---------+----------------
  defaultdb     | root  | NULL           | {}      | NULL
  test          | root  | NULL           | {}      | NULL
  postgres      | root  | NULL           | {}      | NULL
  system        | node  | NULL           | {}      | NULL
```

5.1.3 创建表

本文档提供有关创建表的最佳指导，以及一些简单示例。

5.1.3.1 开始前的准备

在阅读本页之前，请执行以下操作：

- 创建 Hubble 集群
- 查看数据库架构对象
- 创建数据库
- 创建用户定义的架构

5.1.3.2 创建表

表是集群中的逻辑对象，用于存储从应用程序写入的数据，表中以行和列的形式进行数据记录。

要创建表，使用`CREATE TABLE`语句，并且遵循以下步骤：

- 命名一个表
- 定义列
- 选择主键列
- 添加约束
- 执行`CREATE TABLE`创建语句

请参阅以下提供的示例。

5.1.3.2.1 命名表

命名表是创建表的第一步。

`CREATE TABLE`语句通常采用以下形式：

```
CREATE TABLE {schema_name}.{table_name} (
  {elements}
  );
```

范围	描述
`{schema_name}`	用户定义模式的名称。
`{table_name}`	表的名称。
`{elements}`	以逗号分隔的表元素列表，例如列定义。

表命名最佳建议

以下是命名表时要遵循的一些最佳实践：

- 使用完全限定名称 (`CREATE TABLE database_name.schema_name.table_name`)。如果不指定数据库名称，Hubble 将使用 SQL 会话的当前数据库 (`defaultdb`默认情况下)。如果没有在表名中指定用户定义的模式，Hubble 将在预加载的`public`模式中创建表。
- 使用反映表内容的表名。例如，对于客户信息的表，可以使用名称`cust_info`。

表命名示例

在数据库`testdb`中，为表`cust_info`添加一个`CREATE TABLE`语句：

```
CREATE TABLE testdb.public.cust_info (
);
```

5.1.3.2.2 定义列

列定义通过将每行中的值分隔为单一数据类型的列来为表提供结构。

表中列的定义通常采用以下形式：

```
{column_name} {data_type} {column_qualification}
```

范围	描述
`column_name`	列的名称。
`data_type`	列的数据类型。
`column_qualification`	某些列限定条件，例如列级约束或计算列子句。

列定义最佳建议

以下是定义表列时要遵循的一些要求：

- 根据支持的列数据类型，为计划存储在列中的数据选择适当的类型。
- 使用具有固定大小限制的列数据类型，或为可变大小的列数据类型设置最大大小限制 (例如，VARCHAR(n))。
- 根据业务的实际情况，决定是否需要定义主键列。
- 根据业务的实际情况，并决定是否需要向列添加约束。

列定义

在`cust_info`表中加入列

```
CREATE TABLE testdb.public.cust_info (
 cust_name      STRING,
 cust_card_no   STRING,
 cust_phoneno   decimal(15)
);
```

上面显示的`cust_name`和`cust_card_no`列都使用`STRING`数据类型，这意味着任何列中的任何值都必须是数据类型`STRING`。而`cust_phoneno`列中必须为数字。

当然，Hubble 支持许多其他列数据类型，包括DECIMAL、INT、TIMESTAMP、UUID和枚举数据类型以及空间数据类型。

要创建用户定义的类型，使用CREATE TYPE语句

```
create type vtype as ENUM ('football', 'basketball', 'Rugby football');
```

然后您可以vtype用作type列的数据类型：

```
CREATE TABLE game (
  id UUID,
  type vtype,
  begin_time TIMESTAMPTZ
  );
```

type列中只允许vtype中定义的值

5.1.3.2.3 选择主键列

主键是一列或一组列，其值唯一标识数据行。每个表都需要一个主键。

主键在带有列约束CREATE TABLE的语句中定义。该约束要求所有受约束的列仅包含唯一值。

主键最佳实践

以下是选择主键列时要遵循的推荐的做法：

- 避免在单列顺序数据上定义主键。

使用单个顺序列（例如，自动递增列）上的主键查询表可能会导致对性能产生负面影响。

如果正在使用必须在顺序键上建立索引的表，请使用散列分片索引。

- 为每个表定义一个主键。

如果你创建一个没有定义主键的表，Hubble 会自动创建rowid。默认情况下，rowid为列中的每一行生成顺序的唯一标识符。值的顺序性质使rowid可能导致数据在集群中分布不均，这会对性能产生负面影响。此外，因为不能有意义地使用rowid列来过滤表数据，所以主键索引rowid不提供任何性能优化。这将要求用户始终通过为表根据业务自定义主键来提高性能。

- 如果可能，在多个列上定义主键约束

在定义复合主键时，请确保主键前缀第一列中的数据在集群中的节点之间均匀分布。

- 对于单列主键，使用UUID随机生成的默认值的类型化列

随机生成UUID值可确保主键值是唯一的并且在集群中分布良好。

主键示例

表和表中的CREATE TABLE语句需要显式定义主键。

```
create table cust_info (
    first_name string,
    last_name  string,
    phone_no   int,
    CONSTRAINT "primary" primary key (first_name, last_name)
  );
```

此主键将唯一标识用户数据行。

主键列也可以是单列，它们的值也应该在集群中均匀分布。

```
create table gameball (
      id uuid DEFAULT gen_random_uuid() primary key,
      begin_time TIMESTAMPTZ,
      is_null BOOL,
      end_time TIMESTAMPTZ
  );
```

除了PRIMARY KEY约束之外，该 id 列还有一个DEFAULT约束。此约束将列的默认值设置为gen_random_uuid()函数生成的值。此函数生成的值保证是唯一的，并且在集群中分布良好。

5.1.3.2.4 添加约束

除了PRIMARY KEY约束之外，Hubble 还支持许多其他列级约束，包括CHECK、DEFAULT、FOREIGN KEY、UNIQUE ↪ 和NOT NULL。使用约束可以简化表查询，提高查询性能，并确保数据在语义上保持有效。

要约束单个列，请将约束关键字添加到列的定义中。要约束多列，在CREATE TABLE语句中的列列表之后添加整个约束的定义。

使用默认值

要在列上设置默认值，使用DEFAULT约束。默认值无需为每一列指定值即可编写查询。

举例如下：

```
create table trans (
   id uuid DEFAULT gen_random_uuid() primary key,
   trnas_time TIMESTAMPTZ
  );
```

当向表中插入一行数据时，hubble 会为表生成一个随机默认值 id。

唯一约束

要防止列中出现重复值，使用UNIQUE约束。

示例如下：

```
create table users_info (
    first_name string,
    last_name  string,
    card_no    string UNIQUE
);
```

尝试插入表users_info中已存在的card_no值将返回错误。

非空值

要防止列中出现空值，使用NOT NULL约束。如果指定了一个NOT NULL约束，则所有针对具有该约束的表的查询都必须为该列指定一个值。

```
create table users_info (
    first_name string,
```

```
    last_name   string,
    card_no     string not null
);
```

5.1.3.2.5 执行 CREATE TABLE 语句

为表定义CREATE TABLE语句后，可以执行这些语句。

CREATE TABLE 执行事项

以下是执行CREATE TABLE语句时给予的建议：

- 不要以root用户身份创建表。
- 使用数据库架构迁移工具或客户端来执行数据库架构更改。
- 了解联机模式更改的限制。

建议在显式事务之外执行架构更改。

5.1.4 创建用户定义的架构

此页面提供有关创建用户定义模式的最佳指导。

5.1.4.1 准备工作

在阅读本页之前，请执行以下操作：

- 创建 Hubble 集群或启动本地集群。
- 查看数据库架构对象。
- 创建数据库。

5.1.4.2 创建用户定义的架构

5.1.4.2.1 用户定义的架构最佳实践

以下是创建和使用用户定义模式时的一些建议：

- 如果在集群中创建多个具有相同名称的对象（例如：表或视图），请在同一数据库中的不同用户定义模式中执行此操作。
- 如果出于访问或组织目的想要分离较低级别的对象（例如，一组表或视图），建议用户创建定义的架构，然后在用户定义的架构中创建对象。
- 不要在预加载的defaultdb数据库中创建用户定义的模式。
- 创建用户定义的架构时，记下对象的所有者。可以在带有关键字CREATE SCHEMA的语句中指定所有者。
- 当引用数据库中较低级别的对象（例如：表）时，如果数据库中有多个具有相同名称的对象，则在较低级别的对象引用中指定模式名称可以防止用户尝试访问错误的对象。

5.1.4.2.2 示例

若希望将集群中的表和索引分开，这样一个用户管理一组表，而另一个用户管理索引。

CREATE DATABASE下添加以下语句：

```
use test;

create user if not exists guar;
grant create on database test to guar;

create user if not exists aml94;
grant create on database test to aml94;
```

两组语句在数据库中创建名为guar和aml94的用户，对test数据库具有CREATE权限。CREATE特权将允许每个用户在数据库中创建表。

在此基础下，为每个用户的定义模式，用CREATE SCHEMA：

```
drop schema if exists guar_schema CASCADE;
create schema guar_schema  AUTHORIZATION guar;

drop schema if exists aml94_schema CASCADE;
create schema aml94_schema AUTHORIZATION aml94;
```

在每个用户定义模式的create schema语句下，添加一条GRANT语句将模式权限授予其他用户。

```
GRANT USAGE ON SCHEMA guar_schema  to aml94;

GRANT USAGE ON SCHEMA aml94_schema to guar;
```

5.1.5 列族

列族是表中的一组列，它们作为单个键值对存储在底层键值存储中。列族减少了键值存储中的键的数量，从而提高了INSERT，UPDATE，DELETE操作的性能。

本页解释了 Hubble 如何将列组织成系列。

以下介绍默认行为和手动操作的情况，鼓励手动操作。

5.1.5.1 默认行为

创建表时，所有列都存储为单个列族。

在大多数情况下，这种默认方法可确保高效的键值存储和性能。然而，当频繁更新的列与很少更新的列组合在一起时，很少更新的列仍然会在每次更新时被重写。特别是当不频繁更新的列很多时，将它们拆分成一个不同的系列会更高效。也就是说将频繁更新的列与不频繁更新的列放在不同的列族中。

5.1.5.2 手动操作

5.1.5.2.1 在创建表时分配列族

要在创建表时手动分配列族，请使用FAMILY关键字。

假定有一张基础的客户信息表，通常情况下姓名和身份证号是不变的，工作地址会变动，这样可以分为两个列族

```
create table cust (
    id INT PRIMARY KEY,
    name string,
    card_no string,
    address string,
    family f1 (id, name,card_no),
    family f2 (address)
);
```

```
show create cust;
```

```
  table_name |                create_statement
-------------+-----------------------------------------------
  cust       | CREATE TABLE public.cust (
             |     id INT8 NOT NULL,
             |     name STRING NULL,
             |     card_no STRING NULL,
             |     address STRING NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (id ASC),
             |     FAMILY f1 (id, name, card_no),
             |     FAMILY f2 (address)
             | )
(1 row)
```

5.1.5.2.2 添加列时分配列族

当使用ALTER TABLE .. ADD COLUMN语句向表中添加列时，可以将该列分配给新的或现有的列族。

- 使用CREATE FAMILY关键字将新列分配给**新列族**

```
alter table cust add column sex string create family f3;
```

- 使用CREATE FAMILY关键字将新列分配给**现有列族**

```
alter table cust add column tell_no string  family f1;
```

> 如果一个列被添加到表中并且没有指定族，它将被默认添加到第一个列族中。

5.1.6 二级索引

索引是一个逻辑对象，可以帮助 Hubble 更有效地查找数据。当创建索引时，Hubble 为索引选择的列的副本，然后按索引列值对数据行进行排序，而不对表本身中的值进行排序。

Hubble 自动在表的主键列上创建索引。该索引称为主索引。主索引帮助 Hubble 更有效地扫描按表的主键列排序的行，但它无助于查找由任何其他列标识的值。

二级索引（所有不是主索引的索引）提高了查询的性能，这些查询识别具有不在表主键中的列的行。Hubble 自动为具有UNIQUE约束的列创建二级索引。

本页提供了创建二级索引的最佳实践。

5.1.6.1 准备工作

在阅读本页之前，请执行以下操作：

- 创建 Hubble 集群
- 确认数据库架构对象
- 创建数据库
- 创建用户定义的架构
- 创建一个表

5.1.6.2 创建二级索引

要向表添加二级索引，请执行以下操作之一：

- 在CREATE TABLE语句末尾添加一个INDEX子句

INDEX子句通常采用以下形式：

```
index {index_name} ({column_names});
```

范围	描述
{index_name}	索引的名称。
{column_names}	索引的列的名称

- 用CREATE INDEX声明

CREATE INDEX语句通常采用以下形式：

```
create index {index_name} on {table_name} ({column_names});
```

范围	描述
{index_name}	索引的名称。
{column_names}	索引的列的名称
{table_name}	表的名称

5.1.6.3 最佳实践

5.1.6.3.1 索引内容

- 索引计划用于过滤数据的所有列。

基数较高的列（不同值的数量较多）应放在索引中基数较低的列之前。

- 如果需要索引应用于单个表的一个或多个列的函数的结果，请使用该函数创建一个计算列并对该列进行索引。
- 避免在顺序键上建立索引。

使用随机生成的唯一 ID 或多列键。

- 避免创建不需要的二级索引。

如果需要更改主键，并且不打算过滤现有主键列的查询，请不要使用ALTER PRIMARY KEY，因为它会从现有主键创建二级索引。相反，使用DROP CONSTRAINT ... PRIMARY KEY/ADD CONSTRAINT ... PRIMARY KEY，它不会

HUBBLE

创建二级索引。

5.1.6.3.2 索引管理

- 将二级索引的创建和删除限制在非高峰时间。如果在高峰工作时间完成，可能会影响性能。
- 不要以 root 用户身份创建索引。建议以不同的用户身份创建索引，权限更少。
- 尽可能删除未使用的索引。

> 我们强烈建议对所有索引列添加大小限制，其中包括主键中的列。超过 1MiB 的值会导致存储层写入放大，并会影响性能。

5.1.6.4 示例

示例数据如下

```
create type vtype as enum ('table tennis', 'basketball', 'football');

create table game (
      id UUID DEFAULT gen_random_uuid() PRIMARY KEY,
      type vtype,
      begin_time TIMESTAMPTZ DEFAULT now(),
      available bool,
      end_loc varchar(10)
  );
```

应用程序不会根据这些值进行过滤或排序。如果查询中的任何列不在主索引键中，Hubble 将需要对表执行完整扫描以查找值。全表扫描可能代价高昂，应尽可能避免。

为了帮助避免不必要的全表扫描，在索引中添加一个STORING子句：

```
create index type_available_idx on game (type, available) storing (end_loc);
```

5.1.7 计算列

计算列通过列定义中包含的表达式从其他列生成的数据。

存储计算列在插入或更新行时进行计算，并将结果值存储在主索引中，类似于非计算列。

5.1.7.1 使用计算列意义

JSONB计算列在与列或二级索引一起使用时特别有用。

- 可以在计算列上创建二级索引，这在经常对表进行排序时特别有用，以下有示例说明。
- JSONB列用于存储半结构化JSONB数据。计算列允许以下用例：一个包含一个列和一个JSONB列的表payload ↪ ，其主键是根据该payload列的一个字段计算得出的，以下有示例说明。

5.1.7.2 说明事项

计算列：

- 不能用于生成其他计算列。

- 行为与任何其他列一样，除了它们不能直接写入。

虚拟计算列：

- 不存储在表的主索引中。
- 当表达式中的列数据更改时重新计算。
- 不能用作FAMILY定义、CHECK约束或FOREIGN KEY约束的一部分。
- 不能是外键引用。
- 不能存储在索引中。
- 可以是索引列。

5.1.7.3 定义计算列

要定义存储计算列，请使用以下语法：

```
column_name <type> as (<expr>) stored
```

要定义虚拟计算列，请使用以下语法：

```
column_name <type> as (<expr>) virtual
```

范围	描述
{index_name}	计算列的名称。
<type>	计算列的数据类型。
<expr>	用于计算列值的不可变表达式。
stored	计算列与其他列一起存储。
virtual	计算列是虚拟的，这意味着列数据不存储在表的主索引中。

5.1.7.4 例子

5.1.7.4.1 创建一个包含存储计算列的表

```
create table cust_info (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        tell_no string,
        first_name string,
        last_name string,
        all_name string AS (concat(first_name, ' ', last_name)) stored,
        card_no string
);
```

```
insert into cust_info (first_name, last_name) values
    ('liu', 'dehua'),
    ('zhang', 'zijian'),
    ('ren', 'xianqi');
```

```
table cust_info;
```

```
                    id                   | tell_no | first_name | last_name |
                        ↪ all_name    | card_no
----------------------------------------+---------+------------+-----------+---------------+
   ↪
  1ff89b14-d27e-4ad8-b6e7-1d14f6a2dad6 | NULL    | liu        | dehua     | liu
      ↪ dehua     | NULL
  a4413c62-3f99-4bf3-b0a2-2f9770887304 | NULL    | zhang      | zijian    |
      ↪ zhang zijian | NULL
  b17080f0-4a73-4364-966f-89e29937a3fc | NULL    | ren        | xianqi    | ren
      ↪ xianqi    | NULL
```

该all_name列是根据first_name和last_name列计算的

5.1.7.4.2 创建一个表，其中包含一个 JSONB 列和一个存储的计算列

创建一个包含一JSONB列和一个存储的计算列的表：

此示例显示如何添加具有强制类型的存储计算列：

```
create table json_test (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    jsoninfor JSONB
);

insert into json_test (jsoninfor) VALUES ('{"aml": "111.11"}');

alter table json_test ADD COLUMN aml DECIMAL AS ((jsoninfor->>'aml')::DECIMAL)
   ↪ stored;
```

```
select * from json_test;
```

```
                    id                   |     jsoninfor     |  aml
----------------------------------------+-------------------+---------
  8d3968da-ed51-42f0-9446-13f447da4d53 | {"aml": "111.11"} | 111.11
```

5.1.7.4.3 使用 JSONB 数据创建虚拟计算列

在此示例中，创建一个包含JSONB列和虚拟计算列的表：

```
create table custinfo (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    testdata jsonb,
    all_name string AS (concat_ws(' ',testdata->>'firstName', testdata->>'
       ↪ lastName')) VIRTUAL
);
```

```
insert into custinfo (testdata) VALUES
    ('{"id": "d11111", "firstName": "Arthur", "lastName": "Read", "school": "
        ↪ PVPHS", "credits": 100, "sports": "none"}'),
    ('{"firstName": "Buster", "lastName": "Bunny",  "id": "f92222", "school": "
        ↪ THS", "credits": 24, "clubs": "MUN"}'),
    ('{"firstName": "Ernie", "lastName": "Narayan", "school" : "Brooklyn Tech",
        ↪ "id": "t00074", "sports": "Track and Field", "clubs": "Chess"}');
```

```
select * from custinfo;
```

```
                    id                   |
                        ↪
                        ↪ testdata
                        ↪
                        ↪ |    all_name
----------------------------------------+-----------------------------------------------
   ↪
  308b8137-a20f-43f8-94a8-a66a79dea124 | {"clubs": "MUN", "credits": 24, "
      ↪ firstName": "Buster", "id": "f92222", "lastName": "Bunny", "school": "
      ↪ THS"}                              | Buster Bunny
  613ce51c-b25b-4dec-8958-317d56425a72 | {"credits": 100, "firstName": "Arthur",
      ↪  "id": "d11111", "lastName": "Read", "school": "PVPHS", "sports": "none
      ↪ "}                           | Arthur Read
  e1d9c1e0-0e89-410f-b46e-9ed265466db5 | {"clubs": "Chess", "firstName": "Ernie
      ↪ ", "id": "t00074", "lastName": "Narayan", "school": "Brooklyn Tech", "
      ↪ sports": "Track and Field"} | Ernie Narayan
```

虚拟列all_name的计算来自testdata列数据的字段

5.1.7.4.4 在计算列上创建具有二级索引的表

创建一个表，其中包含虚拟计算列和该列的索引：

```
create table totalby (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tname STRING,
    a decimal,
    b decimal,
    c decimal,
    d decimal,
    t_score DECIMAL as (a + b + c + d) VIRTUAL,
    index total ( t_score desc)
  );
```

然后，插入几行数据：

```
insert into totalby (tname, a , b , c , d) values
    ('zahngsan', 15.111, 14.823, 25.600, 12.400),
```

```
    ('lisi', 0, 15.766, 0, 0),
    ('wangwu', 14.500, 0, 16.243, 14.563),
    ('liuliu', 0, 15.453, 0, 0),
    ('cluo', 11.233, 0, 12.440, 14.116);
```

现在，使用二级索引运行查询：

```
select tname,t_score from totalby  order by t_score desc;
```

```
   tname    | t_score
-----------+----------
  zahngsan |  67.934
  wangwu   |  45.306
  cluo     |  37.789
  lisi     |  15.766
  liuliu   |  15.453
```

5.1.7.4.5 将计算列添加到现有表

创建一个表：

```
create table abc (
    a INT NULL,
    b INT NULL AS (a * 3) stored,
    c INT NULL AS (a + 5) stored,
    FAMILY "primary" (a, b, rowid, c)
  );
```

然后，插入一行数据：

```
insert into abc values (2);
```

```
table abc;
```

```
  a | b | c
----+---+----
  2 | 6 | 7
  2 | 6 | 7
```

现在向表中添加另一个虚拟计算列：

```
alter table  abc add column d int AS (a // 2) VIRTUAL;
```

```
table abc;
```

```
  a | b | c | d
----+---+---+----
  2 | 6 | 7 | 1
  2 | 6 | 7 | 1
```

5.1.7.4.6 更改计算列的公式

数据准备:

```
create table abc (
    a INT NULL,
    b INT NULL AS (a * 3) stored,
    c INT NULL AS (a + 5) stored,
    FAMILY "primary" (a, b, rowid, c)
  );
```

```
alter table  abc add column d int AS (a // 2) stored;
```

要更改计算列的公式，必须DROP列，并用ADD语句重新定义列:

```
set sql_safe_updates = false;
alter table abc drop column d;
alter table abc add  column d int as (a // 4) stored;
set sql_safe_updates = true;
```

5.1.7.4.7 将计算列转换为常规列

可以使用ALTER TABLE语句将存储的计算列转换为常规列。

创建一个带有计算列的表:

```
create table zoos (
    id INT PRIMARY KEY,
    first_name string,
    last_name string,
    all_name string AS (CONCAT(first_name, ' ', last_name)) stored
  );
```

表中数据准备

```
insert into zoos (id, first_name, last_name) values
    (10, 'pa', 'hawei'),
    (20, 'ma', 'li'),
    (30, 'da', 'xinging');
```

```
select * from zoos;
```

```
  id | first_name | last_name |  all_name
-----+------------+-----------+-------------
  10 | pa         | hawei     | pa hawei
  20 | ma         | li        | ma li
  30 | da         | xinging   | da xinging
```

该all_name列是根据first_name和last_name列计算得来的

```
show columns from zoos;
```

```
  column_name | data_type | is_nullable | column_default |
     ↪ generation_expression        |  indices   | is_hidden
--------------+-----------+-------------+----------------+-------------------------------
   ↪
  id           | INT8      |    false    | NULL           |
     ↪                                   | {primary} |   false
  first_name   | STRING    |    true     | NULL           |
     ↪                                   | {}        |   false
  last_name    | STRING    |    true     | NULL           |
     ↪                                   | {}        |   false
  all_name     | STRING    |    true     | NULL           | concat(first_name, '
     ↪ ', last_name) | {}        |   false
```

将计算列all_name转换为常规列:

```
alter table zoos alter column all_name drop stored;
```

检查是否转换成功

```
show columns from zoos;
```

```
  column_name | data_type | is_nullable | column_default |
     ↪ generation_expression        |  indices   | is_hidden
--------------+-----------+-------------+----------------+-------------------------------
   ↪
  id           | INT8      |    false    | NULL           |
     ↪                                   | {primary} |   false
  first_name   | STRING    |    true     | NULL           |
     ↪                                   | {}        |   false
  last_name    | STRING    |    true     | NULL           |
     ↪                                   | {}        |   false
  all_name     | STRING    |    true     | NULL           |
     ↪                                   | {}        |   false
```

重新准备数据

```
insert into zoos (id, first_name, last_name,all_name) values (40, 'paper', '
   ↪ tiger', 'This is null');
```

再次展示数据

```
select * from zoos;
```

```
  id | first_name | last_name |   all_name
-----+------------+-----------+---------------
  10 | pa         | hawei     | pa hawei
  20 | ma         | li        | ma li
  30 | da         | xinging   | da xinging
  40 | paper      | tiger     | This is null
```

5.1.8 索引行的子集

部分索引允许指定要添加到索引的行和列的子集。

5.1.8.1 部分索引如何工作

当创建部分索引时，Hubble 会在索引的布尔谓词表达式中的列和行评估为true，创建行值子集的排序副本，而不修改表本身中的值。

Hubble 可以使用部分索引对部分索引隐含的任何行子集高效地执行查询。如果可能，基于成本的优化器会创建一个计划，将对部分索引隐含的行的表扫描限制为仅索引中的行。

部分索引可以通过多种方式提高集群性能：

- 它们包含的行数少于完整索引，因此在集群上创建和存储它们的成本更低。
- 对包含在部分索引中的行的读取查询仅扫描部分索引中的行。
- 对具有部分索引的表的写入查询仅在插入的行满足部分索引谓词时才执行索引写入。这与对具有完整索引的表的写入查询形成对比，后者在插入的行修改索引列时会产生完整索引写入的开销。

以下有具体示例。

5.1.8.2 创建部分索引

要创建部分索引，请使用带有定义谓词WHERE表达式CREATE INDEX的标准子句的语句。

例如，要在表的列x和y表上定义部分索引，过滤列中z大于1的行：

```
create index on test (x, y) where z > 1;
```

以下查询使用部分索引：

```
select x,y from test where z > 1;
```

```
select x,y from test where z = 2;
```

以下查询不使用部分索引：

```
select x,y from test;
```

```
select x,y from test where z = 0;
```

在定义谓词表达式时，请遵循：

- 谓词表达式必须产生一个布尔值。
- 谓词表达式只能引用被索引表中的列。
- 谓词中使用的函数必须是不可变的。例如，now()函数在谓词中是不允许的。

5.1.8.3 唯一的部分索引

可以使用对行的子集强制唯一性CREATE UNIQUE INDEX ... WHERE ...语句。

要在列a和b表上定义唯一的部分索引，过滤列中c等于的行'n'：

```
create unique index on t (a, b) where c = 'n';
```

这会创建一个部分索引和对等于UNIQUE行子集的约束。

5.1.8.4 索引提示

查询可以强制使用特定的部分索引 (也称为'索引提示')，就像使用完整索引一样。但是，与全索引不同，部分索引不能用于满足所有查询。

5.1.8.5 限制

- Hubble 目前不支持导入带有部分索引的表的语句。
- Hubble 目前不支持多个索引的INSERT ON CONFLICT DO UPDATE语句。

5.1.8.6 示例

5.1.8.6.1 创建一个部分索引，在行的子集上强制执行唯一性

```
create table cust_info (
                  id UUID not null DEFAULT gen_random_uuid(),
                  name string NULL,
                  addr string);

insert into cust_info(id, addr, name) values (gen_random_uuid(), 'beijing', '
    ↪ daxing');
insert into cust_info(id, addr, name) values (gen_random_uuid(), 'beijing', '
    ↪ fangshan');
insert into cust_info(id, addr, name) values (gen_random_uuid(), 'beijing', '
    ↪ chaoyang');
```

要限制表中行的子集，以便该子集中特定列的所有值都是唯一的。

例如，地点的每个用户都必须有一个唯一的名字。

```
create unique index on cust_info(name) where addr='beijing';
```

这将创建一个部分索引和仅对行子集的UNIQUE约束

```
insert into cust_info(id, addr, name) values (gen_random_uuid(), 'beijing', '
    ↪ chaoyang');
```

```
ERROR: duplicate key value violates unique constraint "cust_info_name_key"
SQLSTATE: 23505
DETAIL: Key (name)=('chaoyang') already exists.
CONSTRAINT: cust_info_name_key
```

因为唯一部分索引谓词仅暗示行where addr='beijing'，所以UNIQUE约束不适用于表中的所有行。

5.1.9 倒排索引

广义倒排索引或 GIN索引存储从容器列 (例如JSONB文档) 中的值到包含该值的行的映射。GIN索引通常用于文档检索系统。

Hubble 在 GIN索引中存储以下数据类型的内容：

- JSONB
- 数组

5.1.9.1 GIN 索引如何工作

标准索引适用于基于排序数据前缀的搜索。但是，如果不进行全表扫描，就无法查询像JSONB数组这样的数据类型。JSONB数组特别是需要以比标准索引提供的更详细的方式进行索引，这就是GIN索引被证明有用的地方。

GIN索引过滤可标记数据的组成部分。JSONB数据类型建立在两个可以标记化的结构上：

- 对象：键值对的集合，其中每个键值对都是一个标记。
- 数组：有序的值列表，其中数组中的每个值都是一个标记。

例如，JSONB在列中取以下值

```
{
  "firstName": "mike",
  "lastName": "jackson",
  "number": 10,
  "address": {
    "state": "USA",
    "postCode": "10000"
  },
  "sports": [
    "byd",
    "aodi"
  ]
}
```

该对象的GIN索引每个组件都有一个条目，将其映射回原始对象：

```
"firstName": "mike"
"lastName": "jackson"
"number": 10
"address": "state": "USA"
"address": "postCode": "10000"
"sports" : "byd"
"sports" : "aodi"
```

HUBBLE

5.1.9.1.1 创建

可以使用GIN索引来提高使用JSONB或ARRAY列的查询的性能。

- 可以使用以下语法指定jsonb_ops或array_ops操作类（分别用于JSONB和ARRAY列）：

```
CREATE INDEX {optional name} ON {table} USING GIN ({column} {opclass});
```

- 创建表时，使用语法CREATE INVERTED INDEX：

```
CREATE INVERTED INDEX {optional name} ON {table} ({column});
```

5.1.9.1.2 选择

在大多数情况下，Hubble 选择它计算出的索引将扫描最少的行 (即最快的)。Hubble 将使用多个索引的情况包括某些谓词的查询。可以使用EXPLAIN查询语句来查看正在使用哪个索引。

5.1.9.1.3 存储

Hubble 将索引直接存储在键值存储中。

5.1.9.1.4 表现

索引产生了一种权衡：它们极大地提高了查询速度，但略微减慢了写入速度（因为必须复制和排序新值）。

5.1.9.1.5 比较

本节介绍如何对JSONB和ARRAY列进行比较。

JSONB

列上的GIN索引支持以下比较运算符：

- 包含在：<@
- 包含：@>
- 等于：=。要使用=，还必须与->一起使用。例如：

```
SELECT * FROM s WHERE a ->'foo' = '"1"';
```

示例如下：

1. 创建一个包含计算列的表：

```
create table td (
    id int,
    data json,
    t int as ((data->>'t')::int) stored
    );
```

2. 在计算列上创建索引：

```
create index td_idx on td (t);
```

3. 执行查询：

```
select * from td where t > 1;
```

数组

列上的GIN索引支持以下比较运算符：

包含在：<@ 包含：@>

5.1.9.2 部分 GIN 索引

就像使用非容器数据类型的部分索引一样，可以通过包含一个子句的方式来创建部GIN索引：

```
create table  t1 (
  id INT,
  da JSONB,
  INVERTED INDEX idx_da(da) WHERE id > 1
);
```

HUBBLE

5.1.9.3 多列 GIN 索引

创建具有多个列的GIN索引：

```
create table cust_info (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  c_type STRING,
  c_profile JSONB,
  INVERTED INDEX (c_type, c_profile)
);
```

5.1.9.4 示例

5.1.9.4.1 在 JSONB 列上创建一个带有 GIN 索引的表

```
create table cust_info (
    profile_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    login_time TIMESTAMP DEFAULT now(),
    c_profile JSONB,
    INVERTED INDEX c_ids (c_profile)
  );

insert into cust_info (c_profile) values
    ('{"first_name": "Lola", "last_name": "Dog", "location": "NYC", "online" :
       ↪ true, "friends" : 547}'),
    ('{"first_name": "Ernie", "status": "Looking for treats", "location" : "
       ↪ Brooklyn"}'),
    ('{"first_name": "Carl", "last_name": "Kimball", "location": "NYC", "breed":
       ↪  "Boston Terrier"}'
  );
```

现在，运行对JSONB列进行过滤的查询：

```
select  * from cust_info where c_profile @> '{"location":"NYC"}';
```

```
               profile_id              |         login_time         |
                  ↪                                                  c_profile
----------------------------------------+----------------------------+-----------------------
   ↪
  572e728c-55f2-4645-9d52-574dae4f5745 | 2022-12-23 15:12:39.329944 | {"
     ↪ first_name": "Lola", "friends": 547, "last_name": "Dog", "location": "
     ↪ NYC", "online": true}
  e9ef1f94-2d2d-4311-b281-d86520f5deeb | 2022-12-23 15:12:39.329944 | {"breed":
     ↪ "Boston Terrier", "first_name": "Carl", "last_name": "Kimball", "
     ↪ location": "NYC"}
```

查看执行计划

```
explain select  * from cust_info where c_profile @> '{"location":"NYC"}';
```

```
                info
--------------------------------
  distribution: local
  vectorized: false

  • index join
    table: cust_info@primary

      • scan
        missing stats
        table: cust_info@c_ids
        spans: 1 span
```

5.1.9.4.2 将 GIN 索引添加到具有数组列的表

在这个例子中，先创建一个包含ARRAY列的表，然后再添加GIN索引:

```
create table stu (
    id  UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    bins INT ARRAY
  );

insert into stu (bins) values
    (ARRAY[10,20,50]),
    (ARRAY[20,30,60]),
    (ARRAY[30,40,70]
  );
```

未添加索引前，查看执行计划:

```
explain select * from stu where bins @> ARRAY[10];
```

```
              info
--------------------------------
  distribution: full
  vectorized: false

  • filter
    filter: bins @> ARRAY[10]

      • scan
        missing stats
        table: stu@primary
        spans: FULL SCAN
```

向表中添加一个 GIN索引并运行一个过滤查询:

```
create inverted index stu_bins on stu (bins);
```

```
select * from stu where bins @> ARRAY[10];
```

```
                    id                    |     bins
------------------------------------------+-------------
  d864a089-d310-408e-bcca-19084333f771 | {10,20,50}
```

再次查看执行计划:

```
 explain select * from stu where bins @> ARRAY[10];
```

```
            info
-----------------------------
  distribution: local
  vectorized: false

  • index join
    table: stu@primary

      • scan
        missing stats
        table: stu@stu_bins
        spans: 1 span
```

5.1.9.4.3 在 JSONB 列上创建具有 GIN 索引的表

创建一个部分GIN索引:

```
create inverted index idx_online on cust_info(c_profile) where c_profile -> '
    ↪ online' = 'true';
```

```
select * from cust_info where c_profile -> 'online' = 'true';
```

```
               profile_id               |          login_time          |
                   ↪                                                    c_profile
----------------------------------------+------------------------------+-----------------------
    ↪
  572e728c-55f2-4645-9d52-574dae4f5745 | 2022-12-23 15:12:39.329944 | {"
      ↪ first_name": "Lola", "friends": 547, "last_name": "Dog", "location": "
      ↪ NYC", "online": true}
```

查看执行计划:

```
explain select * from cust_info where c_profile -> 'online' = 'true';
```

```
              info
--------------------------------
  distribution: local
  vectorized: false
```

```
  • index join
    table: cust_info@primary

      • scan
        missing stats
        table: cust_info@c_ids
        spans: 1 span
```

5.1.10 更改或删除数据库中的对象

本页概述了如何更改和删除数据库模式中的对象。

5.1.10.1 准备

在阅读本页之前，请执行以下操作：

- 创建 Hubble 集群或启动本地集群。
- 查看数据库架构对象。
- 创建数据库。
- 创建用户定义的架构。
- 创建一个表。
- 添加二级索引。
- 改变数据库。

5.1.10.2 改变数据库模式对象

要更改数据库模式中的现有对象，使用ALTER语句。

ALTER语句通常采用以下形式：

```
ALTER {OBJECT_TYPE} {OBJECT_NAME} {SUBCOMMAND};
```

范围	描述
{OBJECT_TYPE}	对象的类型。
{OBJECT_NAME}	对象的名称。
{SUBCOMMAND}	要进行的更改的子命令。

hubble 支持以下ALTER语句：

- ALTER DATABASE
- ALTER SCHEMA
- ALTER TABLE
- ALTER INDEX
- ALTER VIEW
- ALTER SEQUENCE
- ALTER TYPE
- ALTER USER/ROLE

HUBBLE

5.1.10.2.1 改变对象的例子

要对在创建表中创建的表进行一些更改。具体来说，执行以下操作：

- 添加一个新列:name。
- 将表的主键中的更改为name和email列。
- 将表移动到abc_schema用户定义的模式。
- 将表的所有者更改为abc。

该ALTER TABLE语句具有针对所有这些更改的子命令：

- 要添加新列，请使用ADD COLUMN子命令。
- 要更改表的主键列，请使用ALTER PRIMARY KEY子命令。
- 要将表移动到不同的模式，请使用SET SCHEMA子命令。
- 要更改表的所有者，请使用OWNER TO子命令。

以下示例说明

添加ALTER TABLE添加列的语句name：

```
ALTER TABLE  cust_info ADD COLUMN name string;
```

添加另一个ALTER TABLE语句以将主键列更改为name和email：

```
ALTER TABLE  cust_info ALTER PRIMARY KEY USING COLUMNS (name, email);
```

为了将列添加到现有表的主键索引中，该列必须具有现有NOT NULL约束。综合来讲，添加一个子命令来设置列上的约束：

```
ALTER TABLE cust_info
  ADD   COLUMN name STRING NOT NULL,
  ALTER COLUMN email SET NOT NULL;
```

执行主键添加：

```
ALTER TABLE cust_info ALTER PRIMARY KEY USING COLUMNS (name, email);
```

更改模式的语句：

```
ALTER TABLE IF EXISTS cust_info SET SCHEMA abc_schema;

ALTER TABLE IF EXISTS test.abc_schema.cust_info OWNER TO abc;
```

5.1.10.3 删除数据库模式对象

要从数据库模式中删除对象，请使用DROP语句。

DROP语句通常采用以下形式：

```
DROP {OBJECT_TYPE} {OBJECT_NAME} CASCADE;
```

范围	描述
{OBJECT_TYPE}	对象的类型。
{OBJECT_NAME}	对象的名称。

范围	描述
{CASCADE}	一个可选的关键字，它将删除依赖于被删除对象的所有对象。

hubble 支持以下DROP语句：

- DROP DATABASE
- DROP SCHEMA
- DROP TABLE
- DROP INDEX
- DROP SEQUENCE
- DROP VIEW
- DROP USER/ROLE

5.1.10.3.1 删除最佳实践

- 在使用该选项之前检查要删除的对象的内容和依赖项。CASCADE语句删除对象的所有内容，在模式初始化后应谨慎使用。

5.1.10.3.2 示例

假设想删除一个不经常使用的索引。

```
show indexes from cust_info;
```

```
 table_name  |           index_name                | non_unique | seq_in_index |
    ↪ column_name | direction | storing | implicit  | visible
-------------+-------------------------------+------------+--------------+-------------+-
   ↪
  cust_info  | cust_info_pkey                      |    f       |            1 |
     ↪ name        | ASC       |    f     |    f      |   t
  cust_info  | cust_info_pkey                      |    f       |            2 |
     ↪ email       | ASC       |    f     |    f      |   t
  cust_info  | cust_info_first_name_last_name_key |    f       |            1 |
     ↪ first_name  | ASC       |    f     |    f      |   t
  cust_info  | cust_info_first_name_last_name_key |    f       |            2 |
     ↪ last_name   | ASC       |    f     |    f      |   t
  cust_info  | cust_info_first_name_last_name_key |    f       |            3 |
     ↪ name        | ASC       |    f     |    t      |   t
  cust_info  | cust_info_first_name_last_name_key |    f       |            4 |
     ↪ email       | ASC       |    f     |    t      |   t
  cust_info  | cust_info_email_key                 |    f       |            1 |
     ↪ email       | ASC       |    f     |    f      |   t
  cust_info  | cust_info_email_key                 |    f       |            2 |
     ↪ name        | ASC       |    f     |    t      |   t
```

```
show create table test.abc_schema.cust_info;
```

```
CREATE TABLE abc_schema.cust_info (
                                first_name STRING NOT NULL,
                                last_name STRING NOT NULL,
                                email STRING NOT NULL,
                                name STRING NOT NULL,
                                CONSTRAINT "primary" PRIMARY KEY (name ASC, email
                                    ↪  ASC),
                                UNIQUE INDEX cust_info_first_name_last_name_key (
                                    ↪ first_name ASC, last_name ASC),
                                UNIQUE INDEX cust_info_email_key (email ASC),
                                FAMILY "primary" (first_name, last_name, email,
                                    ↪ name)
                            );
```

cust_info_first_name_last_name_key是一个UNIQUE索引。

执行DROP语句:

```
DROP INDEX  test.abc_schema.cust_info@cust_info_first_name_last_name_key
    ↪ CASCADE;
```

5.2 读取数据

5.2.1 子查询

SQL 子查询允许在另一个查询中重用选择查询的结果。

Hubble 支持两种子查询:

- 关系子查询，在选择查询和表表达式中显示为操作数。
- 标量子查询，在标量表达式中显示为操作数。

5.2.1.1 子查询中的数据写入

当子查询包含数据修改语句时，即使周围查询仅使用结果行的子集，数据修改也始终执行完成。

这适用于使用(...)符号定义的子查询和使用WITH定义的子查询。

例如:

```
WITH a AS (INSERT INTO b(c) VALUES (10), (20), (30))
SELECT * FROM a LIMIT 1;
```

5.2.1.2 相关子查询

当子查询使用在周围查询中定义的表名或列名时，它被称为是相关的。

例如，要查找至少有一个订单的客户:

```
select
      c.name
  from
      customer as c
```

```
  where
      exists (select * from orders as o where o.c_id = c.id);
```

5.2.1.2.1 LATERAL 子查询

LATERAL子查询是在在其语句中引用另一个查询或子查询的相关select子查询，通常在left join或inner join ↪ 的上下文中。与其他相关子查询不同，LATERAL子查询在引用查询中为内部子查询中的每一行迭代，就像for循环一样。

要创建LATERAL子查询，请直接在内部子查询SELECT语句之前使用LATERAL关键字。

```
select empno,ename from emp , LATERAL (select * from dept  d where d.deptno =
    ↪ emp.deptno and  deptno = '50');
```

```
 empno | ename
--------+---------
   7580 | WAL%ER
```

5.2.1.2.2 性能最佳实践

当其他查询开始执行时，标量子查询的结果完全加载到内存中。为防止由于内存耗尽而导致执行错误，请确保子查询返回尽可能少的结果。

5.2.2 物化视图与临时视图

视图是一个存储和命名的选择查询。默认情况下，Hubble 的视图是非实体化的：它们不存储底层查询的结果。相反，每次使用视图时都会重新执行基础查询。

其中，Hubble 还支持物化视图，物化视图是存储其选择查询结果的视图。

普通视图

- 视图可以理解为一张表或多张表的预计算，这些表称为基表。
- 它可以将所需要查询的结果封装成一张虚拟表，基于它创建时指定的查询语句返回的结果集。

物化视图

- 为了防止每次都查询，将结果集存储起来，这种有真实数据的视图，称为物化视图。

物化视图的适用情况

基表没有很多 DML，且每次查询都需要耗费较大资源的情况下，可以考虑用物化视图。

5.2.2.1 视图的作用

使用视图有多种原因，包括：

- 安全性：只将需要的结果呈现出来，查询者不知道具体用了哪些表或哪些字段，因此比较安全；
- 屏蔽复杂性：下层计算可能做了很多复杂的关联操作，只需要让开发者将其实现，将结果以视图呈现给使用者。

HUBBLE

5.2.2.1.1 隐藏查询复杂性

当有一个复杂的查询，例如，连接多个表或执行复杂的计算时，您可以将查询存储为视图，然后像从标准表中一样从视图中进行选择。

比如要查询每个部门下员工的人数：

```
select d.dname ,count(1) from
dept d
join emp e
on  d.deptno=e.deptno
group by d.dname;
```

```
    dname     | count
-------------+--------
  OPERATIONS |     1
  SALES      |     6
  BOSS       |     1
  ACCOUNTING |     8
  RESEARCH   |     9
```

为了更轻松地运行这个复杂的查询，可以创建一个视图：

```
create view dname_c (dname,num)
as
select d.dname ,count(1) from
dept d
join emp e
on  d.deptno=e.deptno
group by d.dname;
```

然后，执行查询就像SELECT单表语句一样简单：

```
select * from dname_c;
```

```
    dname     | count
-------------+--------
  OPERATIONS |     1
  SALES      |     6
  BOSS       |     1
  ACCOUNTING |     8
  RESEARCH   |     9
```

5.2.2.1.2 限制对基础数据的访问

当您不想授予用户访问一个或多个标准表中所有数据的权限时，您可以创建一个仅包含用户应该有权访问的的视图。

示例：假设有一张客户信息表cust_info：

```
select * from cust_info;
```

HUBBLE

```
 cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
    ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435550 | 王吉      | 12022519960321531X   | 15122511874  | 天津武清     | 抵押
  14435551 | 张贺      | 431256197306265320   | 15534343555  | 山西临汾     | 质押
  14435552 | 刘明      | 371452199303034312   | 18967756743  | 陕西延安     | 信用
  14435553 | 李华      | 52112119860621421X   | 15833355455  | 湖北武汉     | 抵押
  14435554 | 郑青      | 213456199102275341   | 13054546567  | 江西南昌     | 质押
```

您希望特定用户abc，能够查看每个用户拥有的帐户类型，而无需查看每个帐户中的身份证号，手机号，因此您创建一个视图以仅显示cust_type和cust_name列：

```
create view users
as
select cust_type, cust_name
from cust_info;
```

您授予abc视图users访问权限：

```
grant select on users to abc;
```

现在，abc在尝试访问基础cust_info表时会出现权限错误：

```
select * from cust_info;
```

```
pq: user abc does not have SELECT privilege on table cust_info
```

```
select * from users;
```

```
  cust_type | cust_name
------------+------------
  抵押      | 王吉
  质押      | 张贺
  信用      | 刘明
  抵押      | 李华
  质押      | 郑青
```

5.2.2.2 视图的工作原理

5.2.2.2.1 创建视图

```
create view dname_c (dname,num)
as
select d.dname ,count(1) from
dept d
join emp e
on  d.deptno=e.deptno
group by d.dname;
```

HUBBLE

任何选择查询都可以CREATE VIEW，而不仅仅是简单的SELECT子句。

5.2.2.2.2 列出视图

创建后，视图将与数据库中的常规表，序列等一起列出：

```
show tables from ora;
```

```
public      | user_table   | table    | root   |                  6 | NULL
public      | userid_seq   | sequence | root   |                  1 | NULL
public      | users        | view     | root   |                  0 | NULL
public      | weihai       | table    | root   |                  0 | NULL
```

要仅列出视图，您可以查询Information Schemaviews中的表：

```
select * from  information_schema.views;
```

5.2.2.2.3 查询视图

要查询视图，请使用表表达式作为目标，例如使用SELECT子句，就像使用存储表一样：

```
select * from users;
```

```
  cust_type | cust_name
------------+------------
  抵 押      | 王 吉
  质 押      | 张 贺
  信 用      | 刘 明
  抵 押      | 李 华
  质 押      | 郑 青
```

SELECT视图执行视图存储的SELECT语句，该语句从基础表返回相关数据。要检查的SELECT视图执行的语句，请使用以下SHOW CREATE语句：

```
show create users;
```

```
  table_name |                                                create_statement
-------------+-------------------------------------------------------------------------------------
   ↪
  users      | CREATE VIEW public.users (cust_type, cust_name) AS SELECT
     ↪ cust_type, cust_name FROM ora.public.cust_info
```

您还可以通过查询Information Schema中的表来检查SELECT视图执行的语句：

```
SELECT view_definition FROM information_schema.views WHERE table_name = 'users';
```

```
                    view_definition
-------------------------------------------------------
  SELECT cust_type, cust_name FROM ora.public.cust_info
```

5.2.2.2.4 查看依赖项

视图取决于其基础查询所针对的对象。因此，尝试重命名视图的存储查询中引用的对象会导致错误：

```
alter table cust_info rename to custs;
```

```
ERROR: cannot rename relation "cust_info" because view "users" depends on it
SQLSTATE: 2BP01
HINT: you can drop users instead.
```

同样，尝试删除视图的存储查询中引用的对象会导致错误：

```
drop table cust_info;
```

```
ERROR: cannot drop relation "cust_info" because view "users" depends on it
SQLSTATE: 2BP01
HINT: you can drop users instead.
```

```
alter table cust_info drop column cust_name;
```

```
ERROR: rejected (sql_safe_updates = true): ALTER TABLE DROP COLUMN will remove
    ↪ all data in that column
SQLSTATE: 01000
```

同样，因为没有视图依赖于表的cust_address列，所以可以删除它。要删除其中包含数据的列，您必须首先设置sql_safe_updates = false

```
set sql_safe_updates = false;
```

```
alter table cust_info drop column cust_address;
```

```
show  columns from cust_info;
```

```
  column_name  |  data_type  | is_nullable | column_default |
    ↪ generation_expression |                indices                |
---------------+-------------+-------------+----------------+-----------------------+-----
  ↪
  cust_no      | STRING      |    false    | NULL           |
    ↪                        | {cust_info_cust_card_no_idx,primary} |
  cust_name    | VARCHAR(30) |    false    | NULL           |
    ↪                        | {}                                   |
  cust_card_no | VARCHAR(18) |    true     | NULL           |
    ↪                        | {cust_info_cust_card_no_idx}         |
  cust_phoneno | DECIMAL(15) |    true     | NULL           |
    ↪                        | {}                                   |
  cust_type    | VARCHAR(10) |    true     | NULL           |
    ↪                        | {}                                   |
```

删除表或删除视图时，您也可以使用CASCADE关键字删除所有依赖对象：

```
drop table cust_info CASCADE;
```

5.2.2.2.5 重命名视图

要重命名视图，请使用以下ALTER VIEW语句：

```
alter view users rename to userinfo;
```

5.2.2.2.6 替换视图

要替换视图，请使用CREATE OR REPLACE VIEW：

```
create or  replace view user_info
as
select cust_type, cust_name
from cust_info;
```

```
select * from user_info limit 5;
```

```
  cust_type | cust_name
------------+------------
  抵押      | 王吉
  质押      | 张贺
  信用      | 刘明
  抵押      | 李华
  质押      | 郑青
```

5.2.2.2.7 删除视图

要删除视图，请使用以下DROP VIEW语句：

```
drop view user_info;
```

5.2.2.3 物化视图

hubble 支持物化视图。物化视图是存储其基础查询结果的视图。

因为物化视图存储查询结果，它们提供比标准视图更好的性能，但代价是存储查询结果所需的额外存储空间和保证结果是最新的。

5.2.2.3.1 用法

物化视图和标准视图在创建、显示、重命名和删除方面共享相似的语法。

要创建物化视图，请使用CREATE MATERIALIZED VIEW语句。

示例：

```
CREATE MATERIALIZED VIEW emppeo
as select empno, ename
  from emp where deptno = 10 ;
```

```
select * from emppeo;
```

HUBBLE

```
 empno |  ename
--------+----------
   9988 | laihui
   7839 | KING
   1357 | FREE
   7367 | PA_I%RS
   7934 | MILLER
   7782 | CLARK
   7777 | aaa
   5555 | kettle
```

现在假设您更新了表

```
delete from emp where deptno =10;
```

现在查询表emp里面部门号位 10 的数据是不存在的

```
select count(1) from emp where deptno =10;
```

```
  count
---------
      0
```

回想一下，物化视图不会自动更新其存储的结果。此时任然返回数据

```
select * from emppeo;
```

```
 empno |  ename
--------+----------
   9988 | laihui
   7839 | KING
   1357 | FREE
   7367 | PA_I%RS
   7934 | MILLER
   7782 | CLARK
   7777 | aaa
   5555 | kettle
```

要更新物化视图的结果，请使用以下REFRESH语句：

```
REFRESH MATERIALIZED VIEW emppeo;
```

```
select * from emppeo;
```

```
 empno |  ename
--------+----------
```

要删除物化视图，请使用DROP MATERIALIZED VIEW：

```
DROP MATERIALIZED VIEW emppeo;
```

HUBBLE

5.2.2.3.2 向物化视图添加索引

要加快对物化视图的查询，您可以向视图添加索引：

```
create index on emppeo (empno) STORING (ename);
```

查看添加索引后计划的变化：

```
explain select ename from emppeo where empno ='1234';
```

```
                info
------------------------------------
  distribution: local
  vectorized: false

  • scan
    missing stats
    table: emppeo@emppeo_empno_idx
    spans: [/1234 - /1234]
```

5.2.2.4 临时视图

5.2.2.4.1 细节

- 会话结束时会自动删除临时视图。
- 临时视图只能从创建它的会话中访问。
- 临时视图在同一会话中的事务中持续存在。
- 临时视图不能转换为持久视图。
- 可以在持久表和临时表上创建临时视图。
- 在临时表上创建视图时，视图会自动变为临时视图。

5.2.2.4.2 用法

要创建临时视图，请将TEMP添加到CREATE VIEW语句中。并且先设置变量SET experimental_enable_temp_tables = 'on'。

例如：

先设置变量：

```
SET experimental_enable_temp_tables = 'on'
```

创建临时视图：

```
create temp view dname_c (dname,num)
as
select d.dname ,count(1) from
dept d
join emp e
on  d.deptno=e.deptno
group by d.dname;
```

HUBBLE

查询视图结果:

```
select * from dname_c;
```

```
    dname     | count
-------------+--------
  OPERATIONS |      1
  SALES      |      6
  BOSS       |      1
  ACCOUNTING |      8
  RESEARCH   |      9
```

5.2.3 分页查询

要一次遍历一个表的一页结果 (也称为分页)，有两个选项，建议只使用其中一个:

- 键集分页 (快速，推荐)
- LIMIT/OFFSET分页 (慢，不推荐)

5.2.3.1 键集分页

Keyset分页 (也称为 seek 方法) 用于从表中快速获取记录的子集。它通过限制使用WHERE和LIMIT子句组合返回的记录集来做到这一点。要获取下一页，请检查WHERE子句中列的值与上一页结果中返回的最后一行的值。

键集分页查询的一般模式是:

```
SELECT * FROM test AS OF SYSTEM TIME ${time}
  WHERE key > ${value}
  ORDER BY key
  LIMIT ${amount}
```

这比使用LIMIT更快，因为键集分页查询OFFSET不是对值进行全表扫描，而是为每次迭代查看固定大小的记录集。如果子句中用于实现分页的键是索引的且唯一，则可以快速完成此操作。

5.2.3.2 举例

建表并插入数据:

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );

insert into cust_info values('14435550','王吉','12022519960321531X',15122511874,
    ↪ '天津武清','抵押');
insert into cust_info values('14435551','张贺','431256197306265320',15534343555,
    ↪ '山西临汾','质押');
```

```
insert into cust_info values('14435552','刘明','371452199303034312',18967756743,
    ↪ '陕西延安','信用');
insert into cust_info values('14435553','李华','52112119860621421X',15833355455,
    ↪ '湖北武汉','抵押');
insert into cust_info values('14435554','郑青','213456199102275341',13054546567,
    ↪ '江西南昌','质押');
```

查询示例：

```
select * from cust_info limit 3;
```

```
   cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
      ↪ cust_type
-----------+-----------+---------------------+--------------+--------------+-------------
   ↪
  14435550 | 王吉       | 12022519960321531X |  15122511874 | 天津武清      | 抵押
  14435551 | 张贺       | 431256197306265320 |  15534343555 | 山西临汾      | 质押
  14435552 | 刘明       | 371452199303034312 |  18967756743 | 陕西延安      | 信用
```

```
select * from cust_info limit 3 offset 3;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+---------------------+--------------+--------------+-------------
   ↪
  14435553 | 李华       | 52112119860621421X |  15833355455 | 湖北武汉      | 抵押
  14435554 | 郑青       | 213456199102275341 |  13054546567 | 江西南昌      | 质押
```

由此可见，先执行数据的offset取消，在执行limit限制。

5.2.3.3 性能比较

> 在查询大量相同数据时候，使用 LIMIT/OFFSET 进行分页的查询慢，所以不推荐使用，具体可以结合 explain 使用执行计划进行查看。

5.2.4 临时表

Hubble 支持会话范围的临时表 (也称为临时表)。与持久表不同，临时表只能从创建它们的会话中访问，并且在会话结束时被删除。

要创建临时表，请将TEMP/TEMPORARY添加到CREATE TABLE或者CREATE TABLE AS语句。

Hubble 还支持临时视图和临时序列。

5.2.4.1 说明

- 临时表会在会话结束时自动删除。
- 只能从创建临时表的会话中访问临时表。
- 临时表在同一会话中的事务中持续存在。

- 临时表可以引用持久表，但持久表不能引用临时表。
- 临时表不能转换为持久表。
- 对于 PostgreSQL 兼容性，Hubble 只支持会话范围的临时表，不支持 PostgreSQL 中用于定义事务范围临时表的子句`ON COMMIT DELETE ROWS`和`ON COMMIT DROP`。

默认情况下，每 30 分钟 Hubble 会清理所有未绑定到活动会话的临时对象。您可以使用`sql.temp_object_cleaner.cleanup_interval`更改清理作业的运行频率。

5.2.4.2 临时模式

临时表不是 public 架构的一部分。相反，当您为会话创建第一个临时表时，Hubble 会为数据库当前会话中的所有临时表、临时视图和临时序列生成一个临时模式。Hubble 会为数据库当前会话中的所有临时表默认`pg_temp_<id>`。

> 因为 SHOW TABLES 语句默认为 public 模式，所以 SHOW TABLES 在不指定模式的情况下使用将不会返回任何临时表。

5.2.4.3 示例

要使用临时表，需要先设置变量`experimental_enable_temp_tables=on`：

```
set experimental_enable_temp_tables=on;
```

5.2.4.3.1 创建临时表

```
create temp table users (
        cust_id UUID,
        city STRING,
        cust_name STRING,
        cust_address STRING,
        CONSTRAINT "primary" PRIMARY KEY (cust_id ASC)
);
```

可以使用`SHOW CREATE`查看临时表：

```
show create table users;
```

```
 table_name |                        create_statement
-------------+-------------------------------------------------------------------
  ↪
  users      | CREATE TEMP TABLE pg_temp_1661498695578411398_8589934593.users (
             |     cust_id UUID NOT NULL,
             |     city STRING NULL,
             |     cust_name STRING NULL,
             |     cust_address STRING NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (cust_id ASC),
             |     FAMILY "primary" (cust_id, city, cust_name, cust_address)
             | )
```

5.2.4.3.2 创建一个引用另一个临时表的临时表

要创建另一个引用的临时表users

```
create temp table trans (
        id UUID NOT NULL,
        city STRING NOT NULL,
        type STRING,
        owner_id UUID,
        creation_time TIMESTAMP,
        CONSTRAINT "primary" PRIMARY KEY ( id ASC),
        CONSTRAINT fk_city_ref_users FOREIGN KEY ( owner_id) REFERENCES users(
            ↪ cust_id)
);
```

5.2.4.3.3 显示会话中的所有临时表

要显示会话临时模式中的所有临时表，请使用SHOW TABLES FROM pg_temp:

```
show tables from  pg_temp;
```

```
               schema_name                | table_name | type  | owner |
                  ↪ estimated_row_count | locality
------------------------------------------+------------+-------+-------+-----------------
   ↪
  pg_temp_1661498695578411398_8589934593 | trans      | table | root  |
     ↪                   0 | NULL
  pg_temp_1661498695578411398_8589934593 | users      | table | root  |
     ↪                   0 | NULL
```

还可以在SHOW语句中使用临时模式的全名(例如,show tables from pg_temp_1661498695578411398_8589934593 ↪)。

5.2.4.3.4 显示临时表 information_schema

虽然临时表不包含在public架构中，但临时表的元数据包含在information_schema和pg_catalog架构中。

例如，该information_schema.tables表包含有关所有数据库中所有模式中的所有表的信息，包括临时表:

```
select * from information_schema.tables where table_schema='
   ↪ pg_temp_1661498695578411398_8589934593';
```

```
  table_catalog |              table_schema              | table_name |
     ↪ table_type    | is_insertable_into | version
----------------+----------------------------------------+------------+-----------------+-
   ↪
  hdb           | pg_temp_1661498695578411398_8589934593 | users      | LOCAL
     ↪ TEMPORARY | YES                |       2
  hdb           | pg_temp_1661498695578411398_8589934593 | trans      | LOCAL
     ↪ TEMPORARY | YES                |       2
```

5.2.4.3.5 取消会话

如果您结束会话，所有临时表都将丢失。

```
show session_id;
```

```
             session_id
------------------------------------
  170ed442b8b9e1860000000200000001
```

```
cancel session '170ed442b8b9e1860000000200000001';
```

```
ERROR: driver: bad connection
warning: connection lost!
opening new connection: all session settings will be lost
```

```
show create table users;
```

```
ERROR: relation "trans" does not exist
SQLSTATE: 42P01
```

5.2.5 截止系统时间查询

您可以使用此子句读取历史数据并通过减少事务冲突来提高性能。

5.2.5.1 用于 AS OF SYSTEM TIME 减少与长时间运行的查询的冲突

如果您有可以容忍稍微过时的读取的长时间运行的查询（例如执行全表扫描的分析查询），请考虑使用该AS OF SYSTEM TIME子句。使用它，您的查询将返回过去某个不同点出现的数据，并且不会与其他并发事务发生冲突，从而提高应用程序的性能。

但是，由于AS OF SYSTEM TIME返回历史数据，您的读取可能会过时。

5.2.5.2 概要

该AS OF SYSTEM TIME子句在多个 SQL 上下文中受支持，包括但不限于：

- 在SELECT从句中，在子句FROM后。
- 在BACKUP...TO子句的参数之后。
- 在RESTORE..FROM子句的参数之后。
- 在BEGIN关键字之后。
- 在SET TRANSACTION关键字之后。

5.2.5.3 示例

5.2.5.3.1 选择查询历史数据

这个例子代表了数据库的当前数据：

```
select empno,ename from emp
where deptno=50;
```

```
  empno | ename
--------+---------
    101 | ALLEN
   7580 | WALLER
```

相反，我们可以检索 2022 年 08 月 3 日 12:45 UTC 的值：

```
select empno,ename from emp
 AS OF SYSTEM TIME '2022-08-03 12:45:00'
where deptno=50;
```

```
  empno | ename
--------+---------
   7580 | WALLER
```

5.2.5.3.2 使用不同的时间戳格式

常见形式

```
select * from test AS OF SYSTEM TIME '2022-08-01 08:00:00'
```

其他形式的查询

```
select * from test AS OF SYSTEM TIME 1451635200000000000
```

```
select * from test AS OF SYSTEM TIME '-4h'
```

```
select * from test AS OF SYSTEM TIME INTERVAL '-4h'
```

5.2.5.3.3 从多个表中选择

在单个FROM子句中选择多个表时，AS OF SYSTEM TIME子句必须出现在最后并应用于整个SELECT子句。

例如：

```
select * from a, b, c AS OF SYSTEM TIME '-4h';
```

```
select * from a JOIN b ON a.x = b.y AS OF SYSTEM TIME '-4h';
```

```
select * from (select * from a), (select * from b) AS OF SYSTEM TIME '-4h';
```

5.2.5.3.4 AS OF SYSTEM TIME 在子查询中使用

为了便于从更简单的查询中组合更大的查询，Hubble 允许AS OF SYSTEM TIME在以下条件下进行子查询：

- 顶级查询还指定AS OF SYSTEM TIME。
- 所有AS OF SYSTEM TIME子句都指定相同的时间戳。

例如：

```
select * from (select * from t AS OF SYSTEM TIME '-6h') t
           JOIN u ON t.x = u.y
           AS OF SYSTEM TIME '-6h'  -- same timestamp as above - OK.
     WHERE t.x < 100;
```

5.2.5.3.5 AS OF SYSTEM TIME 在交易中使用

您可以使用该SET语句使用过去指定时间的数据库内容执行事务

```
BEGIN;
```

从2019-04-09 18:02:52截止到现在的数据

```
SET TRANSACTION AS OF SYSTEM TIME '2019-04-09 18:02:52.0+00:00';
```

```
select * from a;
```

```
select * from b;
```

```
COMMIT;
```

5.2.5.3.6 用于 AS OF SYSTEM TIME 恢复最近丢失的数据

建表并插入数据

```
create table b (id int PRIMARY KEY);
```

```
insert into b VALUES (10), (20);
```

```
SELECT now();
```

```
               now
---------------------------------
  2022-08-29 16:44:00.662129+08
```

```
drop table b;
```

```
select * from b AS OF SYSTEM TIME '2022-08-29 16:44:00.662129+08';
```

```
 id
------
  10
  20
```

如果执行以下语句就会显示表不存在

```
select * from b ;
```

HUBBLE

```
ERROR: relation "b" does not exist
SQLSTATE: 42P01
```

> 一旦发生垃圾回收，AS OF SYSTEM TIME 将无法再恢复丢失的数据。对于更长期的恢复解决方案，请考虑对集群进行完整备份或增量备份。

5.2.6 选择数据行

准备工作

在阅读本页之前，请执行以下操作：

- 创建 Hubble 无服务器集群或启动本地集群。
- 安装驱动程序或 ORM 框架。
- 连接到数据库。
- 插入您现在要对其运行查询的数据。

5.2.6.1 简单选择

5.2.6.1.1 sql 语言

```sql
select empno,ename from emp;
```

5.2.6.1.2 go 语言

```go
// 'db' is an open database connection

rows, err := db.Query("select empno,ename from emp")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
fmt.Println("Initial balances:")
for rows.Next() {
    var id, balance int
    if err := rows.Scan(&id, &balance); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d %d\n", id, balance)
}
```

5.2.6.1.3 JAVA 语言

```java

try (Connection connection = ds.getConnection()) {
    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery("select empno,ename from emp");
```

```
    while (rs.next()) {
        int id = rs.getInt(1);
        int bal = rs.getInt(2);
        System.out.printf("ID: %10s\nBalance: %5s\n", id, bal);
    }
    rs.close();

} catch (SQLException e) {
    System.out.printf("sql state = [%s]\ncause = [%s]\nmessage = [%s]\n",
                      e.getSQLState(), e.getCause(), e.getMessage());
}
```

5.2.6.1.4 PYTHON 语言

```
with conn.cursor() as cur:
    cur.execute("select empno,ename from emp")
    rows = cur.fetchall()
    for row in rows:
        print([str(cell) for cell in row])
```

5.2.6.2 排序

要对查询结果进行排序，请使用ORDER BY子句。

例如:

```
select * from emp order by empno;
```

5.2.6.3 限制查询

要限制查询的结果，请使用 LIMIT 子句。

例如:

```
select * from emp order limit 5;
```

有关参考文档和更多示例，请参阅LIMIT/OFFSET语法页面。

5.2.6.4 join

具有双向连接的选择查询的语法如下所示

```
SELECT
    e.ename, b.dname
FROM
   emp AS e
    JOIN
    dept AS d
    ON
    e.deptno = d.deptno
```

HUBBLE

```
WHERE
    e.empno is not null
ORDER BY
    e.empno
LIMIT
    5;
```

5.2.7 键集分页

对结果进行分页

要一次迭代一页结果 (也称为分页)，有多种选择：

- LIMIT/OFFSET分页 (慢，不推荐)
- 键集分页 (快速，推荐)
- 游标

5.2.7.1 键集分页

键集分页 (也称为'查找方法') 用于快速从表中获取记录子集。它通过使用WHERE子句的组合返回的记录集来实现此目的。

键集分页查询的一般模式是：

```
SELECT * FROM t
  WHERE key > ${value}
  ORDER BY key
  LIMIT ${amount}
```

5.2.7.2 示例

假设使用客户表的数据集，可以将其加载到 Hubble 中

要使用键集分页获取第一页结果，请运行以下语句

```
select * from cust_info
  where cust_no >0
  order by cust_no
  limit 5;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  1         | 王吉      | 12022519960321531X |  15122511874 | 天津武清     | 抵
     ↪ 押
  2         | 张贺      | 431256197306265320 |  15534343555 | 山西临汾     | 质
     ↪ 押
  3         | 刘明      | 371452199303034312 |  18967756743 | 陕西延安     | 信
     ↪ 用
  4         | 李华      | 52112119860621421X |  15833355455 | 湖北武汉     | 抵
     ↪ 押
```

```
  5         | 郑青        | 213456199102275341 |  13054546567 | 江西南昌       | 质
      ↪ 押
(5 rows)
```

要获取第二页结果，请运行：

```
select * from cust_info
  where cust_no >5
  order by cust_no
  limit 5;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
      ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
    ↪
  6         | 王三        | 12047419660327643X |  15854557437 | 天津北辰       | 抵
      ↪ 押
  7         | 刘一        | 234256195606263452 |  13894858958 | 天津滨海       | 质
      ↪ 押
  8         | 杨泰        | 541452199303034312 |  17548587733 | 河南洛阳       | 信
      ↪ 用
  9         | 李青        | 24112119450621342X |  15634879747 | 湖北红安       | 抵
      ↪ 押
  10        | 钱森        | 32434519900221534X |  13934745774 | 辽宁沈阳       | 质
      ↪ 押
(5 rows)
```

5.3 写入数据

5.3.1 删除数据

此页面包含使用 SQL 语句从 Hubble 中删除数据行的说明。

5.3.1.1 准备工作

在阅读本页之前，请执行以下操作：

- 创建 Hubble 无服务器集群或启动本地集群。
- 安装驱动程序或 ORM 框架。
- 连接到数据库。
- 创建数据库架构。
- 插入要删除的数据。

在本页的示例中，我们使用通过命令导入的示例数据。

5.3.1.2 使用 DELETE

要删除表中的行，请使用带有`WHERE`子句的`DELETE`语句，该子句过滤标识要删除的行的列。

5.3.1.2.1 SQL 语法

在 SQL 中，DELETE语句一般采用以下形式：

```
DELETE FROM {table} WHERE {filter_column} {comparison_operator} {filter_value}
```

参数说明：

- {table}是包含要删除的行的表。
- {filter_column}是要过滤的列。
- {comparison_operator}是一个比较运算符，解析为TRUE或FALSE（例如，=）。
- {filter_value}是过滤器的匹配值。

5.3.1.2.2 最佳实践

以下是删除行时要遵循的一些最佳做法：

- 限制DELETE您执行的语句数量。使用单个语句删除多行比执行多个DELETE语句删除单个行更有效。
- 始终在查询中指定WHERE子句。DELETE如果没有WHERE指定子句，hubble 将删除指定表中的所有行。
- 要删除表中的所有行，请使用TRUNCATE语句而不是DELETE语句。
- 要删除大量行，请使用批量删除循环。
- 从应用程序执行DELETE语句时，请确保将 SQL 执行函数包装在重试循环中，以处理可能在争用下发生的事务错误。
- 查看下面的性能注意事项。

5.3.1.2.3 示例 1

删除在非唯一列上过滤的行

假设要删除一定范围编号的历史数据，empno要删除表中介于两个值 (1-1000) 之间的所有行。

sql 语言

```
delete from emp where  empno BETWEEN 1 AND 1000;
```

go 语言

```
// 'db' is an open database connection

tsOne := 1
tsTwo := 1000

if _, err := db.Exec("delete from emp where  empno BETWEEN $1 AND $2", tsOne,
    ↪ tsTwo); err != nil {
  return err
}
return nil
```

JAVA 语言

```
// ds is an org.postgresql.ds.PGSimpleDataSource

int tsOne = 1;
int tsTwo = 1000;
```

```
try (Connection connection = ds.getConnection()) {
    PreparedStatement p = connection.prepareStatement("delete from emp where
        ↪ empno BETWEEN ? AND ?");
    p.setInt(1, tsOne);
    p.setInt(2, tsTwo);
    p.executeUpdate();

} catch (SQLException e) {
    System.out.printf("sql state = [%s]\ncause = [%s]\nmessage = [%s]\n", e.
        ↪ getSQLState(), e.getCause(),
            e.getMessage());
}
```

PYTHON 语言

```
# conn is a psycopg2 connection

tsOne = 1
tsTwo = 1000

with conn.cursor() as cur:
    cur.execute(
        "delete from emp where  empno BETWEEN %s AND %s", (tsOne, tsTwo))
```

5.3.1.2.4 示例 2

删除在唯一列上过滤的行

假设要删除特定编号的历史数据。

sql 语言

```
delete from emp where  empno in (1001,7369,3256);
```

go 语言

```
// 'db' is an open database connection

codeOne := 1001
codeTwo := 7369
codeThree := 3256

if _, err := db.Exec("delete from emp where  empno in ($1, $2, $3)", codeOne,
    ↪ codeTwo, codeThree); err != nil {
  return err
}
return nil
```

JAVA 语言

```
// ds is an org.postgresql.ds.PGSimpleDataSource

int codeOne = 1001;
int codeTwo = 7369;
int codeThree = 3256;

try (Connection connection = ds.getConnection()) {
    PreparedStatement p = connection.prepareStatement("delete from emp where
        ↪ empno in (?, ?, ?)");
    p.setInt(1, codeOne);
    p.setInt(2, codeTwo);
    p.setInt(3, codeThree);
    p.executeUpdate();

} catch (SQLException e) {
    System.out.printf("sql state = [%s]\ncause = [%s]\nmessage = [%s]\n", e.
        ↪ getSQLState(), e.getCause(),
            e.getMessage());
}
```

PYTHON 语言

```
# conn is a psycopg2 connection

codeOne = 1001
codeTwo = 7369
codeThree = 3256

with conn.cursor() as cur:
    cur.execute("delete from emp where  empno in (%s, %s, %s)", (codeOne,
        ↪ codeTwo, codeThree)),
```

5.3.2 插入数据

INSERT页面包含使用 SQL 语句使用各种编程语言将数据写入 Hubble 的说明。

准备工作

在阅读本页之前，请执行以下操作：

- 创建 Hubble 无服务器集群或启动本地集群。
- 安装驱动程序或 ORM 框架。
- 连接到数据库。
- 插入要对其运行查询的数据。

5.3.2.1 插入行

插入多行时，单个多行插入语句比多个单行语句快。

5.3.2.1.1 sql

```sql
create table  emp (id INT PRIMARY KEY, aml INT);
insert into  emp (id, aml) VALUES (1, 100), (2, 200);
```

5.3.2.1.2 go 语言

```go
// 'db' is an open database connection


if _, err := db.Exec(
    "insert into  emp (id, aml) VALUES (1, 100), (2, 200)"); err != nil {
    log.Fatal(err)
}
```

5.3.2.1.3 JAVA

```java
try (Connection connection = ds.getConnection()) {
    connection.setAutoCommit(false);
    PreparedStatement pstmt = connection.prepareStatement("insert into  emp (id,
        ↪  aml) VALUES (? , ?)");

    pstmt.setInt(1, 100);
    pstmt.setInt(2, 200);
    pstmt.addBatch();

    pstmt.executeBatch();
    connection.commit();
} catch (SQLException e) {
    System.out.printf("sql state = [%s]\ncause = [%s]\nmessage = [%s]\n",
                      e.getSQLState(), e.getCause(), e.getMessage());
}
```

5.3.2.1.4 python

```python
with conn.cursor() as cur:
    cur.execute('insert into  emp (id, aml) VALUES (1, 100), (2, 200)')

conn.commit()
```

5.3.2.2 批量插入

如果您需要快速将大量数据导入 Hubble 集群，请使用`IMPORT`语句而不是从应用程序`INSERT`。它会更快，因为它完全绕过 SQL 层并使用低级命令直接写入数据存储。

6 参考

6.1 优化

6.1.1 开发使用规范

6.1.1.1 开发设计

6.1.1.1.1 命名规范

Hubble 数据库数据库命名规范可参考传统关系型 postgresql 数据库。

6.1.1.1.2 处理特定的错误码 `40001`

在客户端接受到服务端返回的异常错误，并且错误错误码为`40001`时，这意味着这个操作影响到的数据与其他事务相冲突了，并在事务管理器内部尝试重试失败后，最后返回客户端。我们建议对此错误码进行特殊处理，即客户端接受`40001`错误码后，主动进行五次以内的重试。

6.1.1.1.3 多节点进行写入

Hubble 数据库为去中心话架构，确保了每个节点服务都可等价的对外提供服务。我们建议上游单个主题日志同步数据流，可以指定到固定的节点中。这样能一定程度的减少数据更新过程中的网络开销和事务冲突。不必担心可能出现的单点故障情况，因为 Hubble 数据库在协议层，很好的兼容了 postgersql，可直接使用 postgresql-jdbc 的高可用特性，配置多个连接目标服务。在出现单点因网络等问题无法访问的情况下，自动按照顺序依次尝试连接访问目标服务节点。当然多个不同的数据源可以由开发人员指定分散到等价的服务节点，避免单点的性能瓶颈，有效使用整个集群的资源。

6.1.1.1.4 查询可适用任何节点

Hubble 数据库为去中心话架构，针对于只有查询类的服务接口，可以通过 HA Proxy、F5 等来实现负载均衡。这样查询类的服务不需要关心实际访问是哪个具体服务节点，完全避免故障切换的问题。同时面对突然增长的连接数，整个集群也能有非常良好的吞吐表现。

6.1.1.1.5 尽可能确保表有主键

设计良好的多列组合主键，可以产生比`uuid`或自增主键更好的性能，这需要在前期模式设计工作与业务数据的理解。我们建议一些单调递增的字段要位于主键的第一个列之后。这能带来如下收益：

- 主键具有足够的随机性，可以将表的数据和查询负载相对均匀的分布在集群中，从而避免出现热点。所谓足够的随机性，是指主键的前缀应该相对均匀地分布在其域，而这域与服务节点数量相对一致是最优的状态。
- 一个有实际业务含义的列在主键中，往往对查询速度也有帮助。

6.1.1.1.6 索引设计

在不使用主键查询的场景中，合理的使用索引，可以大幅度的提升查询效率。同时索引的设计是需要权衡的，在带来查询速度极大提升的同时，也可能会略微影响列更新的操作，因为更新操作必须对表和索引都进行写操作。

我们建议为所有常见的查询创建索引，为了能设计出权衡后最优的索引，符合业务场景的使用，我们可以遵守如下两个设计准则：

- 索引包含的列，尽可能满足所有谓词条件
- 在索引中存有需要查询展现的列

在设计索引时，重要的是考虑查询条件是哪些列以及他们的顺序，以下是一些具体的建议：

- 需要满足索引前缀的要求，如果当前有一个（A,B,C）索引，查询条件为（A）或（A,B）都是可以通过索引进行查询的，但（B）或（B，C）是无非是通过索引查询的。这意味着尽可能的避免创建单列索引，增加索引的通用性，即可在多个场景下复用。
- 谓词条件中出现的列，需要注意等于和包含条件（= ，in），必须出现在范围（<，>）条件之前。否则无法通过索引查。
- 索引中可以存储查用列，来避免回表的情况。命中率高的查询，是否回表对查询速度，将产生非常大的影响。

6.1.1.1.7 尽量避免大表后建索引的情况

Hubble 数据库具备在线建索引的特性，且在建索引过程中，不会出现锁表的情况。但我们需要注意一点，Hubble 数据库在完成创建索引前，需要做索引与数据一致性校验的必要动作，这个行为对于超大数据量的表，无法避免的是一个大 IO 的任务。如果无法为超大数据量表创建索引，提供合理的运维窗口，请给这个索引创建任务留出较大的时间窗口，耐心等待即可，尤其在非 SSD 存储介质基础上。当然几亿数据的表，创建索引还是可以保证效率的，这里的大表，是指 10 亿、20 亿上的表。

6.1.1.1.8 连接数的合理规划

Hubble 数据库创建多个活动连接，可以有效的使用可用的数据库资源。对于创建连接需要经过身份验证的过程，这个过程是 cpu 和内存密集型的，客户端等待数据库验证连接的过程中还增加了延迟。因此合理的控制连接数是必要的工作。我们建议连接数的量参考集群 core 数量进行衡量，计算公式是 core 数据的两倍。

当然 HA proxy 模式也对连接数的规划会产生影响，如较高维度分析型的并发任务，实际最优连接数会比计算公式得出的结果要低。如果集群使用 ssd 硬盘，能一定程度提升连接数的上限。

如应用类程序使用连接池，除了设置最大连接池大小外，还必须设置空闲连接池大小。我们的建议是将空间连接池大小设置为最大连接池相等。这个设置可能会占用更多的应用程序服务器内存，但在并发较高的情况中，可以避免创建新连接带来的性能损耗。

6.1.1.1.9 建表复制数据 `CREATE TABLE AS SELECT`

当执行`CREATE TABLE AS SELECT`语句进行建表并且进行数据复制的时候，将会导致新创建的表中丢失**主键分区和外键**等约束。影响数据检索查询效率。强烈推荐先执行`create table t2 (like t1 INCLUDING ALL EXCLUDING CONSTRAINTS)`复制表结构和约束，再执行`insert into T2 select * from T1`方式导入数据。

参考示例如下：

```
CREATE TABLE t1 (id INT PRIMARY KEY, name INT NOT NULL, INDEX(name));
CREATE TABLE t2 (LIKE t1 INCLUDING ALL EXCLUDING CONSTRAINTS);
insert into T2 select * from T1;
```

6.1.1.2 SQL 规范建议

6.1.1.2.1 使用 UUID 生成唯一主键

建表设计过程中，无法使用业务字段作为主键，我们建议使用 UUID 来作为主键字段。这能提供更好的数据分布情况。

6.1.1.2.2 插入同时返回数据

在插入数据后，需要其更新行的值，可使用`returning`语法，此语法可减少一次查询的操作。这对于使用系统自动生成的`UUID`主键非常友好。

```
insert into t_table(id,name,age) values(default,'willy',18) returning id;
```

6.1.1.2.3 批量写入速度优化

- 表初始化数据使用`import`方式

  `import`导入数据的方式，是将表临时下线后，写底层数据文件的机制实现的。这意味避免了客户端与服务端的网络、sql 解析等不必要的开销。同时`import`大数据量方式，能很好的解决大数据量导入场景中最难处理的数据一致性问题。

- 将小批量数据在一个语句中完成插入

  如 10 行，100 行，1000 行均可理解为是小批量的数据插入。在面对可预期的数据插入行为场景中（即数据源为消息中间件的场景，这个情况下不会有回滚），采用小批量的的导入方式，可以有效的提升插入效率，且提升效率至少在 10 倍之上。

6.1.1.2.4 使用多行的 DML

对于`insert`，`update`，`delete`语句，一个多行 DML 比多个单行 DML 执行更快。如果业务逻辑上允许，我们建议尽可能使用多行 DML，而不是多个单行 DML。

6.1.1.2.5 执行计划分析

在 sql 优化优化过程中，建议使用`explain`查看 sql 的执行计划，这个过程是非常有必要的，也是我们了解 sql 真实的执行过程的唯一方式。为了避免 sql 出现性能问题，我们需要判断以下几个重点：

- 是否有全表扫描的情况，即执行计划中是否有出现`spans ALL`的关键字
- 索引查询的方式与预期是否一致
- 多表`join`的次序，是否与预期一致

6.1.1.2.6 大数据量的分析如何提升性能

我们将大数据量的查询定义为命中数据多（命中率高）的查询，同时查询涉及多表关联。

针对与大查询的优化可以从如下方面入手：

- 通过合理的索引设计，减少表扫描的性能开销
- 使用合理的 join 方式，尽可能避免计算过程中对内存的占用
- 开启副本读的方式，使用分布式数据库多副本的特性，合理利用集群副本资源进行计算 `AS OF SYSTEM TIME experimental_follower_read_timestamp()`
- 在确保当前业务逻辑不变更的前提下，SQL 的实现方式是否有合理的优化空间
- 高维度的查询是否可做降维处理，先汇聚中层维度的计算结果

6.1.1.2.7 列族使用

在创建表时，默认所有字段都存储在一个列族中，这样对大部分的场景综合考量都是性能最优的。然而对于一些频繁更新的情况下，用户可以自行将频繁更新的字段，单独设计一个列簇。因为在更新过程中，单独规划列族的字段，是不需要影响其他不更新字段的列族空间的。

```
CREATE TABLE t_t2 (
    fundid INT8 NOT NULL,
    trans_amount decimal(18,2) NULL,
    data BYTES NULL,
    CONSTRAINT "primary" PRIMARY KEY (fundid ASC),
    FAMILY f1 (fundid, trans_amount),
    FAMILY f2 (data)
);
```

6.1.1.2.8 优雅的清空表

当需要清空一个表时候，建议使用truncate语句：其实际执行的方式是drop掉当前表后，重建一张新表。这相比delete在各方面都更好，因为delete需要在执行过程中生成事务。

6.1.1.2.9 避免使用 CTAS 与 SFU（select for update）

需要在生产环境中尽可能的避免create table as select，insert into table select，update table ↪ select这类 sql。

这类 sql 在执行过程中，由于可能存在大数量的查询，在查询过程中，同时存在数据被更新的并发事务，数据库会生成一个大事务级别的意向锁，这是为了避免数据在大查询未完成阶段被其他事务修改数据。往往这样的操作，在数据库流量较大的情况下，会对在线业务形成危害。

6.1.1.2.10 合理的使用视图

虚拟视图往往可以使 sql 实现变得更为简洁。

6.1.1.2.11 删除大量数据

为了删除大量的行，我们建议分批次迭代执行，直到删除所有不需要的行，当然这是在通过索引条件删除的前提下。

首先我们对删除大量数据的定义是指一万行数据以上，面对这类情况，建议拆分为小批量删除迭代进行，一直到需要被删除的数据全部删除完成。

当然为了快速定位需要删除的数据，删除条件也有必要能符合索引的规划。

6.1.1.2.12 删除对业务没有帮助的索引

索引可以有效的提升查询效率，但索引也是有代价的，会在每次写入时增加额外的开销。在大部分情况下，我们都需要进行权衡。在长期使用的过程中，如因为业务调整等原因，出现不需要使用的索引，我们建议将其删除。

6.1.1.3 部署规范

HUBBLE

6.1.1.3.1 交换分区

在进行 Hubble 集群部署的时候，要对交换分区进行关闭，因为 Hubble 数据库属于对 IO 敏感型基础架构，swapping 会将主内存交换到磁盘，从而对性能造成负面影响。建议永久性关闭交换分区，更多操作请参考部署目录下安装前服务配置。

6.1.2 索引

6.1.2.1 定义

索引是为了加速对表中数据行的检索而创建的一种分散的存储结构。索引是针对表而建立的，它是由数据页面以外的索引页面组成的，每个索引页面中的行都会含有逻辑指针，以便加速检索物理数据。类似的像书的目录，前面有标题，后面有页数，便于方便快速的查找内容，而不是全量搜索。

6.1.2.2 索引工作原理

创建一个索引时，Hubble 对你指定的列进行索引，它创建了列的备份，有了索引 SQL 可以便捷得使用索引来过滤值，而不需要全表逐行数据进行扫描，极大地提高了检索范围，时间上大幅度提高。

6.1.2.3 索引的应用

Hubble 每个表都会自动创建一个`primary`索引，它对主键进行索引，如果没有主键，则为每个行唯一的`rowid`进行索引，我们建议建表时候始终创建一个主键，而不要使用`rowid`，因为创建的主键将提供更好的查询性能。主键索引有助于过滤表的主键，可以通过以下方式对索引进行管理维护：

- 在`CREATE TABLE`建表时的`INDEX`子句中使用，也就是说在建表时候指定索引，Hubble 还会自动为具有`Unique`约束的列创建二级索引。
- 对于已存在的表，使用`CREATE INDEX`语法进行建表。
- 使用`ALTER TABLE`添加唯一约束，它会为约束列自动创建索引。
- 在`WHERE`子句中用等号运算符（`=`、`IN`）过滤的列应在索引中排在第一位，并且在使用不等号运算符引用的列之前。
- 删除索引的语句`drop index index_name`。

6.1.2.4 优缺点

优点：

- 大大加快数据的检索速度
- 加速表和表之间的连接
- 在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间

缺点：

- 索引需要占物理空间
- 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度

6.1.2.5 索引列

在设计索引时，重要的是要考虑索引哪些列以及列出它们的顺序。以下是一些指南，可帮助您做出最佳选择：每个表的主键（我们建议始终定义主键）会自动建立索引。它创建的索引不能更改，也不能在创建表后更改其主键，因此在建表时，确定如何创建主键索引是非常重要的。索引对查询很有帮助，即使仅过滤其列的前缀。例如，如果您创建列的索引`(A, B, C)`，则查询过滤有`(A)`或`(A, B)`的列都可以使用该索引。但是，未过滤带有`(A)`列的查询将不会从索引中得到性能提升的体现。索引在一张表中可以选用多个，但在实际应用中每张表的索引不会超过 3 个。

6.1.2.6 索引示例

6.1.2.6.1 示例 1(主键索引)

```
 create table  custinfo(
    cust_no   varchar(30) not null,
    cust_name varchar(30) ,
    cust_card_no  varchar(18),
    cust_phoneno  DECIMAL(15),
    primary key(cust_no)
);
```

可以通过语句查看索引信息

```
show index from custinfo;
```

```
  table_name | index_name | non_unique | seq_in_index | column_name | direction
     ↪ | storing | implicit
+------------+------------+------------+--------------+-------------+-----------+---------
   ↪
  custinfo   | primary    |   false    |            1 | cust_no     | ASC
     ↪ |  false  |  false
```

6.1.2.6.2 示例 2(非主键索引)

```
create table  custinfo_bak(
    cust_no   varchar(30) not null,
    cust_name varchar(30) ,
    cust_card_no  varchar(18),
    cust_phoneno  DECIMAL(15)

);
```

```
create index custbak_index on custinfo_bak (cust_card_no);
```

6.1.2.6.3 示例 3(联合索引)

> 联合索引特别强调顺序性，联合索引大致类似上面的 B+ 树结构，所以当索引的维护其实是以第一个字段来优先排序的，如果查询条件里没有第一个字段就没法通过索引比较来定位数据 (like 等造成的索引失效除外)。

> 只要列中包含有 NULL 值都将不会被包含在索引中，联合索引中只要有一列含有 NULL 值，那么这一列对于此联合索引就是无效的，所以我们在数据库设计时尽可能不要让字段的默认值为 NULL。

```
 create table  school_student(
    stu_no     varchar(30) not null,
```

HUBBLE

```
    stu_name    varchar(30) ,
    stu_class   varchar(18),
    stu_order   DECIMAL(15)

);
```

```
create index ss_index on school_student (stu_no,stu_name,stu_class);
```

```
> select * from school_student where stu_no='1' and stu_name='zhangsan'; --走索
  ↪ 引
> select * from school_student where stu_no='1' and stu_class='1';       --走索
  ↪ 引
> select * from school_student where stu_no='1' and stu_name='zhangsan' and
  ↪ stu_class='1'; --走索引
> select * from school_student where stu_name='zhangsan' and stu_class='1'; --不
  ↪ 走索引
> select * from school_student where stu_name='zhangsan'  ;--不走索引
> select * from school_student where stu_class='zhangsan' ;--不走索引
```

综上：只有包含第一列 (stu_no) 的时候才会走索引.

6.1.2.6.4 示例 4(二级索引的应用)

o_ol_cnt通过STORING存储在索引中，则不再需要索引联接index join。

```
CREATE TABLE order_t (
    o_id integer NOT NULL,
    o_d_id integer NOT NULL,
    o_w_id integer NOT NULL,
    o_c_id integer not NULL,
    o_ol_cnt integer not NULL,
    o_all_local integer not NULL,
    PRIMARY KEY (o_w_id ASC, o_d_id ASC, o_id DESC),
    UNIQUE INDEX order_idx (o_w_id ASC, o_d_id ASC, o_c_id ASC, o_id DESC)
);
```

```
explain select o_ol_cnt from order_t where o_d_id > 5 and o_w_id > 5 and o_c_id
  ↪ >2000;
```

```
       tree        |    field    |            description
+------------------+-------------+-----------------------------------+
                   | distributed | true
                   | vectorized  | false
  render           |             |
       index-join  |             |
                   | table       | order_t@primary
            scan   |             |
```

```
                    | table        | order_t@order_idx
                    | spans        | /6/6/2001-
                    | filter       | (o_d_id > 5) AND (o_c_id > 2000)
```

```
CREATE TABLE order_bak (
    o_id integer NOT NULL,
    o_d_id integer NOT NULL,
    o_w_id integer NOT NULL,
    o_c_id integer not NULL,
    o_ol_cnt integer not NULL,
    o_all_local integer not NULL,
    PRIMARY KEY (o_w_id ASC, o_d_id ASC, o_id DESC),
    UNIQUE INDEX order_idx (o_w_id ASC, o_d_id ASC, o_c_id ASC, o_id DESC)
       ↪ STORING (o_ol_cnt) --区分点
);
```

```
explain select o_ol_cnt from order_bak where o_d_id > 5 and o_w_id > 5 and
   ↪ o_c_id >2000;
```

```
    tree     |     field     |            description
+------------+---------------+-----------------------------------+
             | distributed   | true
             | vectorized    | false
  render     |               |
       scan  |               |
             | table         | order_bak@order_idx
             | spans         | /6/6/2001-
             | filter        | (o_d_id > 5) AND (o_c_id > 2000)
```

通过比较可看出：数据的查询仅需要从二级索引中读取，因此效率更高。

6.1.3 向量化引擎

Hubble 支持面向列的查询。大部分的数据库执行查询计划，采用逐行获取表数据的方式。

面向行的执行模式可以在联机交易（OLTP）场景中保证很好的性能，而在分析性业务场景（OLAP）中的表现并不好。Hubble 的向量化执行引擎，通过对特定于类型的列数据，使用特性的组件进行批处理查询的方式，极大地改进了面向行的执行的性能。

6.1.3.1 配置方式

默认配置情况下，在面对确定使用内存计算的查询任务，并满足可支持的数据类型时，将会使用向量化执行引擎。

可使用当前session对其进行配置，具体参数名称为vectorize，参数详情如下：

参数名称	详情
auto	内存中执行的查询时，尽可能的使用向量化执行引擎，而不需要将中间结果溢出到磁盘。

参数名称	详情
experimental_on off	所有查询都使用向量化执行引擎，不建议在生产环境使用此配置，避免出现内存问题。 对于所有查询，都不需用向量化执行引擎。

向量化执行的效率随着处理的行数的增加而提高。如查询的是一张非常小表时，那面向行的执行方式效率会远比向量化执行要快。默认配置下，如果表中数据少于 1000 行，将不会使用向量化执行引擎。设置属性名为vectorize_row_count_threshold

6.1.3.2 支持的数据类型

- STRING
- UUID
- BYTES
- DATE
- FLOAT
- INT
- BOOL
- TIMESTAMP

6.1.3.3 示例

设置向量化策略：

```
show vectorize;

vectorize
+---------+
  auto
(1 row)
```

```
set vectorize = 'auto'
```

设置向量化开启阈值：

```
show vectorize_row_count_threshold;

vectorize_row_count_threshold
+-----------------------------+
  1000
(1 row)
```

```
set vectorize_row_count_threshold=10000;
```

6.1.4 语句性能优化

6.1.4.1 概述

为了获得良好的性能，需要通过几个镜头来查看您是如何访问数据库的：

- SQL 语句性能：这是性能问题的最常见原因，表名应该从哪里着手。

- 架构设计：根据 SQL 架构和工作负载的数据访问模式，可能需要进行更改以避免创建事务争用或热点。
- 集群拓扑：作为一个分布式系统，Hubble 要求您权衡延迟与弹性，这需要根据需要选择正确的集群拓扑。

6.1.4.2 sql 语句执行原则

要获得良好的 SQL 语句性能，请遵循以下规则：

- 1. 扫描尽可能少的行。如果您的应用程序扫描的行数超过了给定语句所需的行数，则将难以扩展。
- 2. 使用正确的索引。WHERE语句应该在子句中的索引列上使用。您希望避免全表扫描对性能的影响。
- 3. 使用正确的连接类型。根据您查询的表的相对大小，连接的类型可能很重要。

6.1.4.3 使用 explain 进行语句优化

6.1.4.3.1 全表扫描

- 查询缓慢的最常见原因是全表扫描和不正确使用索引。当基于不在主键或任何二级索引中的列从大表中检索少量行时，通常会获得较差的性能：

```
SELECT * FROM users WHERE name = 'Cheyenne Smith';
```

```
                   id                   |    city     |      name       |
                      ↪ address            | credit_card
----------------------------------------+-----------+----------------+----------------------
   ↪
  326d3a95-fe03-4600-8000-00000003c1d0 | boston     | Cheyenne Smith | 84252
     ↪ Bradley Coves Suite 38 | 2369363335
  c6b45c93-7ecd-4800-8000-0000000ecdfd | amsterdam | Cheyenne Smith | 29149 Jane
     ↪  Lake              | 6072991876
  5edbdde1-c806-4400-8000-00000007114a | seattle    | Cheyenne Smith | 90016
     ↪ Anthony Groves          | 3618456173
  60edde95-4c2d-4000-8000-0000000738c7 | seattle    | Cheyenne Smith | 70310
     ↪ Knight Roads Suite 36   | 9909070365
  00024e8e-d94c-4710-8000-00000000002c | new york   | Cheyenne Smith | 8550
     ↪ Kelsey Flats             | 4374468739
  0c777227-64c8-4780-8000-00000000edc8 | new york   | Cheyenne Smith | 47925 Cox
     ↪ Ways               | 7070681549
(1 row)

Time: 981ms total (execution 981ms / network 0ms)
```

- 要了解此查询为何表现不佳，使用EXPLAIN：

```
EXPLAIN SELECT * FROM users WHERE name = 'Cheyenne Smith';
```

```
                                        info
------------------------------------------------------------------------------------------
   ↪
  distribution: full
  vectorized: true
```

```
  • filter
    estimated row count: 3
    filter: name = 'Cheyenne Smith'

      • scan
        estimated row count: 1,259,634 (100% of the table; stats collected 3
            ↪ minutes ago)
        table: users@primary
        spans: FULL SCAN
```

- 在列上没有二级索引的情况下，Hubble 会扫描表的每一行，users 按主键 (city/id) 排序，直到找到具有正确 name 值的行。

情况一：二级索引过滤

```
CREATE INDEX on users (name);
```

查询将更快的返回

```
SELECT * FROM users WHERE name = 'Cheyenne Smith';
```

```
                   id                   |   city    |      name      |
                      ↪ address          | credit_card
----------------------------------------+-----------+----------------+-------------------------
   ↪
  c6b45c93-7ecd-4800-8000-0000000ecdfd | amsterdam | Cheyenne Smith | 29149 Jane
     ↪  Lake              | 6072991876
  326d3a95-fe03-4600-8000-00000003c1d0 | boston    | Cheyenne Smith | 84252
     ↪ Bradley Coves Suite 38 | 2369363335
  00024e8e-d94c-4710-8000-00000000002c | new york  | Cheyenne Smith | 8550
     ↪ Kelsey Flats           | 4374468739
  0c777227-64c8-4780-8000-00000000edc8 | new york  | Cheyenne Smith | 47925 Cox
     ↪ Ways               | 7070681549
  5edbdde1-c806-4400-8000-00000007114a | seattle   | Cheyenne Smith | 90016
     ↪ Anthony Groves         | 3618456173
  60edde95-4c2d-4000-8000-0000000738c7 | seattle   | Cheyenne Smith | 70310
     ↪ Knight Roads Suite 36  | 9909070365
(6 rows)


Time: 24ms total (execution 3ms / network 0ms)
```

- 这表明 Hubble 从二级索引 (`users@users_name_idx`) 开始。因为是按排序的 name，所以查询可以直接跳转到相关值 (`/'Cheyenne Smith' - /'Cheyenne Smith'`)。但是，查询需要返回不在二级索引中的值，因此 Hubble 抓取与该值一起存储的主键 (city/id)name(主键始终与二级索引中的条目一起存储)，跳转到主索引中的该值，然后返回整行。

情况二：按存储列的二级索引进行过滤

当有一个按特定列过滤但检索表总列的子集的查询时，可以通过将这些附加列存储在二级索引中来提高性能，以防止查询也需要扫描主索引。例如：假设您经常检索用户的姓名和信用卡号：

```
SELECT name,credit_card FROM users WHERE name = 'Cheyenne Smith';
```

```
    name       | credit_card
-----------------+--------------
  Cheyenne Smith | 6072991876
  Cheyenne Smith | 2369363335
  Cheyenne Smith | 4374468739
  Cheyenne Smith | 7070681549
  Cheyenne Smith | 3618456173
  Cheyenne Smith | 9909070365
(1 row)

Time: 14ms total (execution 4ms / network 0ms)
```

- 在当前二级索引开启的情况 name 下，Hubble 仍然需要扫描一级索引来获取信用卡号：

```
EXPLAIN SELECT name, credit_card FROM users WHERE name = 'Cheyenne Smith';
```

```
                                        info
-------------------------------------------------------------------------------------
  ↪
  distribution: local
  vectorized: true

  • index join
    estimated row count: 3
    table: users@primary

      • scan
        estimated row count: 3 (<0.01% of the table; stats collected 2 minutes
          ↪ ago)
        table: users@users_name_idx
        spans: [/'Cheyenne Smith' - /'Cheyenne Smith']
```

- 在上删除并重新创建索引 name，这次将 credit_card 值存储在索引中：

```
DROP INDEX users_name_idx;

CREATE INDEX ON users (name) STORING (credit_card);
```

- 现在 credit_card 值存储在索引中 name，Hubble 只需要扫描该索引：

```
EXPLAIN SELECT name, credit_card FROM users WHERE name = 'Cheyenne Smith';
```

```
                                        info
-------------------------------------------------------------------------------------
  ↪
```

HUBBLE

```
  distribution: local
  vectorized: true

    scan
    estimated row count: 3 (<0.01% of the table; stats collected 27 seconds ago)
    table: users@users_name_idx
    spans: [/'Cheyenne Smith' - /'Cheyenne Smith']
```

- 这会带来更快的性能：

```
SELECT name, credit_card FROM users WHERE name = 'Cheyenne Smith';
```

```
     name       | credit_card
-----------------+--------------
  Cheyenne Smith | 6072991876
  Cheyenne Smith | 2369363335
  Cheyenne Smith | 4374468739
  Cheyenne Smith | 7070681549
  Cheyenne Smith | 3618456173
  Cheyenne Smith | 9909070365
(6 rows)

Time: 2ms total (execution 2ms / network 0ms)
```

6.1.4.4 逻辑优化

谓词下推

将查询语句中的过滤表达式计算尽可能下推到距离数据源最近的地方，进而显著地减少数据传输或计算的开销。

```
create table a (id int primary key, b int);
explain select * from a where b < 1;
```

Min/Max 函数消除

当一个 SQL 满足以下条件时，就会应用这个规则：

- 只有一个聚合函数，且为max或者min函数。
- 聚合函数没有相应的group by语句。

示例：

```
select max(n) from b;
```

这时max/min消除优化规则会将其重写为：

```
select max(n) from (select n from b where a is not null order by n desc limit 1)
    ↪  b;
```

这个 SQL 语句在 n 列存在索引时，能够利用索引只扫描一行数据来得到最大或者最小值，从而避免全表的扫描。

6.1.5 主键

6.1.5.1 定义

表中经常有一个列或多列的组合，其值能唯一地标识表中的每一行。这样的一列或多列称为表的主键，通过它可强制表的实体完整性。当创建或更改表时可通过定义PRIMARY KEY约束来创建主键。一个表只能有一个PRIMARY KEY约束，而且PRIMARY KEY约束中的列不能接受空值。由于PRIMARY KEY约束确保唯一数据，所以经常用来定义标识列。

6.1.5.2 目的

主键的一个目的就是确定数据的唯一性，它跟唯一约束的区别就是，唯一约束可以有一个NULL值，但是主键不能有NULL值，并且有以下的作用：

- 保证实体的完整性
- 加快数据库的操作速度
- 在表中添加新记录时，DBMS 会自动检查新记录的主键值，不允许该值与其他记录的主键值重复

6.1.5.3 特性

虽然主键不是必需的，但最好为每个表都设置一个主键，不管是单主键还是复合主键。它存在代表着表结构的完整性，表的记录必须得有唯一区分的字段，主键主要是用于其他表的外键关联，以及本记录的修改与删除。

6.1.5.4 主键选择原则

- 最少性：在可以同时选择单一主键和组合主键（即用几列的组合来标识唯一行）时，尽管采用单一主键。
- 稳定性：被定义为主键的列，其数据应该想对稳定，不需要经常进行更新，最好永远不会改变。

6.1.5.5 注意

> 建表时候必须在 create table 语句中指定主键，如果建表不指定主键的话，Hubble 数据库在建表时候自带 rowid 默认作为主键。

6.1.5.6 主键示例

6.1.5.6.1 单一主键

给一个字段建立主键，客户号 cust_no 唯一

```
create table  cust_info(
    cust_no    string primary key,
    cust_name varchar(30) not null,
    cust_card_no  varchar(18),
    cust_phoneno  DECIMAL(15)
 );
```

6.1.5.6.2 复合主键：

多个字段建立复合主键，通过身份证号和姓名联合保证数据唯一，这是由于实际数据中姓名可以重复，但姓名与身份证的组合不会重复，能确定唯一值。

```
create table  cust_info_bak(
    cust_name varchar(30) not null,
    cust_card_no  varchar(18),
    cust_phoneno  DECIMAL(15),
    primary key(cust_card_no,cust_name)
);
```

6.1.5.6.3 更改主键

> 主键可以进行更改，但是在被更改的主键字段，必须在 create table 时候有 not null 进行约定，如果选用新增的字段作为主键，必须强调增加的字段语句要有 not null，否则会导致主键更改失败。

```
CREATE TABLE cust_user (
  cust_id STRING PRIMARY KEY,
  cust_no varchar(20) ,
  cust_email STRING ,
  cust_name STRING,
  cust_addr STRING,
  cust_en STRING
);
```

- 此时 cust_id 为主键

```
show columns from cust_user;
```

```
  column_name |  data_type  | is_nullable | column_default |
     ↪ generation_expression |  indices  | is_hidden
--------------+-------------+-------------+----------------+-----------------------+------
   ↪
  cust_id     | STRING      |    false    | NULL           |
     ↪                      | {primary} |   false
  cust_no     | VARCHAR(20) |    false    | NULL           |
     ↪                      | {}        |   false
  cust_email  | STRING      |    true     | NULL           |
     ↪                      | {}        |   false
  cust_name   | STRING      |    true     | NULL           |
     ↪                      | {}        |   false
  cust_addr   | STRING      |    true     | NULL           |
     ↪                      | {}        |   false
  cust_en     | STRING      |    true     | NULL           |
     ↪                      | {}        |   false
  cust_tel    | STRING      |    false    | NULL           |
     ↪                      | {}        |   false
```

- 如果想用 cust_no 作为主键，必须做非空设置

```
ALTER TABLE cust_user ALTER COLUMN cust_no SET NOT NULL;
```

```
ALTER TABLE cust_user ALTER PRIMARY KEY USING COLUMNS (cust_no);
```

- 增加列，并用此列作为新的主键

```
ALTER TABLE cust_user ADD COLUMN cust_tel string NOT NULL;
```

```
ALTER TABLE cust_user ALTER PRIMARY KEY USING COLUMNS (cust_tel);
```

- 此时，主键已经更改成功，查看如下

```
show columns from cust_user;
```

```
  column_name |  data_type   | is_nullable | column_default |
     ↪ generation_expression |              indices              | is_hidden
--------------+-------------+-------------+----------------+-----------------------+------
   ↪
  cust_id     | STRING      |    false    | NULL           |
     ↪                      | {cust_user_cust_id_key}         |   false
  cust_no     | VARCHAR(20) |    false    | NULL           |
     ↪                      | {}                              |   false
  cust_email  | STRING      |    true     | NULL           |
     ↪                      | {}                              |   false
  cust_name   | STRING      |    true     | NULL           |
     ↪                      | {}                              |   false
  cust_addr   | STRING      |    true     | NULL           |
     ↪                      | {}                              |   false
  cust_en     | STRING      |    true     | NULL           |
     ↪                      | {}                              |   false
  cust_tel    | STRING      |    false    | NULL           |
     ↪                      | {cust_user_cust_id_key,primary} |   false
```

6.1.6 使用 EXPLAIN 进行 SQL 调优

本文档介绍了 SQL 查询缓慢的常见原因，并描述了如何使用EXPLAIN来解决针对查询中的问题。

6.1.6.1 问题类型

6.1.6.1.1 全表扫描

查询缓慢的最常见原因是SELECT语句的全表扫描和索引的不正确使用。

数据准备

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
```

```
    cust_address   varchar(30),
    cust_type      varchar(10)
);
```

根据不在主键或任何索引中的列从大型表中检索少量行时，性能通常会很差：

```
SELECT * FROM cust_info WHERE cust_name = '刘明';
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435551 | 刘明       | 431256197306265320 |  15534343555 | 山西临汾      | 质押
  14435552 | 刘明       | 371452199303034312 |  18967756743 | 陕西延安      | 信用
  14435553 | 刘明       | 52112119860621421X |  15833355455 | 湖北武汉      | 抵押
(3 rows)

Time: 442ms total (execution 1ms / network 1ms)
```

要了解此查询执行不佳的原因，使用EXPLAIN：

```
explain SELECT * FROM cust_info WHERE cust_name = '刘明';
```

```
                                        info
-------------------------------------------------------------------------------
   ↪
  distribution: full
  vectorized: false

  • filter
    estimated row count: 1
    filter: cust_name = e'\U00005218\U0000660E'

      • scan
        estimated row count: 5 (100% of the table; stats collected 8 minutes ago
           ↪ )
        table: cust_info@primary
        spans: FULL SCAN
```

spans: FULL SCAN展示，如果列cust_name上没有索引，hubble 会扫描表的每一行，cust_info按主键排序，直到找到具有正确cust_name值的行。

解决方法：按照索引过滤

要加快此查询，在cust_name上添加二级索引：

```
create index on cust_info (cust_name);
```

查询现在将返回得更快：

```
SELECT * FROM cust_info WHERE cust_name = '刘明';
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435551 | 刘明       | 431256197306265320 |  15534343555 | 山西临汾      | 质押
  14435552 | 刘明       | 371452199303034312 |  18967756743 | 陕西延安      | 信用
  14435553 | 刘明       | 52112119860621421X |  15833355455 | 湖北武汉      | 抵押
(3 rows)

Time: 6ms total (execution 1ms / network 1ms)
```

要了解性能提高的原因，使用EXPLAIN查看新的查询计划：

```
explain SELECT * FROM cust_info WHERE cust_name = '刘明';
```

```
                                        info
-------------------------------------------------------------------------------------
   ↪
  distribution: local
  vectorized: false

  • index join
    estimated row count: 1
    table: cust_info@primary

      • scan
        estimated row count: 1 (20% of the table; stats collected 18 minutes ago
           ↪ )
        table: cust_info@cust_info_cust_name_idx
        spans: [/e'\U00005218\U0000660E' - /e'\U00005218\U0000660E']
```

有索引后，查询可以直接跳转到相关值，然后返回整行。

解决方案：通过存储额外列的二级索引进行过滤

例如，经常检索用户名和卡号

```
SELECT cust_name,cust_card_no FROM cust_info WHERE cust_name = '刘明';
```

```
 cust_name |     cust_card_no
-----------+----------------------
  刘明       | 431256197306265320
  刘明       | 371452199303034312
  刘明       | 52112119860621421X

(3 rows)
```

```
Time: 4ms total (execution 2ms / network 0ms)
```

有了当前的二级索引cust_name，hubble 仍然需要扫描一级索引来获取卡号

删除并重新创建索引cust_name，这次将cust_card_no值存储在索引中

```
drop index cust_info_cust_name_idx;
```

```
create index on cust_info (cust_name) STORING (cust_card_no);
```

现在cust_card_no值存储在cust_name上的索引中，hubble 只需要扫描该索引：

```
explain SELECT cust_name,cust_card_no FROM cust_info WHERE cust_name = '刘明';
```

```
                                    info
-------------------------------------------------------------------------------
  distribution: local
  vectorized: false

  • scan
    estimated row count: 3 (60% of the table; stats collected 14 minutes ago)
    table: cust_info@cust_info_cust_name_idx
    spans: [/e'\U00005218\U0000660E' - /e'\U00005218\U0000660E']
```

```
SELECT cust_name,cust_card_no FROM cust_info WHERE cust_name = '刘明';
```

```
 cust_name |     cust_card_no
-----------+---------------------
  刘明      | 431256197306265320
  刘明      | 371452199303034312
  刘明      | 52112119860621421X

(3 rows)

Time: 2ms total (execution 2ms / network 0ms)
```

这样会使查询更快

6.1.6.1.2 低效连接

基于成本的优化器无法执行查找连接，因为该查询没有可用的表主键前缀，因此必须读取整个表并搜索匹配项，从而导致速度慢询问：

```
EXPLAIN SELECT * FROM emp JOIN dept on  emp.deptno = dept.deptno limit 1;
```

```
                                    info
-------------------------------------------------------------------------------------
  ↪
  distribution: full
  vectorized: false
```

```
• limit
  estimated row count: 1
  count: 1

    • hash join
      estimated row count: 27
      equality: (deptno) = (deptno)

        • scan
          estimated row count: 27 (100% of the table; stats collected 3
        ↪ minutes ago)
          table: emp@primary
          spans: FULL SCAN

        • scan
          estimated row count: 5 (100% of the table; stats collected 2 minutes
            ↪ ago)
          table: dept@primary
          spans: FULL SCAN
```

解决方法：提供主键允许lookup join

为了加快查询速度，可以提供主键以允许基于成本的优化器执行查找连接而不是散列连接：

```
EXPLAIN SELECT * FROM emp JOIN dept on  emp.deptno = dept.deptno and emp.empno=
  ↪ dept.empno limit 1;
```

```
                                        info
-----------------------------------------------------------------------------------
  ↪
  distribution: full
  vectorized: false

  • limit
    estimated row count: 0
    count: 1

    • lookup join
      estimated row count: 0
      table: emp@primary
      equality: (empno) = (empno)
      equality cols are key
      pred: deptno = deptno

        • scan
          estimated row count: 5 (100% of the table; stats collected 2 minutes
            ↪ ago)
```

```
            table: dept@primary
            spans: FULL SCAN
```

6.1.6.1.3 连接来自不同表的数据

在连接来自不同表的数据时，二级索引也很重要。

要计算在给定区间的员工数。

```
SELECT count(DISTINCT emp.empno) FROM emp INNER JOIN dept ON emp.loc = dept.loc
    ↪ WHERE dept.deptno BETWEEN 20 AND 40;
```

```
  count
---------
     17
(1 row)

Time: 90ms total (execution 90ms / network 0ms)
```

要了解发生了什么，使用EXPLAIN查看查询计划：

```
explain SELECT count(DISTINCT emp.empno) FROM emp INNER JOIN dept ON emp.loc =
    ↪ dept.loc WHERE dept.deptno BETWEEN 20 AND 40;
```

```
                                        info
-----------------------------------------------------------------------------------------------
    ↪
  distribution: full
  vectorized: false

  • group (scalar)
    estimated row count: 1

      • distinct
        estimated row count: 7
        distinct on: empno

          • hash join
            estimated row count: 9
            equality: (loc) = (loc)

              • filter
                estimated row count: 9
                filter: (deptno >= 20) AND (deptno <= 40)

                  • scan
                    estimated row count: 27 (100% of the table; stats collected
              ↪ 52 minutes ago)
                    table: emp@primary
```

```
                    spans: FULL SCAN

              • filter
                estimated row count: 2
                filter: (deptno >= 20) AND (deptno <= 40)

                  • scan
                    estimated row count: 5 (100% of the table; stats collected
                      ↪ 51 minutes ago)
                    table: dept@primary
                    spans: FULL SCAN
(31 rows)

Time: 2ms total (execution 1ms / network 0ms)
```

hubble 首先进行全表扫描dept以获取指定行，然后进行另一次全表扫描emp以查找匹配的行并计算计数。

解决方案: 在存储连接键的条件上创建二级索引

```
CREATE INDEX ON dept(deptno) STORING (loc);
```

```
SELECT count(DISTINCT emp.empno) FROM emp INNER JOIN dept ON emp.loc = dept.loc
  ↪ WHERE dept.deptno BETWEEN 20 AND 40;
```

```
  count
---------
     17
(1 row)

Time: 7ms total (execution 90ms / network 0ms)
```

可见，添加二级索引减少了查询时间。

要了解性能提高的原因，请再次使用EXPLAIN查看新的查询计划：

```
explain SELECT count(DISTINCT emp.empno) FROM emp INNER JOIN dept ON emp.loc =
  ↪ dept.loc WHERE dept.deptno BETWEEN 20 AND 40;
```

```
                                          info
------------------------------------------------------------------------------------------
  ↪
  distribution: full
  vectorized: false

  • group (scalar)
    estimated row count: 1

      • distinct
        estimated row count: 0
```

```
        distinct on: empno

          • hash join
            estimated row count: 0
            equality: (loc) = (loc)

              • scan
                estimated row count: 27 (100% of the table; stats collected 10
              ↪ minutes ago)
                table: emp@primary
                spans: FULL SCAN

              • scan
                estimated row count: 3 (60% of the table; stats collected 8
                   ↪ minutes ago)
                table: dept@dept_deptno_idx
                spans: [/20 - /40]
```

hubble 现在开始使用dept@dept_deptno_idx索引来检索数据，而无需扫描整个表。

6.1.7 SQL 性能最佳实践

此页面提供了优化 Hubble 查询性能的最佳实践。

6.1.7.1 DML 最佳实践

6.1.7.1.1 使用多行语句而不是多个单行语句

对于INSERT、UPSERT和DELETE语句，单个多行语句比多个单行语句更快。尽可能对DML查询使用多行语句，而不是多个单行语句。

当需要修改多行数据时，推荐使用单个 SQL 多行数据的语句：

```
insert into t values (10, 'aa'), (20, 'bb'), (30, 'cc');

delete from t where id in (10, 20, 30);
```

不推荐使用多个 SQL 单行数据的语句：

```
insert into t values (10, 'aa');
insert into t values (20, 'bb');
insert into t values (30, 'cc');

delete from t where id = 10;
delete from t where id = 20;
delete from t where id = 30;
```

6.1.7.1.2 避免不必要的查询

如非必要，不要总是用select *返回所有列的数据，下面查询是低效的：

```
select * from students where name = 'Mike';
```

应该仅查询需要的列信息，例如：

```
select sex,class from students where  name = 'Mike';
```

6.1.7.1.3 在没有二级索引的表上使用 UPSERT 代替 INSERT ON CONFLICT

当插入或者更新表的所有列时，表没有二级索引，建议使用UPSERT语句而不是等效INSERT ON CONFLICT语句。虽然INSERT ON CONFLICT总是执行读取以确定必要的写入，但UPSERT语句只写入而不读取，从而使其速度更快。UPSERT对于有二级索引的表，和INSERT ON CONFLICT之间没有性能差异。

6.1.7.2 批量插入最佳实践

6.1.7.2.1 使用多行 INSERT 语句向现有表中批量插入

要将数据批量插入到现有表中，在一个多行INSERT语句中批量处理多行。通过监控不同批量大小 (10 行、100 行、1000 行) 的数据性能，通过实验确定应用程序的最佳批量大小。

插入多行示例

```
insert into cust_info (id, city, name, address, credit_card) values
    ('8', 'beijing', 'LUCY', '400 Broad St', '1111111111'),
    ('9', 'new york', 'MIKE', '214 43rd St',  '2222222222'),
    ('10', 'shanghai', 'daiwei', '04 41rd St', '3333333333');
```

6.1.7.2.2 使用 IMPORT 代替 INSERT 批量插入到新表中

要将数据批量插入到全新的表中，该IMPORT语句的性能优于INSERT。

6.1.7.3 批量删除最佳实践

6.1.7.3.1 使用 TRUNCATE 而不是 DELETE 删除表中的所有行

该TRUNCATE语句通过删除表并重新创建同名的新表来删除表中的所有行。这比使用DELETE执行得更好，后者执行多个事务以删除所有行。

当需要删除一个表的所有数据时，推荐使用TRUNCATE语句：

```
truncate table t;
```

或者

```
truncate  t;
```

不推荐使用DELETE全表数据：

```
delete from t;
```

6.1.7.3.2 批量删除过期数据

支持表行的生存时间 (TTL) 过期，也称为行级 TTL。行级 TTL 是一种机制，通过该机制，表中的行被视为'已过期'，并且一旦这些行的存储时间超过指定的过期时间，就可以自动删除这些行。

6.1.7.3.3 使用批量删除删除大量行

要删除大量行，我们建议反复删除成批的行，直到删除所有不需要的行。

6.1.7.4 列族

6.1.7.4.1 分配列族

列族是表中的一组列，作为单个键值对存储在底层键值存储中。创建表时，所有列都存储为单个列族。在大多数情况下，这种默认方法可确保高效的键值存储和性能。然而，当频繁更新的列与很少更新的列组合在一起时，很少更新的列仍然会在每次更新时被重写。特别是当很少更新的列很大时，将它们分配给不同的列族会提高性能。

6.1.7.5 唯一 ID 最佳实践

在 Hubble 这样的分布式数据库中生成唯一 ID 的最佳实践与单节点数据库有很大不同。为单节点数据库生成唯一 ID 的传统方法包括：

- 使用`SERIAL`列的伪类型来生成随机的唯一 ID。这可能会导致性能瓶颈，因为在时间上彼此靠近生成的 ID 具有相似的值，并且在表的存储中物理上彼此靠近。
- 通过使用带有往返的事务来生成单调递增的 ID。这具有非常高的性能成本，因为它使所有`INSERT`事务都等待轮到他们插入下一个 ID。在某些情况下，使用更改数据捕获 (CDC) 可以帮助避免严格的 ID 排序要求。如果想避免严格的 ID 排序要求，则可以使用下面概述的一种更高性能的 ID 策略。

上述方法可能会在 Hubble 中为读取和写入创建热点。为避免此问题，建议采用以下方法:

方法	优点	缺点
使用多列主键	如果做得好，可能最快	复杂，需要预先设计和测试以确保性能
用于`UUID`生成唯一 ID	很好地分散负载，简单的选择	可能会保留一些性能，需要其他列在查询中有用
与从句一起使用`INSERT RETURNING`	易于查询，熟悉的设计	性能比其他选项慢

6.1.7.5.1 使用多列主键

设计良好的多列主键可以产生比`UUID`主键更好的性能，但它需要更多的前期架构设计工作。要获得最佳性能，请确保任何单调递增的字段都位于主键的第一列之后。如果操作正确，这样的复合主键应该会产生：

- 一个单调递增的列，它是主键的一部分，这在查询中也很有用。
- 主键中有足够的随机性以在集群中相对均匀地分布表数据查询负载，这将避免热点。足够的随机性指主键的前缀应该相对均匀地分布在其域中。

示例数据

```
create table cust_info (
    name string,
    res_timestamp TIMESTAMP,
    CONSTRAINT ci_pk PRIMARY KEY(name, res_timestamp)
);
```

这将使以下查询变得高效

```
select * from cust_info
```

```
        where  name = 'mike'
      ORDER BY res_timestamp DESC
         LIMIT 5;
```

要了解原因，看一下EXPLAIN输出

```
EXPLAIN (VERBOSE)
select * from cust_info
        where  name = 'mike'
      ORDER BY res_timestamp DESC
         LIMIT 5;
```

```
                    info
-------------------------------------------
  distribution: local
  vectorized: false

  • revscan
    columns: (name, res_timestamp)
    ordering: -res_timestamp
    estimated row count: 5 (missing stats)
    table: cust_info@ci_pk
    spans: /"mike"-/"mike"/PrefixEnd
    limit: 5
```

上述查询遵循索引子句中所有列的索引最佳实践。

6.1.7.5.2 INSERT 与子句一起使用 RETURNING 以生成唯一 ID

生成随机唯一 ID

假设表架构如下：

```
CREATE TABLE test (
    ID1 INT,
    ID2 INT,
    ID3 INT DEFAULT 1,
    PRIMARY KEY (ID1,ID2)
  );
```

生成随机唯一 ID 的常用方法是使用以下语句：

```
BEGIN;

INSERT INTO test VALUES (1,1);

SELECT * FROM X WHERE ID1=1 AND ID2=1;

COMMIT;
```

性能最佳实践是使用RETURNING子句，而不是使用事物：

```
INSERT INTO test VALUES (1,1),(2,2),(3,3)
    RETURNING ID1,ID2,ID3;
```

6.1.8 函数索引

简介

函数索引就是给字段加了函数的索引，这里的函数也可以是表达式；所以也叫表达式索引。

使用场景

比如：如果某张表的数据量特别大，并且其中某个字段在当前数据库中保存大小写共存，当需要查询的时候忽略大小写，那么此时一般使用到的方法就是使用 sql 中的upper()函数，但是使这时候应用upper()函数后，sql 语句是不会走索引的，建议为该字段创建一个函数索引，从而提高查询效率。

6.1.8.1 函数索引示例

6.1.8.1.1 示例

- 建表

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_ctime     timestamp default now(),
    index(cust_card_no)
 );
```

- insert数据

```
insert into cust_info values('SJDHADJu','王吉','12022519960321531X',15122511874,
    ↪ '天津武清','2021-02-02 00:00:00');
insert into cust_info values('HJSucjUN','张贺','431256197306265320',15534343555,
    ↪ '山西临汾','2021-03-02 00:00:00');
insert into cust_info values('AHAHuicn','刘明','371452199303034312',18967756743,
    ↪ '陕西延安','2021-04-02 00:00:00');
insert into cust_info values('AusdnACJ','李华','52112119860621421X',15833355455,
    ↪ '湖北武汉','2021-05-02 00:00:00');
insert into cust_info values('CCisuWEN','郑青','213456199102275341',13054546567,
    ↪ '江西南昌','2021-06-02 00:00:00');
```

- 创建函数索引

```
create index on cust_info(upper(cust_no));
```

- 查看执行计划

```
explain
select * from cust_info
where upper(cust_no)='CCISUWEN';
```

```
                    info
---------------------------------------------
  distribution: local
  vectorized: true

  • index join
    table: cust_info@cust_info_pkey

      • scan
        missing stats
        table: cust_info@cust_info_expr_idx
        spans: [/'CCISUWEN' - /'CCISUWEN']
(10 rows)
```

6.1.8.1.2 示例 2

- 建表并建立索引

```
create table test1 (a date );
```

```
create index idx_funtochar on test1(to_char(create time));
```

- 查看执行计划

```
explain select * from test1 where to_char(create_time)='2023-08-23';
```

```
                                        info
-------------------------------------------------------------------------------
  ↪
  distribution: local
  vectorized: true

  • index join
    estimated row count: 0
    table: test1@test1_pkey

      • scan
        estimated row count: 1 (99% of the table; stats collected 46 seconds ago
          ↪ )
        table: test1@idx_funtochar
        spans: [/'2023-08-23' - /'2023-08-23']
(11 rows)
```

HUBBLE

其他场景还有不同函数，都可以按照此类此种方式去建立函数索引。

6.2 安全

6.2.1 SQL 审计日志

SQL 审计日志记录了执行的查询的详细信息。记录针对包含个人标识信息的表运行的所有查询时，该特性特别有用。提供了一个在 Hubble 中的 SQL 审计日志的示例，包括：

- 如何打开和关闭审计日志。
- 审计日志文件所在的位置。
- 审计日志文件是什么样子。

6.2.1.1 参数说明

`ALTER TABLE ... EXPERIMENTAL_AUDIT SET ...`

范围	描述
`table_name`	您要为其创建审核日志的表的名称。
`read`	将所有表读取记录到审核日志文件中。
`write`	将所有表写入记录到审计日志文件。
`off`	关闭审计日志。

6.2.1.2 操作步骤

6.2.1.2.1 步骤一：创建示例表

使用下面的语句来创建：

- 创建 users 表。
- 展示如何打开 users 表的审计日志。

```
create table  users (
    id uuid  primary key default gen_random_uuid(),
    name string not null,
    password string not null,
    address string not null,
    telephone int not null,
    email string unique not null
);
```

6.2.1.2.2 步骤二：打开 users 表的审计

我们使用`ALTER TABLE` 的 `EXPERIMENTAL_AUDIT` 子命令打开对表的审计。

```
alter table users  experimental_audit set read  write;
```

6.2.1.2.3 步骤三：找到审计日志文件用滚动日志方式打开此文件

HUBBLE

默认情况下，活动审计日志文件的前缀是 hubble-sql-audit，并存储在 Hubble 数据库的标准日志目录中。要将审计日志文件存储在特定的目录中，请将——`sql-audit-dir`标志传递给 hubble start。与其他日志文件一样，它根据——`log-file-max-size`设置进行调整。当我们查看这个示例的审计日志时，我们会正如预期的那样看到下面的行，显示了到目前为止我们所运行的每个命令。

- 根据 Hubble 启动的 service，查找`--log-config-file=log.yaml`中的这个文件 `log.yaml`具体位置
- 再根据这个`log.yaml`文件查看审计日志文件路径

例如：

```
[root@poc-hubble01 ~]# more /var/lib/hubbletp317/log.yaml
```

```
# configuration after validation:

file-defaults:
  # 日志的路径
  dir: /data/hubbledir310/logs

  max-file-size: 10MiB

  max-group-size: 100MiB

  buffered-writes: true

  filter: INFO

  format: hubbledb-v2

  redact: false

  redactable: true

  exit-on-error: true
```

滚动审计日志

```
[root@poc-hubble01 logs]# pwd
/data/hubbledir317/logs
[root@poc-hubble01 logs]# tail -f hubble317-sql-audit.log
```

6.2.1.2.4 步骤四: 给 users 表添加测试数据

先打开审计日志文件查看是否有滚动日志，再执行插入以及查询语句

```
insert into users (name, password, address, telephone, email) values (
    'scottwolf',
    'scottwolf',
    '北京海淀',
    18210301210,
    '16666666666@qq.com'
```

```
),(
    'scott',
    'scott',
    '北京朝阳',
    18210301200,
    '177777777@qq.com'
);
```

验证是否已成功添加数据:

```
root@poc-hubble01:35432/defaultdb> select * from users;
```

```
                    id                   |    name     | password   | address   |
                        ↪ telephone  |        email
-----------------------------------------+-------------+------------+-----------+-------------+-
    ↪
  5b3dba74-8894-4964-a418-83479d3aca41 | scottwolf | scottwolf | 北京海淀 |
      ↪ 18210301210 | 16666666666@qq.com
  ce24b852-1a33-4b2a-9377-9b6b9cc1a20a | scott     | scott     | 北京朝阳 |
      ↪ 18210301200 | 177777777@qq.com
(2 rows)

Time: 4ms total (execution 3ms / network 0ms)
```

6.2.1.2.5 步骤五: 验证审计日志开启

通过查看审计日志是否有滚动

```
I220616 05:52:04.516870 1228462113 8@util/log/event_log.go:32   [n1,client
    ↪ =<192.168.1.11:19420>,hostssl,user=root] 25 ={"Timestamp"
    ↪ :1655358724514188553,"EventType":"sensitive_table_access","Statement":"<
    ↪ INSERT INTO \"\".\"\".users(name, password, address, telephone, email)
    ↪ VALUES ('scottwolf', 'scottwolf', e'\\U00005317\\U00004EAC\\U00006D77\\
    ↪ U00006DC0', 18210301210, '16666666666@qq.com'), ('scott', 'scott', e'\\
    ↪ U00005317\\U00004EAC\\U0000671D\\U00009633', 18210301200, '177777777@qq.
    ↪ com')>","Tag":"INSERT","User":"root","DescriptorID":7842,"ApplicationName"
    ↪ :"$ hubble sql","ExecMode":"exec","NumRows":2,"Age":1.920044,"TxnCounter"
    ↪ :59,"AccessMode":"rw"}
I220616 05:52:29.741715 1228462113 8@util/log/event_log.go:32   [n1,client
    ↪ =<192.168.1.11:19420>,hostssl,user=root] 26 ={"Timestamp"
    ↪ :1655358749739046887,"EventType":"sensitive_table_access","Statement":"<
    ↪ SELECT * FROM \"\".\"\".users>","Tag":"SELECT","User":"root","DescriptorID
    ↪ ":7842,"ApplicationName":"$ hubble sql","ExecMode":"exec","NumRows":2,"Age
    ↪ ":1.679085,"FullTableScan":true,"TxnCounter":61,"TableName":"<defaultdb.
    ↪ public.users>","AccessMode":"r"}
```

操作 sql 语句写入到审计日志中

6.2.1.2.6 步骤六: 关闭审计日志

关闭审计日志语句:

```
root@poc-hubble01:35432/defaultdb> alter table  users experimental_audit set
    ↪ off;
ALTER TABLE

Time: 55ms total (execution 54ms / network 0ms)
```

设置完关闭审计日志，先打开审计日志文件，再执行对 users 表查询，验证是否滚动日志

查询对应表，查看审计日志文件是否有日志输出

```
[root@poc-hubble01 logs]# pwd
/data/hubbledir310/logs
[root@poc-hubble01 logs]# tail -f hubble317-sql-audit.log
```

```
root@poc-hubble01:35432/defaultdb> select count(1) from users;
```

```
  count
---------
      2
(1 row)

Time: 3ms total (execution 2ms / network 0ms)
```

操作查询语句，审计日志文件没有滚动日志。结论：users 表审计日志关闭

6.2.2 安全证书更换

本数据库允许您轮换安全证书而无需重新启动节点

6.2.2.1 何时轮换证书

在以下情况下，您可能需要轮换节点，客户端或 CA 证书:

- 节点，客户端或 CA 证书即将到期。
- 您组织的合规性政策要求定期进行证书轮换。
- 密钥已被泄露（对于节点，客户端或 CA。
- 您需要修改证书的内容，例如，添加另一个 DNS 名称或负载均衡器的 IP 地址，通过该 IP 地址可以访问节点。在这种情况下，您只需要轮换节点证书。

6.2.2.2 轮换客户证书

1. 创建一个新的客户端证书和密钥:

```
  hubble cert create-client <username>
  --certs-dir=certs \
  --ca-key=mysafedirectory/ca.key
```

1. 使用您的首选方法将新的客户端证书和密钥上载到客户端。

2. 让客户端使用新的客户端证书。

   此步骤是特定于应用程序的，可能需要重新启动客户端。

6.2.2.3 轮换节点证书

要轮换节点证书，请创建新的节点证书并密钥，然后将其重新加载到节点上。

1. 创建一个新的节点证书和密钥：

```
hubble cert create-node \
<node hostname> \
<node other hostname> \
<node yet another hostname> \
--certs-dir=certs \
--ca-key=mysafedirectory/ca.key \
--overwrite
```

由于必须在与现有证书和密钥相同的目录中创建新的证书和密钥，因此请使用`--overwrite`标志覆盖现有文件。另外，请确保指定可以到达该节点的所有地址。

1. 将节点证书和密钥上载到节点：

```
scp certs/node.crt \
certs/node.key \
<username>@<node address>:~/certs
```

1. 通过向`hubble`进程发出`SIGHUP`信号来重新加载节点证书，而无需重新启动节点：

```
pkill -SIGHUP -x hubble
```

`SIGHUP`信号必须由运行该进程的同一用户发送（例如，如果`hubble`进程在 root 用户下运行，则使用 sudo 运行）。

1. 使用管理界面中的“本地节点证书”页面来验证证书轮换是否成功：

```
https://<address of node with new certs>:8080/#/reports/certificates/local
```

滚动到节点证书详细信息，并确认“有效期至”字段显示新证书的到期时间。

6.2.2.4 轮换 CA 证书

要轮换 CA 证书，您需要创建一个新的 CA 密钥以及一个包含新 CA 证书和旧 CA 证书的组合 CA 证书，然后在节点和客户端上重新加载新的 CA 证书。一旦所有节点和客户端都具有合并的 CA 证书，就可以创建使用新 CA 证书签名的新节点和客户端证书，并将这些证书重新加载到节点和客户端上。

> Tip：为什么 Hubble 创建组合的 CA 证书
>
> 轮换 CA 证书时，重新扫描 certs 目录后，节点将具有新的 CA 证书，而客户端在重新启动时将具有新的 CA 证书。但是，在轮换节点证书和客户端证书之前，仍然使用旧的 CA 证书对节点和客户端证书进行签名。因此，节点和客户端无法使用新的 CA 证书来验证彼此的身份。
>
> 为了解决此问题，我们利用了多个 CA 证书可以同时处于活动状态这一事实。在验证另一个节点或客户端的身份时，他们可以检查上载到它们的多个 CA 证书。因此，Hubble 可以创建一个组合的 CA 证书，其中包含新的 CA 证书，后跟旧的 CA 证书，而不仅仅是在轮换 CA 证书时

仅创建新证书。轮换节点证书和客户端证书时，将使用组合的 CA 证书来验证旧的以及新的节点证书和客户端证书。

Tip：为什么要提前轮换 CA 证书?

在轮换 CA 证书后轮换节点和客户端证书时，将使用新的 CA 证书对节点和客户端证书进行签名。重新扫描节点上的 certs 目录后，节点将使用新节点和 CA 证书。但是，客户端仅在重新启动客户端后才使用新的 CA 和客户端证书。因此，尚不具有新 CA 证书的客户端将不接受由新 CA 证书签名的节点证书。为确保所有节点和客户端都拥有最新的 CA 证书，请以完全不同的时间表轮换 CA 证书；理想情况下，是在更改节点和客户端证书之前的几个月。

1. 重命名已存在的 CA 证书：

```
mv mysafedirectory/ca.key mysafedirectory/ca.old.key
```

1. 使用`--overwrite`标志覆盖旧的 CA 证书，创建一个新的 CA 证书和密钥：

```
hubble cert create-ca \
--certs-dir=certs \
--ca-key=mysafedirectory/ca.key \
--overwrite
```

- 这将生成组合的 CA 证书 ca.crt，其中包含新证书，后跟旧证书。

警告：CA 密钥永远不会由 hubble 命令自动加载，因此应在一个单独的目录中创建该目录，该目录由--ca-key 标志标识。

1. 将新的 CA 证书上载到每个节点：

```
scp certs/ca.crt <username>@<node1 address>:~/certs
```

1. 使用您的首选方法将新的 CA 证书上载到每个客户端。
2. 在每个节点上，通过向 Hubble 进程发出`SIGHUP`信号来重新加载 CA 证书，而无需重新启动该节点：

```
pkill -SIGHUP -x hubble
```

`SIGHUP`信号必须由运行该进程的同一用户发送（例如，如果 Hubble 进程在 root 用户下运行，则使用 sudo 运行）。

1. 在每个客户端上重新加载 CA 证书。此步骤是特定于应用程序的，可能需要重新启动客户端。
2. 使用管理界面中的“本地节点证书”页面来验证证书轮换是否成功：

```
https://<address of node with new certs>:8080/#/reports/certificates/local
```

应显示旧的和新的 CA 证书的详细信息。确认新 CA 证书的有效期至字段显示新证书的到期时间。

1. 一旦确定所有节点和客户端都具有新的 CA 证书，就轮换节点证书并轮换客户端证书。

6.3 SQL

6.3.1 SQL 语法和语言结构

6.3.1.1 公用表表达式

公共表表达式 (CTE)，也称为查询，在用于更大的查询上下文之前为可能复杂的子查询用WITH提供简写名称。这提高 SQL 代码的可读性。

您可以将 CTE 与SELECT子句和INSERT、DELETE、UPDATE和UPSERT数据修改语句结合使用。

6.3.1.1.1 语法图

```
WithCommonTable ::=
'WITH' 'RECURSIVE'? table_name name 'AS' ('NOT' 'MATERIALIZED')? Prepar_Stmt


Preparable_Stmt ::=
   insert_stmt
|  update_stmt
|  delete_stmt
|  upsert_stmt
|  select_stmt
```

6.3.1.1.2 参数介绍

参数	说明
table_name	用于引用随附查询或语句中的公用表表达式的名称。
name	新定义的公用表表达式中的列之一的名称。
prepar_stmt	用作公用表表达式的语句或子查询。
table_name	需要添加列的表

6.3.1.1.3 说明

语句为子查询的结果WITH x AS (y)创建临时表名，以便在上下文中重用。

示例如下：

```
with e as (select * from  emp where deptno=30)
  select e.empno,e.ename,d.loc from e left join dept as d  on e.deptno = d.
     ↪ deptno ;
```

```
   empno | ename  |   loc
--------+--------+----------
   7698 | BLAKE  | CHICAGO
   7499 | ALLEN  | CHICAGO
   7844 | TURNER | CHICAGO
   7521 | WARD   | CHICAGO
   7654 | MARTIN | CHICAGO
   7900 | JAMES  | CHICAGO
```

在以上示例中，子句定义了子查询，用WITH引用了临时名称e，并且该名称成为表名，用于后续子句的表达式中。

也可以使用单个WITH子句同时定义多个公用表表达式，用逗号分隔。后面的子查询可以按名称引用前面的子查询。例如：

```
with e as (select * from  emp where deptno=30),
res as (select e.empno,e.ename,d.loc from e left join dept as d  on e.deptno = d
    ↪ .deptno)
select * from res;
```

```
   empno | ename  |   loc
--------+--------+----------
   7698 | BLAKE  | CHICAGO
   7499 | ALLEN  | CHICAGO
   7844 | TURNER | CHICAGO
   7521 | WARD   | CHICAGO
   7654 | MARTIN | CHICAGO
   7900 | JAMES  | CHICAGO
```

在此示例中，第二个 CTE 按名称res引用了e。

6.3.1.1.4 用 WITH 嵌套子语句

可以WITH在子查询中使用子句，WITH在另一个子句中使用WITH子句。例如：

```
with  d as
    (select  *  from
         (with dn as (select  *  from dept ) select  *  from dn))
select  *  from  d;
```

```
 deptno |   dname    |  loc
---------+------------+-----------
      50 | BOSS       | BEIJING
      10 | ACCOUNTING | NEW YORK
      20 | RESEARCH   | DALLAS
      30 | SALES      | CHICAGO
      40 | OPERATIONS | BOSTON
      60 | programmer | bj
```

6.3.1.1.5 数据变更语句

包含数据变更语句的子句位于查询的上层，您就可以将数据修改语句用作公共表表达式。例如：

```
with loc as
   (insert into dept (deptno,dname,loc)
values (90,'CTO','SHANGHAI')
returning dname)
select dname from loc;
```

```
  dname
---------
  CTO
```

6.3.1.1.6 递归公用表表达式

递归公用表表达式是包含引用其自身输出的子查询的公用表表达式。

递归 CTE 定义采用以下形式：

```
WITH RECURSIVE <cte name> (<columns>) AS (
    <initial subquery>
  [UNION | UNION ALL]
    <recursive subquery>
)
<query>
```

编写递归 CTE：

- 在 CTE 定义中的运算符之后和 CTE 名称之前直接添加RECURSIVE关键字。
- 定义一个初始的非递归子查询，此子查询定义 CTE 的初始值。
- 在初始子查询之后添加UNION、UNION ALL关键字。该UNION变体对行进行重复数据删除。
- 定义一个引用它自己的输出的递归子查询。与初始子查询不同，此子查询还可以引用 CTE 名称。
- 编写一个评估 CTE 结果的父查询。

Hubble 评估递归 CTE 如下：

- 评估初始查询。其结果存储在 CTE 中的行中，并复制到临时工作表中。此工作表在递归子查询的迭代中更新。
- 在工作表的内容上迭代地评估递归子查询。每次迭代的结果都会替换工作表的内容。结果也存储到 CTE 的行中。递归子查询迭代直到没有结果返回。

举例如下：

以下递归 CTE 计算数字 0 到 10 的阶乘：

```
with RECURSIVE w (n, f) as (
    values (0, 1)
  union all
    select n+1, (n+1)*f from w where n < 10
)
select * from  w;
```

```
n  |    f
-----+----------
   0 |        1
   1 |        1
   2 |        2
   3 |        6
   4 |       24
   5 |      120
```

```
   6 |      720
   7 |     5040
   8 |    40320
   9 |   362880
  10 | 3628800
```

如果WHERE示例中未定义子句，则递归子查询将始终返回结果并无限循环，从而导致错误：

```
with RECURSIVE w (n, f) as (
    values (0, 1)
  union all
    select n+1, (n+1)*f from w
)
select * from  w;
```

```
ERROR: integer out of range
SQLSTATE: 22003
```

如果您不确定您的递归子查询是否会无限循环，您可以使用LIMIT关键字限制 CTE 的结果，避免integer out
↪ of range错误：

```
with RECURSIVE w (n, f) as (
    values (0, 1)
  union all
    select n+1, (n+1)*f from w
)
select * from  w limit 11;
```

这种做法适用于测试环境中，但不建议在生产中使用。

6.3.1.2 空值介绍

NULL是用于表示缺失值的术语。表中的NULL值是字段中显示为空白的值。NULL值的字段是没有值的字段。

本页总结了 Hubble 中值为NULL的处理方式。

使用内置客户端时，空值显示值NULL。这将它们与包含空字符串 ("") 的字符字段区分开来。

6.3.1.2.1 NULL 简单比较

NULL值和结果之间的任何简单比较。

```
create table tb(
  a int2,
  b int4,
  c int8
);
```

```
insert into tb values(1, 0, 0);
insert into tb values(2, 0, 1);
insert into tb values(3, 1, 0);
```

```
insert into tb values(4, 1, 1);
insert into tb values(5, NULL, 0);
insert into tb values(6, NULL, 1);
insert into tb values(7, NULL, NULL);
```

```
select * from tb;
```

```
+---+------+------+
| a |  b   |  c   |
+---+------+------+
| 1 |    0 |    0 |
| 2 |    0 |    1 |
| 3 |    1 |    0 |
| 4 |    1 |    1 |
| 5 | NULL |    0 |
| 6 | NULL |    1 |
| 7 | NULL | NULL |
+---+------+------+
```

```
select * from tb where b is null and c is not null;
```

```
| a |  b   | c |
+---+------+---+
| 5 | NULL | 0 |
| 6 | NULL | 1 |
+---+------+---+
```

6.3.1.2.2 NULL 和条件运算符

条件运算符 (包括IF, COALESCE) 仅根据IFNULL条件操作数的值评估某些操作数，它们的结果并不总是NULL取决于给定的操作数。

例如，COALESCE(1, NULL)也将返回1。

```
SELECT COALESCE(1, NULL);
```

```
  coalesce
------------
     1
```

6.3.1.2.3 NULL 和算术

涉及NULL值的算术运算将产生NULL结果。

```
select a, b, c, b*0, b*c, b+c from tb;
```

```
| a |  b   |  c   | b * 0 | b * c | b + c |
+---+------+------+-------+-------+-------+
| 1 |    0 |    0 |     0 |     0 |     0 |
```

```
| 2 |     0 |     1 |      0 |      0 |      1 |
| 3 |     1 |     0 |      0 |      0 |      1 |
| 4 |     1 |     1 |      0 |      1 |      2 |
| 5 | NULL  |     0 | NULL   | NULL   | NULL   |
| 6 | NULL  |     1 | NULL   | NULL   | NULL   |
| 7 | NULL  | NULL  | NULL   | NULL   | NULL   |
+---+-------+-------+--------+--------+--------+
```

6.3.1.2.4 NULL 和聚合函数

聚合函数是对一组行进行操作并返回单个值的函数。为了更容易理解结果，此处重复了示例数据。

```
select * from tb;
```

```
 a |  b    |  c
----+------+-------
  1 |     0 |     0
  2 |     0 |     1
  3 |     1 |     0
  4 |     1 |     1
  5 | NULL  |     0
  6 | NULL  |     1
  7 | NULL  | NULL
```

```
select COUNT(*), COUNT(b), SUM(b), AVG(b), MIN(b), MAX(b) from tb;
```

```
  count | count | sum | avg | min | max
--------+-------+-----+-----+-----+------
      7 |     4 |   2 | 0.5 |   0 |   1
```

事项说明：

- NULL值不包含在COUNT()列中。COUNT(*)返回 7 而COUNT(b)返回 4。
- NULL值不被视为其中的最大值MIN()或最小值MAX()。
- AVG(b)在返回时候也不考虑空值作为有效值。

6.3.1.2.5 NULL 作为不同的值

NULL值被认为与其他值不同，并包含在列中的不同值列表中。

```
select distinct c from tb;
```

```
  c
--------
     0
     1
  NULL
```

但是，计算不同值的数量不包括NULL，这与COUNT()函数一致。

```
select count(distinct c) from tb;
```

```
| count(distinct c) |
+-------------------+
|                 2 |
+-------------------+
```

6.3.1.2.6 NULL 作为其他值

在某些情况下，如果希望NULL在算术或聚合函数计算中包含值。为此，请在计算期间使用该IFNULL()函数替换一个空值。

例如，假设想要计算列的平均值，c即SUM()所有数字c除以总行数，而不管c的值是否为NULL。在这种情况下，将使用AVG(IFNULL(c, 0))，其中在计算期间用IFNULL(c, 0)将NULL值替换为0。

```
select avg(c) as m, avg(ifnull(c,0)) as n from tb;
```

```
  m   |          n
------+-------------------------
  0.5 | 0.42857142857142857143
```

6.3.1.2.7 NULL 和集合操作

NULL值被视为UNION集合操作的一部分。

```
select c from tb union select c from tb;
```

```
   c
--------
   0
   1
  NULL
```

6.3.1.2.8 NULL 排序

在对值的列进行排序包含NULL时，当进行升序时候首先进行null排序，后进行列值比较：

```
select * from tb order by c asc;
```

```
  a |   b    |   c
----+--------+-------
  7 | NULL   | NULL
  3 |      1 |      0
  5 | NULL   |      0
  1 |      0 |      0
  2 |      0 |      1
  4 |      1 |      1
  6 | NULL   |      1
```

HUBBLE

当进行降序时候先行列值比较，而后展示null

```
select * from tb order by c desc;
```

```
  a |   b    |   c
----+------+-------
  2 |     0 |     1
  4 |     1 |     1
  6 | NULL |     1
  1 |     0 |     0
  3 |     1 |     0
  5 | NULL |     0
  7 | NULL | NULL
```

6.3.1.2.9 NULL 与唯一约束

NULL值不被认为是唯一的。因此，如果一个表对一个或多个可选列具有约束，则可以在这些列中插入具有值为NULL的多行，如下例所示

```
create table tt(m int, n int unique);
```

```
insert into tt values(1, 1);
insert into tt values(3, null);
insert into tt values(5, null);
```

```
select * from tt;
```

```
  m |   n
----+-------
  1 |     1
  3 | NULL
  5 | NULL
```

6.3.1.2.10 NULL 和 CHECK 约束

计算结果为的CHECK约束表达式NULL被认为是通过，例如a < b而不用担心添加OR a IS NULL子句。当需要非空验证时，通常的NOT NULL约束可以与CHECK约束一起使用。

```
create table shops (id string primary key, b int not null check (b > 0), a int,
    ↪ check (a < b));
```

```
insert into shops (id, b) values ('nccdd4477', -5);
```

```
failed to satisfy CHECK constraint (b > 0:::INT8)
```

```
insert into shops values ('nccdffh87', 5, 13);
```

```
failed to satisfy CHECK constraint (a < b)
```

6.3.1.2.11 NULL 与其他类型的连接

非NULL值和NULL值之间的连接会产生一个NULL值。

```
select NULL || 1 as t;
```

```
   t
--------
  NULL
```

```
select null || 'qwe' as t;
```

```
   t
--------
  NULL
```

6.3.1.3 选择查询

选择查询读取和处理 Hubble 中的数据。它们比简单的SELECT子句更通用：它们可以使用集合操作对一个或多个选择子句进行分组，并且可以请求特定的排序或行限制。

可能会发生选择查询：

- 在查询的顶层，与其他 SQL 语句一样。
- 括号之间作为子查询。
- 作为将表格数据作为输入的其他语句的操作数，例如INSERT、UPSERT、CREATE TABLE AS或ALTER ... SPLIT AT。

6.3.1.3.1 条款

TABLE 条款

语法

```
TableStmt ::=
   'TABLE' tab_ref
```

子句从指定的TABLE表中读取表格数据。结果表数据的列以表的模式命名。

TABLE a相当于select * from a。

```
create table tb_copy as table tb;
```

此语句将表中的内容复制tb到新表中。但是，该TABLE子句不保留从其读取的表的架构中的索引、外键或约束和默认信息。

SELECT 条款

有关详细信息，请参阅简单查询子句。

6.3.1.3.2 设置操作

集合操作结合了来自两个选择子句的数据。它们作为其他集合操作的操作数或作为选择查询中的主要组件是有效的。

语法

```
SelectStmt ::=
   select_clause (UNION|INTERSECT|EXCEPT) (ALL|DISTINCT)? select_clause
```

集合运算符

SQL 允许您比较多个选择子句的结果。您可以将每个集合运算符视为表示布尔运算符:

- UNION=OR
- INTERSECT=AND
- EXCEPT=NOT

默认情况下，这些比较中的每一个仅显示每个值的一个副本（类似于SELECT DISTINCT）。

建表及其数据准备:

```
create table cust_infobak(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );

insert into cust_infobak values('14435551','张贺','431256197306265320'
   ↪ ,15534343555,'山西临汾','质押');
insert into cust_infobak values('14435552','刘明','371452199303034312'
   ↪ ,18967756743,'陕西延安','信用');
insert into cust_infobak values('14435553','李华','52112119860621421X'
   ↪ ,15833355455,'湖北武汉','抵押');
insert into cust_infobak values('14435554','郑青','213456199102275341'
   ↪ ,13054546567,'江西南昌','质押');
```

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );


insert into cust_info values('14435550','王吉','12022519960321531X',15122511874,
   ↪ '天津武清','抵押');
insert into cust_info values('14435551','张贺','431256197306265320',15534343555,
   ↪ '山西临汾','质押');
```

```
insert into cust_info values('14435552','刘明','371452199303034312',18967756743,
    ↪ '陕西延安','信用');
```

UNION: 合并两个查询

UNION将两个查询的结果合并为一个结果。

```
select * from cust_info
union
select * from cust_infobak;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435551 | 张贺                    | 431256197306265320 |  15534343555 | 山西临
     ↪ 汾                        | 质押
  14435552 | 刘明                    | 371452199303034312 |  18967756743 | 陕西延
     ↪ 安                        | 信用
  14435550 | 王吉                    | 12022519960321531X |  15122511874 | 天津武
     ↪ 清                        | 抵押
  14435553 | 李华                    | 52112119860621421X |  15833355455 | 湖北武
     ↪ 汉                        | 抵押
  14435554 | 郑青                    | 213456199102275341 |  13054546567 | 江西南
     ↪ 昌                        | 质押
```

要显示重复的行，您可以使用UNION ALL

```
select * from cust_info
union all
select * from cust_infobak;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435550 | 王吉                    | 12022519960321531X |  15122511874 | 天津
     ↪ 武清                        | 抵押
  14435551 | 张贺                    | 431256197306265320 |  15534343555 | 山西
     ↪ 临汾                        | 质押
  14435552 | 刘明                    | 371452199303034312 |  18967756743 | 陕西
     ↪ 延安                        | 信用
  14435551 | 张贺                    | 431256197306265320 |  15534343555 | 山西
     ↪ 临汾                        | 质押
  14435552 | 刘明                    | 371452199303034312 |  18967756743 | 陕西
     ↪ 延安                        | 信用
  14435553 | 李华                    | 52112119860621421X |  15833355455 | 湖北
     ↪ 武汉                        | 抵押
```

```
  14435554 | 郑青                    | 213456199102275341 |  13054546567 | 江西
     ↪ 南昌                      | 质押
```

INTERSECT: 检索两个查询的交集

INTERSECT仅选择两个查询操作数中都存在的值。

```
select * from cust_info
intersect
select * from cust_infobak;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+---------------------+--------------+--------------+-----------
   ↪
  14435551 | 张贺                    | 431256197306265320 |  15534343555 | 山西
     ↪ 临汾                      | 质押
  14435552 | 刘明                    | 371452199303034312 |  18967756743 | 陕西
     ↪ 延安                      | 信用
```

EXCEPT：从另一个查询中排除一个查询的结果

EXCEPT选择存在于第一个查询操作数中但不存在于第二个查询操作数中的值。

```
select cust_no from cust_info
except
select cust_no from cust_infobak;
```

```
  cust_no
------------
  14435550
```

6.3.1.3.3 排序查询

按一列对数据进行排序

```
select * from cust_info order by cust_phoneno asc;--升序排列，  asc可以省略
select * from cust_info order by cust_phoneno desc;--降序排列
```

按多列对数据进行排序

ORDER BY m,n按列对行先进行m排序，然后对列具有相同m值的行，按照n排序。

```
select * from tb order by b asc,a desc;
```

```
  a |   b    |   c
----+--------+-------
  7 | NULL   | NULL
  6 | NULL   |    1
  5 | NULL   |    0
  2 |    0   |    1
```

```
  1 |     0 |     0
  4 |     1 |     1
  3 |     1 |     0
```

6.3.1.3.4 限制行数

可以使用LIMIT减少数据的查询量。

```
select * from tb limit 3;
```

6.3.1.3.5 用于并发控制的行级锁定 SELECT FOR UPDATE

该SELECT FOR UPDATE语句用于通过控制对表的一行或多行的并发访问来对事务进行排序。

它通过锁定选择查询返回的行来工作，这样试图访问这些行的其他事务被迫等待锁定这些行的事务完成。这些其他事务根据尝试读取锁定行的值的时间有效地放入队列。

因为这种排队发生在读取操作期间，如果多个并发执行的事务尝试访问相同的数据，会阻止该选择的结果，Hubble 还可以防止可能发生的事务重试。

有关如何使用它的示例，请参阅SELECT FOR UPDATE。

6.3.1.3.6 可组合性查询

使用选择子句作为选择查询

可以将选择子句用作选择查询而无需更改。

例如，构造select * from tb是一个选择子句。它也是一个有效的选择查询，因此可以通过附加分号用作独立语句。

```
select * from tb;
```

```
 a |   b    |   c
----+------+-------
  1 |     0 |     0
  2 |     0 |     1
  3 |     1 |     0
  4 |     1 |     1
  5 | NULL |     0
  6 | NULL |     1
  7 | NULL | NULL
```

使用表表达式作为选择子句

例如，表达式TABLE emp和SELECT * FROM emp是等效的选择子句。

同样，SQL 连接表达式emp e JOIN dept d ON e.deptno= d.deptno是一个表表达式。您可以将其转换为有效的选择子句，从而成为有效的选择查询，如下所示：

```
table emp e JOIN dept d ON e.deptno= d.deptno;
```

```
select * from emp e JOIN dept d ON e.deptno= d.deptno;
```

使用选择查询作为表表达式

可以将选择查询（或选择子句）用作表表达式，方法是将其括在括号中，从而形成子查询。

要使其表表达式有效，可以将其括在括号中，如下所示：

```
select cust_name from (select * from cust_info limit 2);
```

```
select e.ename
from  emp e
join (select * from dept ) as d
  on e.deptno = d.deptno;
```

6.3.1.4 常数介绍

SQL 常量代表一个不变的值。

6.3.1.4.1 介绍

Hubble 中有五类常量：

- 字符串文字：这些定义字符串值，但它们的实际数据类型将从上下文中推断出来，例如nihao。
- 数字文字：这些定义了数值，但它们的实际数据类型将从上下文中推断出来，例如-52.7。
- 位数组文字：这些定义了数据类型为的位数组值 BIT，例如B'20020202'。
- 字节数组字面量：这些定义了数据类型为的字节数组值BYTES。例如b'\141\。
- 解释文字：这些定义具有显式类型的任意值，例如INTERVAL '5 days'。
- 命名常量：这些具有预定义类型的预定义值，例如，TRUE。

6.3.1.4.2 字符串文字

Hubble 支持以下字符串文字格式：

- 标准 SQL 字符串文字。
- 带有 C 转义序列的字符串文字。
- 美元符号引用的字符串文字。

这些格式还允许编码为 UTF-8 的任意 Unicode 字符。

在任何情况下，字符串文字的实际数据类型都是使用它出现的上下文来确定的。

标准 SQL 字符串文字

SQL 字符串文字由单引号 (') 之间的任意字符序列组成，例如'hello mike'。

要在字符串中包含单引号，请使用双单引号。例如：

```
select 'nihao' as c, 'he''s a good boy' as d;
```

```
   c    |         d
--------+------------------
 nihao  | he's a good boy
```

为了与 SQL 标准兼容，Hubble 还识别以下特殊语法：由换行符分隔的两个简单字符串文字自动连接在一起形成一个常量。例如：

```
select 'nihao'
' doctor!' as h;
```

```
       h
-----------------
  nihao doctor!
```

这种特殊语法仅在两个简单文字由换行符分隔时才有效，例如`'hello' ' world!'`不起作用，这是 SQL 标准规定的。

带有字符转义的字符串文字

Hubble 还支持包含转义序列的字符串文字，就像编程语言 C 中一样。这些文字是通过在字符串文字前加上字母来构造的 e，例如`e'nihao\nlisan!'`

支持以下转义序列：

转义序列	解释
`\a`	ASCII 代码 7（BEL）
`\b`	退格（ASCII 8）
`\t`	制表符（ASCII 9）
`\n`	换行符（ASCII 10）
`\v`	垂直制表符（ASCII 11）
`\f`	换页（ASCII 12）
`\r`	回车（ASCII 13）
`\xHH`	十六进制字节值
`\ooo`	八进制字节值
`\uXXXX`	ASCII 代码 7 (BEL)
`\a`	16 位十六进制 Unicode 字符值

美元符号引用的字符串文字

为了更容易在 SQL 代码中编写某些类型的字符串常量，Hubble 支持美元符号引用的字符串文字。这对于需要包含大量单引号 (') 或反斜杠 () 的字符串特别有用。

美元符号引用的字符串具有以下形式：`$+`（可选）标签 + `$+` 任意文本 + `$+`（可选）标签 `+$`

例如：

```
SELECT char_length($MyCoolString$

For example, here is a regular expression using backticks: $myRegex$[foo\tbar]
    ↪ $myRegex$

Finally, you can use $stand-alone dollar signs without the optional tag$.
    ↪ $MyCoolString$);
```

```
 char_length
---------------
    165
```

6.3.1.4.3 位数组文字

位数组文字由B前缀后跟一串用单引号括起来的二进制数字（位）组成。

例如：`B'1001010101'`

6.3.1.4.4 字节数组文字

Hubble 支持两种字节数组字面量格式：

- 带有 C 转义序列的字节数组文字。
- 十六进制编码的字节数组文字。

带有字符转义的字节数组文字

这使用与包含字符转义的字符串文字相同的语法，使用b前缀而不是e. 任何字符转义都被解释为字符串文字。

例如：`b'nihao,\x32Hubble'`

字节数组文字和带有字符转义的字符串文字之间的两个区别如下：

- 字节数组文字总是具有数据类型BYTES，而字符串文字的数据类型取决于上下文。
- 字节数组文字可能包含无效的 UTF-8 字节序列，而字符串文字必须始终包含有效的 UTF-8 序列。

十六进制编码的字节数组文字

这是一个 Hubble 特定的扩展，用于表示字节数组文字：分隔符x'后跟任意十六进制数字序列，然后是结束符。

例如，以下格式都等效于b'cat'：

- x'636174'

6.3.1.4.5 命名常量

Hubble 识别以下 SQL 命名常量：

- TRUE和FALSE，数据类型的两个可能值BOOL。
- NULL，表示不存在值的特殊 SQL 符号。

这NULL对于任何类型都是有效的常量：其在表达式评估期间的实际数据类型是根据上下文确定的。

6.3.1.5 标量表达式

大多数 SQL 语句可以包含从数据中计算新值的标量表达式。例如，在查询SELECT ceil(num)FROM a中，表达式ceil(num)计算列中值的四舍五入值。

标量表达式产生适合存储单个表格单元格（一行一列）中的值。它们可以与表达式和选择查询进行对比，后者产生结构化的结果。

6.3.1.5.1 常数

常量表达式表示一个不变的简单值。

6.3.1.5.2 列的引用

查询中的表达式可以通过两种方式引用当前数据源中的列：

列的名称

- 如果列的名称也是 SQL 关键字，则必须适当地引用该名称。例如：SELECT "Default" FROM conf。

- 如果名称不明确（例如，在连接多个表时），可以通过在列名前加上表名来消除歧义。例如，SELECT res.num FROM res。

列的序号位置

- 例如，SELECT @1 FROM res选择res中的第一列。

6.3.1.5.3 一元和二元运算

以一元运算符为前缀的表达式，或由二元运算符分隔的两个表达式构成一个新表达式。

有关 Hubble 运算符的完整列表，以及有关它们的优先顺序以及每个运算符的有效操作数的数据类型的详细信息，请参阅函数。

值比较

标准运算符< （小于）、> （大于）、<= （小于或等于）、>= （大于或等于）、= （等于） <>和!= （不等于）、（IS等于）和 （IS NOT不等于） ）可以应用于来自单一数据类型的任何一对值，以及来自不同数据类型的一些值对。

以下特殊规则适用：

- NULL总是比其他所有值都小，甚至是它自己。
- 要检查一个值是否是NULL，请使用IS运算符或条件表达式IFNULL(...)。

多值比较

语法：

```
<expr> <comparison> ANY  <expr>
<expr> <comparison> SOME <expr>
<expr> <comparison> ALL  <expr>
```

值比较运算符<, >, =, <=,>=,<>和 !=以及模式匹配运算符[NOT] LIKE可用于将左侧的单个值与右侧的多个值进行比较。

这是通过使用关键字ANY/SOME/ALL或组合运算符来完成的。

比较结果为真当且仅当：

- 对于ANY/SOME，左侧值的比较对于右侧的任何元素都是true。
- 对于ALL，对于右侧的每个元素，左侧值的比较都为真。

比如：

```
SELECT 1 = ANY (1, 2, 3) as bools;
```

```
 bools
---------
  true
```

```
SELECT 1 = ALL (1, 2, 3) as bools;
```

```
 bools
---------
  false
```

```
SELECT 4 = ANY ARRAY[56, 4, 10] as bools;
```

```
 bools
---------
  true
```

设置成员规则

语法：

```
<expr> IN <expr>
<expr> IN ( ... subquery ... )

<expr> NOT IN <expr>
<expr> NOT IN ( ... subquery ... )
```

当且仅当左操作数的值是计算右操作数的结果的一部分时才返回TRUE。

```
select ('a') in (select * from abc);
```

```
select  a in (1, 2, 3) from test_table;
```

字符串模式匹配

语法：

```
<expr> LIKE      <expr>

<expr> NOT LIKE  <expr>
```

将两个表达式都计算为字符串，然后测试左侧的字符串是否与右侧给出的模式匹配。

模式可以包含_匹配任何单个字符，或%匹配任何零个或多个字符的序列。

例如：

```
SELECT 'tuesday' LIKE '%day' AS a, 'tursday' LIKE 'tur_day' AS b;
```

```
  a    |  b    |
------+-------
  true| true |
```

使用 POSIX 正则表达式匹配字符串

语法：

```
<expr> ~   <expr>
<expr> ~*  <expr>
<expr> !~  <expr>
<expr> !~* <expr>
```

将两个表达式都计算为字符串，然后测试左侧的字符串是否与右侧给出的模式匹配。

带有星号的*使用不区分大小写的匹配；否则匹配区分大小写。

与LIKE模式不同，正则表达式可以匹配字符串中的任何位置，而不仅仅是开头。

例如：

```
select 'tuesday' ~ 'day' AS x, 'tuesday' ~ 't[uU][eE]sday' AS y, 'tuesday' ~* 'T
    ↪ .*y' AS z;
```

```
  x    |  y    |  z
-------+------+-------
  true| true | true
```

使用 SQL 正则表达式匹配字符串

语法：

```
<expr> SIMILAR TO      <expr>
<expr> NOT SIMILAR TO <expr>
```

将两个表达式都计算为字符串，然后测试左侧的字符串是否与右侧给出的模式匹配。

该模式使用 SQL 标准的正则表达式定义来表示。这是 SQL 中LIKE模式和 POSIX 正则表达式的混合：

- _和%分别表示任何字符或任何字符串。
- .专门匹配句点字符，不像在 POSIX 中它是通配符。
- 大多数其他 POSIX 语法照常适用。
- 该模式匹配整个字符串（如LIKE，与 POSIX 正则表达式不同）。

例如：

```
select 'tuesday' SIMILAR TO '__esday' AS x, 'tuEsday' SIMILAR TO 't[uU][eE]sday'
    ↪  AS y, 'tuesday' SIMILAR TO 'tu%y' AS z;
```

```
  x    |  y    |  z
-------+------+-------
  true| true | true
```

6.3.1.5.4 函数调用和 SQL 特殊形式

语法：

```
<name> ( <arguments...> )
```

内置函数名称后跟左括号，后跟逗号分隔的表达式列表，后跟右括号。

这会将命名函数应用于括号之间的参数。当函数的命名空间没有前缀时，名称解析规则决定调用哪个函数。

此外，还支持以下 SQL 特殊形式：

特殊形式	相当于
AT TIME ZONE	timezone()
CURRENT_CATALOG	current_catalog()
COLLATION FOR	pg_collation_for()
CURRENT_DATE	current_date()
CURRENT_ROLE	current_user()
CURRENT_SCHEMA	current_schema()

特殊形式	相当于
CURRENT_TIMESTAMP	current_timestamp()
CURRENT_USER	current_user()
EXTRACT(<part> FROM <value>)	extract("<part>", <value>)
OVERLAY(<text1> PLACING <text2> FROM <int1> FOR <int2>)	overlay(<text1>, <text2>, <int1>, <int2>)
OVERLAY(<text1> PLACING <text2> FROM <int>)	overlay(<text1>, <text2>, <int>)
SESSION_USER	current_user()
SUBSTRING(<text> FOR <int1> FROM <int2>))	substring(<text>, <int2>, <int1>)
SUBSTRING(<text> FOR <int>)	substring(<text>, 1, <int>))
SUBSTRING(<text> FROM <int1> FOR <int2>)	substring(<text>, <int1>, <int2>)
SUBSTRING(<text> FROM <int>)	substring(<text>, <int>)
TRIM(<text1> FROM <text2>)	btrim(<text2>, <text1>)
TRIM(<text2>, <text1>)	btrim(<text2>, <text1>)
TRIM(FROM <text>)	btrim(<text>)
TRIM(LEADING <text1> FROM <text2>)	ltrim(<text2>, <text1>)
TRIM(LEADING FROM <text>)	ltrim(<text>)
TRIM(TRAILING <text1> FROM <text2>)	rtrim(<text2>, <text1>)
TRIM(TRAILING FROM<text>)	rtrim(<text>)
USER	current_user()

6.3.1.5.5 下标表达式

例如，如果名称 A 引用了一个包含 10 个值的数组，A[5] 则将检索第 5 个值。第一个值的索引为 1。

如果索引小于或等于 0，或者大于数组的大小，则下标表达式的结果为NULL。

6.3.1.5.6 条件表达式

表达式可以测试条件表达式，并根据是否满足或满足哪个条件，计算一个或多个附加操作数。

这些表达式格式共享以下属性：它们的某些操作数仅在条件为真时才被评估。当操作数否则无效时，这一点尤其重要。例如，如果 a 为 0，IF(a=0, 0, x/a)则返回 0 ，否则返回 x/a 的值。

IF 表达式

语法：

```
IF ( <cond>, <expr1>, <expr2> )
```

```
select IF(tb.a=0, 0, b/a) from tb;
```

```
             if
---------------------------
                          0
                          0
  0.33333333333333333333
                       0.25
```

CASE 表达式

语法:

```
CASE <cond>
  WHEN <condval1> THEN <expr1>
  [ WHEN <condvalx> THEN <exprx> ] ...
  [ ELSE <expr2> ]
END
```

例如:

```
select
b,
case b
when 1 then 10
when 2 then 20
when 3 then 30
else null
end as t
from tb;
```

```
  b    | t
-------+-----
    0 | NULL
    0 | NULL
    1 | 10
    1 | 10
```

搜索 CASE 表达式

```
CASE WHEN <cond1> THEN <expr1>
   [ WHEN <cond2> THEN <expr2> ] ...
   [ ELSE <expr> ]
END
```

例如:

```
select
b,
case
when b=1 then 10
when b=2 then 20
when b=3 then 30
else null
end as t
from tb;
```

```
  b    | t
-------+-----
    0 | NULL
```

```
    0 | NULL
    1 | 10
    1 | 10
```

NULLIF 表达式

语法:

```
NULLIF ( <expr1>, <expr2> )
```

相当于: IF (<expr1> = <expr2>, NULL, <expr1>)

例如:

```
select b,c,nullif(b,c) from tb;
```

```
   b    |   c    | nullif
-------+------+---------
     0 |      0 | NULL
     0 |      1 |       0
     1 |      0 |       1
     1 |      1 | NULL
```

COALESCE 和 IFNULL 表达式

语法:

```
IFNULL   ( <expr1>, <expr2> )
COALESCE ( <expr1> [, <expr2> [, <expr3> ] ...] )
```

COALESCE首先计算第一个表达式。如果它的值不是 NULL，则直接返回它的值。否则，它返回COALESCE对剩余表达式应用的结果。如果所有表达式都是NULL，则返回NULL。

IFNULL(a, b)相当于COALESCE(a, b)。

例如:

```
select c,b, IFNULL(c,b) from tb;
```

```
 c    |   b    | coalesce
-------+------+-----------
    0 |      0 |         0
    1 |      0 |         1
    0 |      1 |         0
    1 |      1 |         1
    0 |   NULL |         0
    1 |   NULL |         1
  NULL |   NULL |      NULL
```

HUBBLE

6.3.1.5.7 逻辑运算符

语法：

```
NOT <expr>
<expr1> AND <expr2>
<expr1> OR  <expr2>
```

AND和OR是可交换的。此外，输入AND和OR不以任何特定顺序进行评估。如果仅使用另一个操作数就可以完全确定结果，则某些操作数甚至可能根本不求值。

6.3.1.5.8 聚合表达式

聚合表达式的语法与函数调用相同，但有一个特殊情况COUNT：

```
<name> ( <arguments...> )
COUNT ( * )
```

聚合表达式和函数调用的区别在于前者使用聚合函数，并且只能出现在SELECT子句中呈现的表达式列表中。

聚合表达式根据使用的聚合函数计算当前选定的所有行的组合值。

6.3.1.5.9 显式类型强制

语法：

```
<expr> :: <type>
CAST (<expr> AS <type>)
```

计算表达式并将结果值转换为指定的类型。

例如：

```
select now(),CAST(now() AS DATE);
```

```
                now                |            now
+----------------------------------+---------------------------+
  2022-08-09 09:27:21.158674+08:00 | 2022-08-09 00:00:00+00:00
```

6.3.1.5.10 数组构造函数

语法：

```
ARRAY[ <expr>, <expr>, ... ]
```

计算为包含指定值的数组。

例如：

```
select ARRAY[6,5,1] AS a;
```

```
     a
-----------
  {6,5,1}
```

数组的数据类型是从提供的表达式的值推断出来的。数组中的所有位置必须具有相同的数据类型。

HUBBLE

6.3.1.5.11 元组构造函数

语法:

```
(<expr>, <expr>, ...)
ROW (<expr>, <expr>, ...)
```

计算一个包含所提供表达式的值的元组。

例如:

```
select ('a', 13, 2.1) as a;
```

```
       a
----------------
  (a,13,2.1)
```

结果元组的数据类型是从值中推断出来的，元组中的每个位置都可以具有不同的数据类型。

6.3.1.6 表表达式

表表达式在FROM子句的SELECT子句中定义数据源或作为TABLE子句的参数。

6.3.1.6.1 语法图

```
selectOpt ::=
   (table_name '@' index_name|func_application|select_stmt|joined_table) ('WITH'
      ↪  'ORDINALITY')? ('AS')? table_alias_name  (name)?
```

6.3.1.6.2 参数介绍

参数	说明
table_name	表或视图名称。
table_alias_name	在别名表表达式中使用的名称。
name	列名的一个或多个别名，用于别名表表达式。
index_name	强制索引选择的可选语法。
func_application	函数的结果。
row_source_extension_stmt	受支持语句的结果行。
select_stmt	用作子查询的选择查询。
joined_table	一个连接表达式。

6.3.1.6.3 生成数据的表表达式

表和视图名称

语法

```
identifier
identifier.identifier
identifier.identifier.identifier
```

HUBBLE

表表达式中的单个 SQL 标识符指定当前数据库中具有该名称的表、视图或序列的内容。

如果名称由两个或多个标识符组成，则应用名称解析规则。

例如：

```
select * from cust; -- uses table `cust` in the current database
```

```
select * from ora.cust; -- uses table `cust` in database `ora`
```

强制索引选择

强制索引扫描

强制扫描特定索引的语法是：

```
select * from table@my_idx;
```

例如：

```
select * from cust_info@cust_info_cust_card_no_idx;
```

强制部分索引扫描

要强制进行部分索引扫描，您的语句必须有一个WHERE子句。

```
create table te (
  t INT,
  index idx (t) where t > 1);

insert into te(t) values (3);

select * from te@idx where t > 1;
```

强制部分 GIN 索引扫描

要强制进行部分 GIN 索引扫描，语句必须有一个WHERE子句：

- 隐含部分索引。
- 约束 GIN 索引扫描。

```
create table tee (
  j JSON,
  INVERTED INDEX idx (j) WHERE j->'a' = '1');

insert into tee(j)
  values ('{"a": 1}'),
         ('{"a": 3, "b": 2}'),
         ('{"a": 1, "b": 2}');
```

```
select * from tee@idx WHERE j->'a' = '1' AND j->'b' = '2';
```

```
        j
--------------------
  {"a": 1, "b": 2}
```

访问公用表表达式

表表达式中的单个标识符可以引用前面定义的公用表表达式。

例如：

```
with m as (select * from cust)
  select * from m;
```

函数的结果

表表达式可以将函数应用程序的结果用作数据源。

函数名的解析遵循与表名解析相同的规则。有关详细信息，请参阅名称解析。

标量函数作为数据源

当返回单个值的函数用作表表达式时，它被解释为包含函数结果的单列和单行的表格数据。

例如：

```
select * from sin(2.6);
```

```
         sin
----------------------
  0.5155013718214642
```

表生成器函数

一些函数直接从单个函数应用程序生成具有多行的表格数据。这也称为集合返回函数 (SRF)。

例如：

```
select * from generate_series(2,4);
```

```
+-----------------+
| generate_series |
+-----------------+
|               2 |
|               3 |
|               4 |
+-----------------+
```

6.3.1.6.4 扩展表表达式的运算符

别名表表达式

别名表表达式在当前查询的上下文中临时重命名表和列。

语法：

```
<table expr> AS <name>
<table expr> AS <name>(<colname>, <colname>, ...)
```

在第一种形式中，表表达式等价于它的左操作数，为整个表使用了一个新名称，其中的列保留了它们的原始名称。

在第二种形式中，列也被重命名。

例如：

```
select c.x from (select count(*) as x from cust_info) as c;
```

```
select c.x from (select count(*) from cust_info) as c(x);
```

序数注释

将名为的列附加ordinality到表表达式操作数中指定的数据源，其值描述每行的序数。

语法：

```
<table expr> WITH ORDINALITY
```

例如：

```
select * from  (values('b'),('d'),('f')) ;
```

```
+---------+
| column1 |
+---------+
| b       |
| d       |
| f       |
+---------+
```

```
select * from  (values('b'),('d'),('f')) WITH ORDINALITY;
```

```
  column1 | ordinality
----------+-------------
  b       |          1
  d       |          2
  f       |          3
```

6.3.1.6.5 使用另一个查询作为表表达式

使用子查询

可以将括号内的选择查询用作表表达式，这称为子查询。

例如：

```
select c+2 from (select COUNT(*) AS c from cust_info);
```

```
select * from (values(1), (2), (3));
```

```
select ename || ' ' || empno from (TABLE emp);
```

使用另一个语句的输出

语法：

```
with table_expr as ( <stmt> ) select .. from table_expr
```

WITH查询将执行语句的输出指定为行源。支持以下语句作为表表达式的行源：

- DELETE
- EXPLAIN
- INSERT
- SELECT
- SHOW
- UPDATE
- UPSERT

例如：

```
WITH a AS (SHOW COLUMNS from cust) SELECT "column_name" FROM a;
```

```
 column_name
---------------
  cardno
  name
  email
  age
  sal
  rowid
```

6.3.1.6.6 可组合性

可以在SELECT子句和TABLE子句中使用表表达式。因此，它们可以出现在任何可能出现选择子句的地方。例如：

```
SELECT ... FROM <table expr>, <table expr>, ...
TABLE <table expr>
INSERT INTO ... SELECT ... FROM <table expr>, <table expr>, ...
INSERT INTO ... TABLE <table expr>
CREATE TABLE ... AS SELECT ... FROM <table expr>, <table expr>, ...
UPSERT INTO ... SELECT ... FROM <table expr>, <table expr>, ...
```

6.3.1.7 更改列

本文介绍更改列的场景以及用法。

6.3.1.7.1 更改列的名称

```
create  table  sss( id  int primary key ,name string ,card int);
```

```
alter table sss rename column name to names;
```

6.3.1.7.2 更改列的长度

```
create  table cust (
  cardno varchar(10),
  name  string,
  email string,
  age int,
  sal decimal(10,2)
);
```

更改字段长度

```
alter table cust alter cardno type varchar(20);
```

6.3.1.7.3 更改列的类型

情景一：更改字段类型 (主键)

> 主键的字段的数据类型 (如 int 改成 string 或 varchar) 不能被执行，只能通过重建表，然后导入数据。

情景二：更改字段类型 (非索引字段)

> string 改成 varchar(n)，必须保证 n 大于现有数据的最大长度，否则会导致数据缺失。

> 要更改数据的类型，首先要将会话变量设置为 true，可以理解为先开会话，后改数据类型。

```
set enable_experimental_alter_column_type_general = true;
```

int类型改为string类型

```
alter table cust alter age type string;
```

DECIMAL改为string

```
alter table cust alter sal type string;
```

情景三：更改字段数据类型 (索引字段)

如果所在字段有索引 (并且不能是主键)，比如修改 age 的数据类型

```
create table cust (
  cardno varchar(10),
  name string,
  email string,
  age int,
  sal decimal(10,2),
  index(age)
);
```

开启会话变量

```
set enable_experimental_alter_column_type_general = true;
```

先删除索引

```
show index from cust;

drop index cust@cust_age_idx;
```

修改数据类型

```
alter table cust alter age type string;
```

```
explain select * from cust where age='20';
```

```
                                        info
-----------------------------------------------------------------------------------
  ↪
  distribution: full
  vectorized: true

  • filter
    estimated row count: 1
    filter: age = '20'

      • scan
        estimated row count: 1 (100% of the table; stats collected 2 minutes ago
          ↪ )
        table: cust@primary
        spans: FULL SCAN
```

添加索引

```
create index custbak_index on cust(age);
```

```
 explain select * from cust where age='20';
```

```
  filter: age = '20'

      • scan
        estimated row count: 1 (100% of the table; stats collected 2 minutes ago
          ↪ )
        table: cust@primary
        spans: FULL SCAN
                                        info
-----------------------------------------------------------------------------------
  ↪
  distribution: local
```

HUBBLE

```
  vectorized: true

  • index join
    estimated row count: 1
    table: cust@primary

      • scan
        estimated row count: 1 (100% of the table; stats collected 14 minutes
          ↪ ago)
        table: cust@custbak_index
        spans: [/'20' - /'20']
```

6.3.1.8 注释

本文档介绍 Hubble 的注释语法。

6.3.1.8.1 COMMENT ON

该COMMENT ON语句将注释与数据库、表、列或索引相关联。

所需权限

用户必须对他们正在评论的对象具有CREATE特权。

语法图

```
CommentStmt ::=
'COMMENT' 'ON'
('DATABASE' database_name
|'SCHEMA' schema_name
|'TABLE' table_name
|'COLUMN' column_name
|'INDEX' table_index_name)
 'IS' comment_text
```

参数介绍

参数	详情
database_name	您正在查看的数据库的名称
schema_name	您正在查看的架构的名称
table_name	您正在查看的表的名称
column_name	您正在查看的列的名称。
table_index_name	您正在查看的索引的名称
comment_text	STRING您与对象关联的注释。

- 数据库添加注释

```
create database ora;
comment on database ora is '测试数据库';
```

HUBBLE

```
show databases with comment;
```

```
  database_name | owner   | primary_region | regions | survival_goal |   comment
----------------+--------+----------------+---------+---------------+-------------
  ↪
  ora           | root    | NULL           | {}      | NULL          | 测试数据库
```

- 表添加注释

```
create table person(
id int primary key,
name string
);
comment on table person is '用户信息表';
```

```
show tables with comment;
```

```
  schema_name | table_name |    type    | owner   | estimated_row_count | locality
    ↪ |   comment
--------------+------------+----------+--------+---------------------+----------+---------
  ↪
  public      | person     | table    | hubble |                   2 | NULL
    ↪ | 用户信息表
```

- 字段添加注释

```
comment on column person.id is '唯一id';
comment on column person.name is '用户名';
show columns from person with comment;
```

```
  column_name |   data_type   | is_nullable | column_default |
    ↪ generation_expression |   indices   | is_hidden | comment
--------------+--------------+-------------+----------------+----------------------+-----
  ↪
  id          | INT8         |    true     | NULL           |
    ↪                        | {}          |   false    | 唯一id
  name        | STRING       |    true     | NULL           |
    ↪                        | {}          |   false    | 用户名
```

- 从数据库中删除注释

要从数据库中删除评论:

```
comment on database ora is null;
```

6.3.1.8.2 其它注释

- 用-- 注释

HUBBLE

只能注释一行

```
select 1+10  as num;      -- 注释文字
```

```
  num
-------
   11
(1 row)
```

- 用/* */注释

既可以注释一行，更能注释多行

注释一行

```
select 1 /* 这是行内注释文字 */ - 19 as num;
```

```
  num
-------
  -18
```

注释多行

```
select 100+
/*
这个示例属于
多行注释
*/
    1  as 求和;
```

```
  求和
-------
  101
```

别名可以用中文

6.3.1.9 schema 对象名

本文介绍 Hubble 语句中的模式对象名。

6.3.1.9.1 说明

模式对象名用于命名 Hubble 中所有的模式对象，包括database、table、index、column等。在 SQL 语句中，可以通过标识符来引用这些对象。

标识符可以被引号包裹，即select * from t 也可以写成 select * from "t"。

6.3.1.9.2 示例

```
select * from "cust_info" WHERE "cust_info".cust_name = 'liudehua';
```

HUBBLE

在创建表的时候，表名也可以被引号包裹起来

```
create table "user_info"(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );
```

表中的字段也可以用引号包裹

```
create table user_cust(
    cust_no          string primary key,
    cust_name        varchar(30) not null,
    cust_card_no     varchar(18),
    "cust_phoneno"   decimal(15),
    cust_address     varchar(30),
    cust_type        varchar(10),
    index(cust_card_no)
 );
```

在查询语句中，别名部分可以用标识符或者字符串：

```
select 1 as "one col",
       2 as "two col";
```

```
  one col | two col
----------+----------
        1 |        2
```

对象名字有时可以被限定或者省略。例如在创建表的时候可以省略数据库限定名：

```
create table a (b int);
```

如果之前没有使用 USE 指定到特定的数据库，会报 No database selected 错误，此时可以指定数据库限定名：

```
create table test.a (b int);
```

.的左右两端可以出现空格，`table_name.colname` 等同于 `table_name . colname`。

如要应用这个模式对象，请使用：

```
table_name.colname
```

或

```
"table_name"."colname"
```

或

```
table_name."colname"
```

或

```
"table_name".colname
```

6.3.1.10 属性

6.3.1.10.1 UUID

概述

数据类型UUID存储由 RFC 4122、ISO/IEC 9834-8:2005 以及相关标准定义的通用唯一标识符 (UUID)。

UUID(Universally Unique Identifier) 是一个 128 位长度的唯一标识符，它可以用来标识计算机系统中的实体。UUID的生成算法保证了它在全球范围内的唯一性，即使在不同的计算机系统和网络中也是如此。

UUID有多种版本，最常见的是基于时间戳的版本和随机数的版本。

基于时间戳的UUID使用当前的时间戳和计算机的 MAC 地址生成，确保在同一台计算机上生成的UUID具有唯一性，并且可以根据时间戳的顺序进行排序。

随机数的UUID则是完全随机生成的，不依赖于任何计算机特定的信息。这种UUID的唯一性是基于随机数生成算法的强大性能保证的。

UUID在很多领域得到广泛应用，比如数据库的主键、分布式系统中的节点标识、文件和目录的唯一标识等。由于其高度的唯一性和不可预测性，UUID在避免冲突和提高系统安全性方面有着重要的作用。

定义

UUID使用 16 进制表示，共有 36 个字符组成，格式为 8-4-4-4-12。

示例：

```
xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
3ea70a2e-51c6-4c39-9c7a-87b55a3c2d3c
21457791-1d8f-44c2-9cf7-1f9df79518ba
```

作用

UUID的目的是让分布式系统中的所有元素都能有唯一的识别信息。如此一来，每个人都可以创建不与其它人冲突的UUID，就不需考虑数据库创建时的名称重复问题。其作用视场景而定。

首先，想象一下一张客户表。例如，我们不希望使用cust_name或cust_addr作为唯一标识符之类的字段，因为可能有多个客户可能具有相同的姓名或共享相同的地址。然而，为每一行分配唯一的标识是个好办法；

其次，在建表过程中，无法使用有效的业务字段作为主键；

因此，建议用UUID来作为主键列。此函数生成的值保证是唯一的，并且在集群中分布良好。

UUID被广泛使用的部分原因是它们很可能在整个环境内是唯一的，这意味着我们的行的UUID不仅在我们的数据库表中是唯一的。

示例

要自动生成唯一的行标识符，使用UUID带有gen_random_uuid()函数的列作为默认值。

数据库，尤其是分布式数据库，具有内置的UUID生成。例如，在 Hubble 中，我们建议使用UUID作为行标识符，这样做就像使用gen_random_uuid()函数一样简单

创建一张带有uuid的表

```
create table cust_info (
        id uuid not null DEFAULT gen_random_uuid(),
        city string not null,
        name string null,
        address string null,
        credit_card string null,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC)
);
```

向表中插入数据

```
insert into cust_info (name, city) values ('liu dehua', 'beijing'), ('gao
    ↪ yuanyuan', 'shanghai'), ('zhen zidan', 'hk');
```

```
select * from cust_info;
```

```
                  id                   |   city   |     name      | address |
                      ↪ credit_card
---------------------------------------+----------+---------------+---------+----------
    ↪
  3ea70a2e-51c6-4c39-9c7a-87b55a3c2d3c| beijing  | liu dehua     | NULL    | NULL
  95b1d540-eb0f-4c80-84d0-3bb576fd6487| hk       | zhen zidan    | NULL    | NULL
  21457791-1d8f-44c2-9cf7-1f9df79518ba| shanghai | gao yuanyuan  | NULL    | NULL
```

6.3.1.10.2 序列

序列 (SEQUENCE) 是序列号生成器，可以为表中的行自动生成序列号，产生一组等间隔的数值 (类型为数字)；不占用磁盘空间，占用内存。

其主要用途是生成表的主键值，可以在插入语句中引用，也可以通过查询检查当前值，或使序列增至下一个值。

```
create sequence sequence_name
MINVALUE  1
MAXVALUE  n
INCREMENT 1
START     1
```

1、INCREMENT用于定义序列的步长，如果省略，则默认为 1。

2、START定义序列的初始值 (即产生的第一个值)，默认为 1。

3、MAXVALUE定义序列生成器能产生的最大值。对于递减序列，最大值是-1。

4、MINVALUE定义序列生成器能产生的最小值。对于递增序列，最小值是 1。

- 创建SEQUENCE

```
CREATE SEQUENCE user_seq
MINVALUE 1
MAXVALUE 9223372036854775807
INCREMENT 1
START 1
```

```
show create user_seq;
```

```
table_name |     create_statement
-----------+------------------------------
userid_seq | CREATE SEQUENCE user_seq
           | MINVALUE 1
           | MAXVALUE 9223372036854775807
           | INCREMENT 1
           | START 1
```

建表时指定主键根据sequence自动增长

```
create table user_table(
  user_id int PRIMARY KEY DEFAULT nextval('user_seq'),
  phone_number string,
  create_date date
);
```

插入几条测试数据

```
insert into user_table (phone_number, create_date)
values
('13920163254','2012-11-21'),
('15122537891','2004-05-05'),
('18812680426','2018-10-20');
```

```
select * from user_table;
```

```
  user_id | phone_number | create_date
----------+--------------+--------------
        1 | 13920163254  | 2012-11-21
        2 | 15122537891  | 2004-05-05
        3 | 18812680426  | 2018-10-20
```

当数据被删除之后，仍然可以从删除的位置的序列值自增

```
truncate table user_table;
```

```
select * from user_table;
```

```
  user_id | phone_number | create_date
----------+--------------+--------------
(0 rows)
```

```
insert into user_table (phone_number, create_date)
                          values
                          ('13954775785','2017-09-23'),
                          ('13148934774','2012-11-24'),
                          ('18597548993','2014-08-13');
```

```
select * from user_table;
```

```
  user_id | phone_number | create_date
----------+--------------+--------------
        4 | 13954775785  | 2017-09-23
        5 | 13148934774  | 2012-11-24
        6 | 18597548993  | 2014-08-13
```

- 查看序列的当前值

要查看当前值而不增加序列，请使用：

```
select * from user_seq;
```

> 使用SEQUENCE比使用内置函数gen_random_uuid()、uuid_v4()、unique_rowid()自动生成的唯一 ID 要慢。

6.3.1.10.3 unique_rowid()

如果生成的 ID 必须存储在相同的键值范围内，则可以使用整数类型使用unique_rowid()功能作为默认值

```
create table user_info (
        id int default unique_rowid(),
        city string not null,
        name string null,
        CONSTRAINT "primary" PRIMARY KEY (city asc, id asc),
        FAMILY "primary" (id, city, name)
);
```

```
insert into user_info (name, city) VALUES ('liudehua', 'HK'), ('huangxiaomimg',
    ↪ 'shanghai'), ('wanglikun', 'meimeng');
```

```
select * from user_info;
```

```
          id          |   city   |      name
----------------------+----------+----------------
  891846286839218177 | HK       | liudehua
  891846286839349249 | meimeng  | wanglikun
  891846286839316481 | shanghai | huangxiaomimg
```

6.3.1.10.4 三种属性值的比较

	UUID	unique_rowid()	序列
大小	16 字节	8 字节	1 至 8 字节
排序属性	无序高度	有序	高度有序
价值分配	均匀分布 (128 位)	包含时间和空间 (节点 ID) 组件	密集、小数值
数据位置	最大分布	近距离生成的值位于同一位置	高度本地化
insert 用作密钥时的延迟	小，对并发不敏感	小，但随着并发 INSERT 的增加而增加	较高的
insert 用作密钥时的吞吐量	最高	受 1 个节点上的最大吞吐量限制	受 1 个节点上的最大吞吐量限制
用作密钥时的读取吞吐量	最高 (最大并行度)	有限	有限

6.3.1.11 关键字

SQL 语句由两个基本组件组成：

- 关键字：在 SQL 中具有特定含义的单词，如`UNIQUE`、`CONSTRAINT`等
- 标识符：数据库和函数等事物的名称

6.3.1.11.1 关键字

关键字构成 SQL 的词汇表，并且能够在语句中拥有特定的含义。Hubble 支持的每个 SQL 关键字包括以下四个方面之一：

- 保留关键字
- 键入函数名称关键字
- 列名关键字
- 未保留的关键字

保留关键字具有固定的含义，通常不允许用作标识符。所有其他类型的关键字都被认为是非保留的；它们在某些上下文中具有特殊含义，并且可以在其他上下文中用作标识符。

关键字使用

多数询问关键字的使用者希望从以下方面了解更多关于它们的信息：

- 对象的名称，在本页标识符中介绍
- 语法，在我们的 SQL 语句和 SQL 语法页面中介绍

6.3.1.11.2 标识符

标识符最常用作数据库、表或列等对象的名称，因此，术语“名称”和“标识符”经常互换使用。

标识符规则

在我们的 SQL 语法中，所有接受定义的值需要满足以下条件：

- 以 Unicode 字母或下划线(`_`)开头。后续字符可以是字母、下划线、数字 (0-9) 或美元符号(`$`)。
- 不与任何 SQL 关键字相同，除非该关键字被元素的语法所接受。例如，`name`接受 `Unreserved`或`Column Name`关键字 (即便有的关键字被元素的语法所接受，我们也建议避免使用任何关键字)。

要绕过这些规则中的任何一个，需用双引号将标识符括起来。您还可以使用双引号来保持数据库、表、视图和列名中的大小写敏感性。但是，对此类标识符的所有引用都必须还包括双引号。

6.3.1.11.3 示例

对于保留字，必须使用引号包裹，才能作为标识符被使用。例如：

```
create table insert (n int);
```

```
invalid syntax: statement ignored: at or near "insert": syntax error
```

正确的引用方式：

```
create table "insert" (n int);
```

而非保留字则不需要引号也能直接作为标识符，例如BEGIN是非保留字：

```
create table "select" (begin string);
```

特殊情况下，如果应用了限定符.，那么也可以不用引号：

```
create table test.select (close int);
```

6.3.1.11.4 关键字列表

下表列出了 Hubble 中关键字，其中保留字用 (A) 来标识，窗口函数的保留字用 (A-Window) 来标识。

A

- ACTION
- ADD(A)
- ADMIN(A)
- AGO
- ALL(A)
- ALTER(A)
- ANALYZE(A)
- AND(A)
- ANY
- ARRAY(A)
- AS(A)
- ASC(A)
- ASCII
- ATTRIBUTE
- ATTRIBUTES
- AVG

B

- BACKUP
- BACKUPS
- BEGIN
- BETWEEN(A)
- BIGINT(A)

HUBBLE

- BIT
- BLOB(A)
- BOTH(A)
- BUCKETS(A)
- BY(A)
- BYTE

C

- CACHE
- CALL(A)
- CANCEL(A)
- CAPTURE
- CASCADE(A)
- CASCADED
- CASE(A)
- CHANGE(A)
- CHAR(A)
- CHARACTER(A)
- CHARSET
- CHECK(A)
- CLIENT
- CLOSE
- CLUSTER
- COALESCE
- COLUMN(A)
- COLUMNS
- COMMENT
- COMMIT
- COMMITTED
- CONTEXT
- CONTINUE(A)
- CONVERT(A)
- CPU
- CREATE(A)
- CURRENT
- CURRENT_DATE(A)
- CURRENT_ROLE(A)
- CURRENT_TIME(A)
- CURRENT_TIMESTAMP(A)
- CURRENT_USER(A)
- CURSOR(A)
- CYCLE

D

- DATA
- DATABASE(A)
- DATABASES(A)

HUBBLE

- DATE
- DATETIME
- DAY
- DDL(A)
- DECIMAL(A)
- DECLARE
- DEFAULT(A)
- DEFINER
- DELETE(A)
- DENSE_RANK (A-Window)
- DESC(A)
- DIRECTORY
- DISABLE
- DIV(A)
- DO
- DOUBLE(A)
- DROP(A)

E

- ELSE(A)
- ELSEIF(A)
- ENABLE
- ENABLED
- END
- ENUM
- ERROR
- ERRORS
- EVENT
- EVENTS
- EXCEPT(A)
- EXECUTE
- EXISTS(A)
- EXIT(A)
- EXPLAIN(A)

F

- FAULTS
- FETCH(A)
- FIELDS
- FILE
- FIRST
- FLOAT(A)
- FLUSH
- FOR(A)
- FORCE(A)
- FOREIGN(A)
- FORMAT

HUBBLE

- FOUND
- FROM(A)
- FULL
- FUNCTION

G

- GENERAL
- GLOBAL
- GRANT(A)
- GRANTS
- GROUP(A)
- GROUPS (A-Window)

H

- HASH
- HAVING(A)
- HELP
- HISTORY
- HOSTS
- HOUR

I

- IDENTIFIED
- IF(A)
- ILIKE(A)
- IMPORT
- IN(A)
- INDEX(A)
- INDEXES
- INFILE(A)
- INNER(A)
- INOUT(A)
- INSERT(A)
- INSTANCE
- INT(A)
- INT2(A)
- INT4(A)
- INT8(A)
- INTEGER(A)
- INTERSECT(A)
- INTERVAL(A)
- INTO(A)
- INVOKER
- IO
- IS(A)
- ISOLATION
- ISSUER

HUBBLE

J

- JOB(A)
- JOBS(A)
- JOIN(A)
- JSON

K

- KEY(A)
- KEYS(A)

L

- LAG (A-Window)
- LAST
- LAST_VALUE (A-Window)
- LEAD (A-Window)
- LEAVE(A)
- LEFT(A)
- LESS
- LEVEL
- LIKE(A)
- LIMIT(A)
- LIST
- LOAD(A)
- LOCAL
- LOCATION
- LOCK(A)
- LOCKED
- LOGS
- LONG(A)

M

- MASTER
- MATCH(A)
- MEMBER
- MEMORY
- MINUTE
- MIN_ROWS
- MOD(A)
- MODE
- MODIFY
- MONTH

N

- NAMES
- NEVER
- NEXT
- NEXTVAL

HUBBLE

- NO
- NODE_ID(A)
- NONE
- NOT(A)
- NOWAIT
- NULL(A)
- NULLS

O

- OF(A)
- OFF
- OFFSET
- ON(A)
- ONLINE
- ONLY
- OPEN
- OPTIMIZE(A)
- OPTION(A)
- OR(A)
- ORDER(A)
- OUT(A)
- OVER (A-Window)

P

- PARTITION(A)
- PARTITIONS
- PASSWORD
- PAUSE
- PERCENT
- PERCENT_RANK (A-Window)
- PLUGINS
- POINT
- POLICY
- PRIMARY(A)

Q

- QUERY
- QUICK

R

- RANGE(A)
- RANK (A-Window)
- READ(A)
- REAL(A)
- REBUILD
- RECOVER
- REGEXP(A)
- REGION(A)

HUBBLE

- REGIONS(A)
- RELEASE(A)
- RELOAD
- REMOVE
- RENAME(A)
- REPAIR
- REPEAT(A)
- REPLACE(A)
- REPLICA
- REPLICAS
- REPLICATION
- RESOURCE
- RESPECT
- RESTART
- RESTORE
- RESUME
- REUSE
- REVERSE
- REVOKE(A)
- RIGHT(A)
- RLIKE(A)
- ROLE
- ROLLBACK
- ROUTINE
- ROW(A)
- ROW_COUNT
- ROW_FORMAT
- ROW_NUMBER (A-Window)
- ROWS (A-Window)
- RTREE

S

- SAMPLES(A)
- SAVEPOINT
- SECOND
- SELECT(A)
- SEPARATOR
- SEQUENCE
- SERIAL
- SERIALIZABLE
- SESSION
- SET(A)
- SETVAL
- SHARE
- SHARED
- SHOW(A)
- SHUTDOWN

HUBBLE

- SIGNED
- SIMPLE
- SKIP
- SLAVE
- SLOW
- SMALLINT(A)
- SOME
- SOURCE
- SPATIAL(A)
- SPLIT(A)
- SQL(A)
- SSL(A)
- START
- STATUS
- STORAGE
- SUBJECT
- SUPER
- SYSTEM

T

- TABLE(A)
- TABLES
- TEMPORARY
- TEXT
- THAN
- THEN(A)
- TIME
- TIMESTAMP
- TINYBLOB(A)
- TINYINT(A)
- TINYTEXT(A)
- TO(A)
- TOPN(A)
- TRACE
- TRANSACTION
- TRUE(A)
- TRUNCATE
- TTL
- TYPE

U

- UNICODE
- UNION(A)
- UNIQUE(A)
- UNKNOWN
- UNLOCK(A)
- UNTIL(A)

HUBBLE

- UPDATE(A)
- USAGE(A)
- USE(A)
- USER
- USING(A)

V

- VALUE
- VALUES(A)
- VARBINARY(A)
- VARCHAR(A)
- VARCHARACTER(A)
- VARIABLES
- VARYING(A)
- VIEW

W

- WAIT
- WARNINGS
- WEEK
- WHEN(A)
- WHERE(A)
- WHILE(A)
- WINDOW (A-Window)
- WITH(A)
- WITHOUT
- WRITE(A)

Y

- YEAR

6.3.2 SQL 语句

6.3.2.1 DCL

(Data Control Language)：数据控制语言，用来定义访问权限和安全级别。

6.3.2.1.1 USER

SHOW USERS

所需权限

查看所有的用户，需要admin权限。

语法图

```
ShowUserStmt
          ::= 'SHOW' 'USERS'
```

展示用户

```
show users;
```

CREATE USER

CREATE USER 语句创建库用户，使你可以控制数据库的表的权限。

事项说明

角色名:

- 不区分大小写
- 必须以字母或下划线开头
- 必须仅包含字母，数字或下划线
- 必须介于 1 到 63 个字符之间
- 创建用户后，必须授予他们对数据库和表的权限
- 在 secure 集群上，你必须为用户创建客户端证书，并且用户必须验证其对集群的访问权限

所需权限

要创建其他用户，该用户必须是管理员角色或具有创建者角色。

语法图

```
    CreateUserStmt ::=
       'CREATE' 'USER' ('IF' 'NOT' 'EXISTS')? name



       ('WITH' 'PASSWORD') ?  password
```

参数介绍

参数	简介
if not exists	仅当数据库中不存在同名时，才创建新用户
name	要创建的用户名

创建用户 (不含密码，不判断用户是否存在)

```
create user beagledata;
```

创建用户 (包含密码，不判断用户是否存在)

```
create user beagledata with password 'beagledatapwd';
```

创建用户 (能够设置密码的有效期限)

```
create user beagledata with login password 'beagledatapwd' valid until '
    ↪ 2023-10-24';
```

创建用户 (包含密码，并判断用户是否存在)

```
create user if not exists beagledata with password 'beagledatapwd';
```

创建用户 (能够创建其他用户并且管理身份认证)

```
create user do_create_user with createrole createlogin;
```

创建用户 (能够创建数据库)

```
create user do_create_user with createdb;
```

创建用户 (能够取消、暂停、恢复非管理员的作业)

```
create user do_control_job with controljob;
```

DROP USER

所需权限

非管理员用户无法删除管理员用户。要删除非管理员用户，该用户必须是管理员角色或具有创建者角色。

用户被删除前，必须删除该用户的所有权限。

语法图

```
    DropUserStmt ::=
       'DROP' 'USER' ( 'IF' 'EXISTS' )? user_name ( ',' user_name )*
```

参数介绍

参数	简介
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
user_name	数据库用户名

展示针对单个表的权限

```
show grants on hubble.orders for beagledata;
```

```
+--------+------------+------------+
| Table  |     User   | Privileges |
+--------+------------+------------+
| orders | beagledata | CREATE     |
| orders | beagledata | INSERT     |
| orders | beagledata | UPDATE     |
+--------+------------+------------+
(3 rows)
```

展示针对用户的所有权限:

```
show grants for beagledata;
```

回收表的权限 (被收回的权限需是显式授权给用户的权限):

```
revoke create,insert,update on hubble.orders from beagledata;
```

删除用户 (**当用户显式授权未全部收回时，无法删除用户**):

```
drop user beagledata;
```

ALTER USER

该ALTER USER语句可用于添加、更改或删除用户的密码以及更改用户的角色选项

所需权限

要更改其他用户，该用户必须是管理员角色或具有创建者角色。

语法图

```
    AlterUserStmt ::=
       'ALTER' 'USER' ( 'IF' 'EXISTS' )? user_name 'WITH' 'PASSWORD' password
```

参数介绍

参数	简介
password	数据库的密码
user_name	数据库用户名

修改用户只需要将 create 关键字换成 alter

例如：修改密码

```
alter user beagledata with password 'password123';
```

6.3.2.1.2 ROLE

该SHOW ROLES语句列出了所有数据库的角色

SHOW ROLES

查看所有数据库上的角色，需要admin权限。

语法图

```
    ShowRoleStmt
             ::= 'SHOW' 'ROLES'
```

展示角色语句

```
show roles;
```

CREATE ROLE

CREATE ROLE语句创建 SQL 角色

所需权限

创建一个角色，除非角色是管理员角色或者角色必须具有创建者权限。

语法图

```
    CreateRoleStmt ::=
       'CREATE' 'ROLE' IfNotExists RoleSpec (',' RoleSpec)*
```

```
IfNotExists  ::=
   ('IF' 'NOT' 'EXISTS')?


RoleSpec ::=
   name
```

参数介绍

参数	简介
name	要创建的角色的名称
if not exists	仅当数据库中不存在同名时，才创建新角色

创建角色示例

```
create role uat;
```

创建角色 (包含密码，不判断用户是否存在)

```
create role beagledata with password 'beagledatapwd';
```

创建角色 (能够设置密码的有效期限)

```
create role beagledata with login password 'beagledatapwd' valid until '
    ↪ 2023-10-24';
```

创建角色 (能够创建其他用户并且管理身份认证)

```
create role do_create_user with createrole createlogin;
```

创建角色 (能够创建数据库)

```
create role do_create_user with createdb;
```

创建角色 (能够取消、暂停、恢复非管理员的作业)

```
create role do_control_job with controljob;
```

DROP ROLE

该DROP ROLE语句删除一个或多个角色

事项说明

- 该admin角色不能被删除，并且root必须始终是的成员admin。
- 如果角色具有特权，则不能删除它。用于REVOKE删除权限。
- 拥有对象（如数据库、表、模式和类型）的角色在所有权转移给另一个角色之前不能被删除。

所需权限

若要删除非管理员角色，该角色必须是管理员角色或具有创建者角色。

语法图

```
    DropRoleStmt ::=
       'DROP' 'ROLE' ( 'IF' 'EXISTS' )? RolenameList



    RolenameList ::=
        Rolename ( ',' Rolename )*
```

参数介绍

参数	简介
Rolename	角色的名称
IF EXISTS	如果存在则删除角色

展示用的角色

```
show grants on table_name for uat;
```

```
+------------+--------+-----------+---------+------------+
|  Database  | Schema |   Table   |  User   | Privileges |
+------------+--------+-----------+---------+------------+
| test       | public | table_name|  uat    | INSERT     |
+------------+--------+-----------+---------+------------+
```

收回用户的权限才可以删除

```
revoke insert on  table_name from uat;
```

删除示例:

```
drop role uat;
```

6.3.2.1.3 PRIVILEGE

显示权限授予

使用以下语法显示授予用户对数据库对象的权限，DATABASE省略时，将列出当前数据库中的模式、表和类型

```
show grants [ON [DATABASE | SCHEMA | TABLE | TYPE] <targets...>] [FOR <users
    ↪ ...>]
```

显示角色授权

使用以下语法显示集群中用户的角色授予

```
show grants  on role [<roles...>] [FOR <users...>]
```

SHOW GRANTS

列出所有数据库对象的权限信息。

```
show grants ;
```

列出角色是admin的用户。

```
show grants on role admin;
```

列出admin角色的权限信息。

```
show grants for admin;
```

列出某个数据库的权限信息。

```
show grants on database my_hubble_database;
```

列出beagledata用户在数据库my_hubble_database的权限信息。

```
show grants on database my_hubble_database for beagledata;
```

列出某张表的权限信息。

```
show grants on table my_hubble_table;
```

GRANT

用于添加一个角色或者角色中的一个用户。

所需权限

授予角色成员资格的用户必须是角色管理员或者是admin角色的成员

- 用户和角色可以作为角色成员。
- root 用户会自动作为 admin 角色且对所有数据库拥有 ALL 权限。
- 角色成员会自动继承其角色的所有权限。

GRANT用于控制每一个角色或者用户和指定数据库或数据表上拥有的sql权限，对于指定语句所需要的权限，请查看各自相关sql语句的文档。

支持的权限如下：

权限	级别
ALL	Database,Table
CREATE	Database,Table
DROP	Database,Table
GRANT	Database,Table
SELECT	Table
INSERT	Table
DELETE	Table
UPDATE	Table

语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType Objectname 'TO' UserSpecList
```

```
PrivElemList ::=
    PrivElem ( ',' PrivElem )*


PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?


PrivType ::=
    'ALL' 'PRIVILEGES'?
|    'ALTER' 'ROUTINE'?
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|   'DELETE'
|   'DROP' 'ROLE'?
|   'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|   'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'


ObjectType ::=
    'TABLE'?
|   'DATABASE'


UserSpecList ::=
    UserSpec ( ',' UserSpec )*
```

参数介绍

参数	简介
PrivType	一般指的是具体的权限类型
ObjectType	对象类型，一般是TABLE或DATABASE
Objectname	具体的数据库名或者表名
UserSpec	具体角色或者权限名称

给用户授予数据库的权限

```
grant create on database hubbletest to beagledata;
```

```
show  grants on database hubbletest;
```

给角色授予数据库的权限

```
grant create on database hubbletest to uat;
```

```
show  grants for uat;
```

给用户授予某张表的权限

```
grant delete  on table hubbletest.orders to beagledata;
```

```
show  grants  on table hubbletest.orders;
```

授予用户 (数据库 test 下) 所有表insert和update权限

```
grant insert,update  on table test.* to do_create_user;
```

创建用户后根据实际需要赋予 admin 权限，一般情况下，同一应用系统只有一个用户能拥有 admin 权限

```
grant admin to beagledata with admin option;
```

REVOKE

该REVOKE语句撤销用户和或角色的权限。

所需权限

- 要撤消权限，撤消权限的用户必须GRANT对目标数据库、模式、表或用户定义类型具有权限。除了GRANT特权之外，撤销特权的用户还必须对目标对象具有被撤销的特权。例如，一个用户将SELECT一个表的权限撤销给另一个用户，就必须拥有该表的GRANT和SELECT权限。
- 要撤销角色成员资格，撤销角色成员资格的用户必须是角色管理员（即，带有的成员WITH ADMIN OPTION）或admin角色成员。要删除admin角色的成员资格，用户必须拥有WITH ADMIN OPTION该admin角色。

支持的权限如下：

权限	级别
ALL	Database,Table
CREATE	Database,Table
DROP	Database,Table
GRANT	Database,Table
SELECT	Table
INSERT	Table
DELETE	Table
UPDATE	Table
USAGE	Schema, Type
ZONECONFIG	Database, Table

语法图

```
RevokeStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType Objectname 'FROM' UserSpecList
```

```
PrivElemList ::=
    PrivElem ( ',' PrivElem )*


PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?


PrivType ::=
    'ALL' 'PRIVILEGES'?
|    'ALTER' 'ROUTINE'?
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|   'DELETE'
|   'DROP' 'ROLE'?
|   'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|   'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'


ObjectType ::=
    'TABLE'?
|   'DATABASE'


UserSpecList ::=
    UserSpec ( ',' UserSpec )*
```

参数介绍

参数	简介
PrivType	一般指的是具体的权限类型
ObjectType	对象类型，一般是TABLE或DATABASE
Objectname	具体的数据库名或者表名
UserSpec	具体角色或者权限名称

回收用户的数据库的权限

```
revoke create on database hubbletest from beagledata;
```

```
show grants on database hubbletest;
```

回收给角色的数据库的权限

```
revoke create on database hubbletest from uat;
```

```
show grants on database hubbletest for uat;
```

回收给用户授予的表的权限

```
revoke delete on table hubbletest.orders from beagledata;
```

```
show grants on table hubbletest.orders;
```

6.3.2.2 DDL

DDL 语句用于创建数据库对象，包括库、表、视图和索引等；用户在进行 DDL 操作时，系统首先判断用户是否拥有执行此项操作的权限。

6.3.2.2.1 ADD COLUMN

在表内添加一个新列，ADD COLUMN是ALTER TABLE的子命令，用于ADD COLUMN向现有表添加列。

所需权限

用户必须有表的CREATE权限。

语法图

```
AlterTableStmt  ::=
     'ALTER'  'TABLE' ('IF' 'EXISTS')? table_name  AddColumnStmt


AddColumnStmt   ::=
     'ADD' 'COLUMN' ( 'IF' 'NOT' 'EXISTS')? column_name typename
        ↪ ColQualification


ColQualification  ::=
     col_qualification
```

参数介绍

参数	说明
table_name	需要添加列的表
column_name	需要添加列的名称（必须满足命名规则，且在表内唯一，但可与索引、约束同名）
typename	需要添加新列的数据类型
col_qualification	列定义的可选列表，可能包括列级约束，排序规则或列族分配。需要注意，无法直接添加具有外键约束的列。可以添加没有约束的列，然后使用CREATE INDEX索引列，再使用ADD CONSTRAINT将外键约束添加到列。

- 创建表

```
create table cust (cust_id int);
```

- 添加单列

```
alter  table cust add column age int;
```

- 添加多列

```
alter  table cust add column address string,add column iphone_number int;
```

- 添加具有NOT NULL约束和DEFAULT值的列

```
alter table cust add column money decimal not null default (decimal '123456789')
    ↪ ;
```

- 添加带有UNIQUE约束列，并且值非空

```
alter table cust add column tel decimal unique not null;
```

- 添加带有排序规则的列

```
alter table cust add column email string collate en;
```

- 添加一列并将其分给新的列族，用CREATE FAMILY语句

```
alter table cust add column creat_time date create family new_time;
```

- 添加一列分给现有列族

```
alter table cust add column time string family new_time;
```

- 如果列族不存在，则添加列并创建列族

```
alter table cust add column new_time string create if not exists family n_time;
```

- 查看建表语句

```
show create table cust;
```

```
  table_name |                                    create_statement
-------------+-------------------------------------------------------------------------------------
    ↪
  cust       | CREATE TABLE public.cust (
             |     cust_id INT8 NULL,
             |     rowid INT8 NOT VISIBLE NOT NULL DEFAULT unique_rowid(),
             |     age INT8 NULL,
             |     address STRING NULL,
             |     iphone_number INT8 NULL,
             |     money DECIMAL NOT NULL DEFAULT 123456789:::DECIMAL,
             |     tel DECIMAL NOT NULL,
             |     email STRING COLLATE en NULL,
```

```
              |     creat_time DATE NULL,
              |     "time" STRING NULL,
              |     new_time STRING NULL,
              |     CONSTRAINT "primary" PRIMARY KEY (rowid ASC),
              |     UNIQUE INDEX cust_tel_key (tel ASC),
              |     FAMILY "primary" (cust_id, rowid, age, address, iphone_number
                 ↪ , money, tel, email),
              |     FAMILY new_time (creat_time, "time"),
              |     FAMILY n_time (new_time)
              | )
```

6.3.2.2.2 ADD CONSTRAINT

用于为列添加约束：

- Check
- Foreign Keys
- UNIQUE

主键的NOT NULL约束只能通过CREATE TABLE创建，DEFAULT约束通过ALTER COLUMN管理。要将主键约束添加到表中，应该在创建表时显式定义主键。要替换现有的主键，可以使用ADD CONSTRAINT ... PRIMARY KEY。

所需权限

需要有表的CREATE权限。

语法图

```
AlterConstraintStmt  ::=
'ALTER' 'TABLE' ('IF' 'EXISTS')? table_name 'ADD' 'CONSTRAINT' con_name con_elem

Con_Name ::=
constraint_name

Con_Elem ::=
constraint_elem ?
```

参数介绍

参数	说明
table_name	包含要约束的列的表的名称
constraint_name	约束的名称，它必须对其表唯一并遵循这些标识符规则
constraint_elem	要添加的CHECK，foreign key约束。添加、更改约束是通过. 不支持通过添加、更改表格；它只能在表创建期间指定。

查看架构更改

此架构更改语句已注册为作业。您可以使用SHOW JOBS查看长时间运行的作业。

更改主键 ADD CONSTRAINT ... PRIMARY KEY

当您使用更改主键时ALTER TABLE ... ALTER PRIMARY KEY，现有的主键索引将变为二级索引。创建的二级索

引ALTER PRIMARY KEY会占用节点内存，并且会降低集群的写入性能。

如果满足以下条件之一，可以使用ADD CONSTRAINT ... PRIMARY KEY向现有表添加主键：

- 在创建表时没有明确定义主键。在这种情况下，表是使用默认主键rowid创建的。使用DROP CONSTRAINT ↪ ... PRIMARY KEY删除默认主键并用新的主键替换它。
- 在同一事务中，DROP CONSTRAINT语句在ADD CONSTRAINT ... PRIMARY KEY语句之前。具体信息，请参阅下文删除并添加主键约束。
- 建表语句

```
create table shop
(id int,
name varchar(20),
balance  DECIMAL(10,2)
);
```

- 添加UNIQUE约束

要求该列的每一个值都是唯一的，NULL除外

```
alter table shop add constraint id_unique unique (id);
```

- 添加约束：check语法

```
alter table shop add constraint balance_check check (balance > 0);
```

在添加约束的过程中 Hubble 将运行一个后台作业来验证现有的表数据，如果 Hubble 在验证步骤中发现了违反约束的行，则ADD CONSTRAINT语句将失败。

在以下情况下，将回滚整个事务，包括添加的任何新列：

- 如果发现现有列包含违反新约束的值。
- 如果新列具有默认值或者是包含违反新约束的值的计算列。
- 查看约束

```
show  constraints from shop;
```

```
  table_name | constraint_name | constraint_type |        details        |
     ↪ validated
-------------+-----------------+-----------------+-----------------------+-------------
   ↪
  shop       | balance_check   | CHECK           | CHECK ((balance > 0)) |
     ↪ true
  shop       | id_unique       | UNIQUE          | UNIQUE (id ASC)       |
     ↪ true
```

- 添加FOREIGN KEY约束

> 在添加FOREIGN KEY约束前，该列必须已添加索引。如果该列未索引，使用CREATE INDEX来创建索引，然后再通过ADD CONSTRAINT 添加 FOREIGN KEY 约束

```
create table customers (
      id int primary key,
      name string,
      email string
  );
```

```
create table shipments (
      tracking_number uuid default gen_random_uuid() primary key,
      carrier string,
      status string,
      customer_id int
    );
```

```
alter table shipments add constraint fk_customers foreign key (customer_id)
    ↪ references customers(id);
```

- 删除并添加主键约束

比如：要添加name到cust_info表的复合主键，而不创建现有主键的二级索引。

建表语句如下：

```
create table cust_info (
                  id UUID NOT NULL,
                  city VARCHAR NOT NULL,
                  name VARCHAR NULL,
                  address VARCHAR NULL,
                  credit_card VARCHAR NULL,
                  CONSTRAINT cust_info_pkey PRIMARY KEY (city ASC, id ASC)
              );
```

首先，使用向列添加NOT NULL约束。

```
ALTER TABLE cust_info ALTER COLUMN name set not null;
```

然后，在同一个事务中，DROP现有主键约束和ADD新约束：

```
begin;
alter table cust_info DROP CONSTRAINT cust_info_pkey;
alter table cust_info ADD  CONSTRAINT cust_info_pkey PRIMARY KEY (city, name, id
    ↪ );
commit;
```

```
show create table cust_info;
```

```
 table_name |                              create_statement
-------------+-----------------------------------------------------------------------
    ↪
  cust_info  | CREATE TABLE public.cust_info (
```

```
                |     id UUID NOT NULL,
                |     city VARCHAR NOT NULL,
                |     name VARCHAR NOT NULL,
                |     address VARCHAR NULL,
                |     credit_card VARCHAR NULL,
                |     CONSTRAINT cust_info_pkey PRIMARY KEY (city ASC, name ASC, id
                  ↪  ASC),
                |     FAMILY "primary" (id, city, name, address, credit_card)
                | )
```

6.3.2.2.3 CREATE DATABASE

该CREATE DATABASE语句用于创建一个新的数据库。

所需权限

只有admin角色成员才能创建新数据库。

语法图

```
CreateDatabaseStmt ::=
    'CREATE' 'DATABASE' IfNotExists DBName DatabaseOptionListOpt

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

DBName ::=
    Identifier

DatabaseOptionListOpt ::=
    DatabaseOptionList?
```

参数介绍

范围	参数
IF NOT EXISTS	仅当不存在同名数据库时才创建新数据库；如果确实存在，则不返回错误。
DBname	要创建的数据库的名称，它必须是唯一的并遵循这些标识符规则。

- 创建数据库 (判断数据库是否存在)

```
create database if not exists hubble_database;
```

- 创建数据库 (不判断数据库是否存在)

```
create database  hubble_db;
```

SQL 不会生成错误，而是会响应，CREATE DATABASE即使没有创建新数据库也是如此。

- 切换到数据库hubble_db

```
use  hubble_db;
```

- 创建多区域数据库

如果在集群启动时设置了区域，可以在集群中创建使用集群区域的多区域数据库。

使用以下命令在创建数据库时指定区域和生存目标：

```
CREATE DATABASE bank PRIMARY REGION "center1" REGIONS "center1", "center2", "
    ↪ center3" SURVIVE REGION FAILURE;
```

```
show regions from database bank;
```

```
 database | region   | primary | zones
-----------+---------+---------+--------
  bank     | center1 |  true   | {}
  bank     | center2 |  false  | {}
  bank     | center3 |  false  | {}
```

6.3.2.2.4 CREATE TABLE

该CREATE TABLE语句在数据库中创建一个新表。

所需权限

要创建表，用户必须具备以下条件之一：

- admin集群角色的成员资格。
- 数据库所有者角色的成员身份。
- 数据库的CREATE权限。

语法图

```
CreateTableStmt ::=
    'CREATE' Temporary 'TABLE' IfNotExists TableName ( CreateTableOpt  FamilyOpt
        ↪  PartitionOpt | LikeTableWithOrWithoutParen )

Temporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?


CreateTableOpt ::=
    TableOptionList?

FamilyOpt ::=
FAMILY opt_family_name

PartitionOpt ::=
```

HUBBLE

```
( 'PARTITION' 'BY' PartitionMethod Partition  PartitionDefinitionListOpt )?
```

参数介绍

参数	详情
if not exists	仅当数据库中不存在同名表时才创建新表; 如果确实存在，则不返回错误
table_name	要创建的表的名称，在其数据库中必须是唯一的，并遵循定义规则. 如果没有设置默认父数据库，则必须将名称格式设置为database.name。
TEMPORARY	将表定义为会话范围的临时表。
familyOpt	可选项，定义列族的逗号分隔列表。列族名称在表中必须是唯一的，但可以与列，约束或索引具有相同的名称。列族是一组列，它们作为单个键值对存储在基础键值存储中。Hubble 自动将列分组到列族中，以确保有效的存储和性能。但是，有时你可能希望手动将列分配给族，更多详细信息，请参见列族文档。
partition_opt	允许你在行级别定义表分区。你可以按列表或按range定义表分区。

支持范围

Hubble 支持以下列限定条件:

- 列级约束
- 排序规则
- 列族分配
- DEFAULT表达式
- 标识列 (序列填充的列)

像现有表一样创建表

支持CREATE TABLE LIKE基于现有表的模式创建新表的语法。

支持以下选项:

- 添加CHECK源表中的所有约束。
- 添加DEFAULT源表中的所有列表达式。
- 添加源表中的所有计算列表达式。
- 添加源表中的所有索引。
- 包括上述所有说明符。

> CREATE TABLE LIKE 语句不能从现有表中复制列族、分区和外键约束。如果希望在新表中使用这些列限定条件，则必须手动重新创建它们。

支持LIKE的说明符也可以与普通CREATE TABLE说明符混合使用。例如:

```
CREATE TABLE t1 (a INT PRIMARY KEY, b INT NOT NULL DEFAULT 3 CHECK (b > 0),
    ↪ INDEX(b));

CREATE TABLE t2 (LIKE table1 INCLUDING ALL EXCLUDING CONSTRAINTS, c INT, INDEX(b
    ↪ ,c));
```

在此示例中，使用t1的索引和默认值创建t2。

创建表

- 创建定义主键的表

```
create table cust_info (
        id UUID PRIMARY KEY,
        city STRING,
        name STRING,
        address STRING,
        credit_card STRING,
        dl STRING,
        index(name)
);
```

```
show index from cust_info;
```

```
 table_name |      index_name      | non_unique | seq_in_index | column_name |
    ↪ direction | storing | implicit
-------------+----------------------+------------+--------------+-------------+-----------+-
   ↪
  cust_info  | cust_info_name_idx |    true    |            1 | name        |
     ↪ ASC       |  false  |  false
  cust_info  | cust_info_name_idx |    true    |            2 | id          |
     ↪ ASC       |  false  |   true
  cust_info  | primary            |   false    |            1 | id          |
     ↪ ASC       |  false  |  false
```

- 创建不定义主键的表，在 Hubble 中，每个表都需要一个主键。如果未明确定义，则会自动添加一个INT类型的名为rowid的列作为主键，并使用unique_rowid()函数确保新行始终默认为唯一的rowid值。自动为主键创建索引。

```
create table my_table1(
    user_id int,
    cre_date date
  );
```

严格地说，主键的唯一索引并没有创建，它是由数据储存层的key来区分的，因此不需要额外的空间。但是，在使用show index等命令时，它会显示为正常的唯一索引。

```
show index from my_table1;
```

```
table_name|index_name|non_unique|seq_in_index|column_name|direction|storing|
   ↪ implicit
----------+----------+----------+------------+-----------+---------+-------+--------
   ↪
my_table1 |  primary |  false   |           1|   rowid   |ASC      | false |
   ↪ false
(1 row)
```

- 创建表并且定义主键，例如：我们创建一个包含三列的表。一列是主键，另一列是唯一约束，第三列没有约束。具有唯一约束的主键和列将自动创建索引。

```
create table my_table2 (
    user_id int primary key,
    user_email string unique,
    logoff_date date
  );
```

```
show columns from  my_table2;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |              indices                | is_hidden
--------------+-----------+-------------+----------------+-----------------------+--------
   ↪
  user_id     | INT8      |    false    | NULL           |
     ↪                      | {my_table2_user_email_key,primary} |   false
  user_email  | STRING    |    true     | NULL           |
     ↪                      | {my_table2_user_email_key}         |   false
  logoff_date | DATE      |    true     | NULL           |
     ↪                      | {}                                 |   false
```

```
show index from my_table2;
```

```
+---------+---------------------+------+---+----------+--------+-------+--------+
   ↪
| Table   |        Name         |Unique|Seq|  Column  |irection|Storing|Implicit
   ↪ |
+---------+---------------------+------+---+----------+--------+-------+--------+
   ↪
|my_table2|primary              |true  |  1|user_id   | ASC    | false | false
   ↪ |
|my_table2|logoff_user_email_key|true  |  1|user_email| ASC    | false | false
   ↪ |
|my_table2|logoff_user_email_key|true  |  2|user_id   | ASC    | false | true
   ↪ |
+---------+---------------------+------+---+----------+--------+-------+--------+
   ↪
(3 rows)
```

- 创建具有自动生成的唯一行 ID 的表

要自动生成唯一的行标识符，请使用具有UUID的列作为默认值

```
create  table user_info (
        id UUID NOT NULL DEFAULT gen_random_uuid(),
        city STRING NOT NULL,
        name STRING NULL,
        id_card STRING NULL,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
```

```
        FAMILY "primary" (id, city, name,  id_card)
);
```

生成的 ID 都是 128 位的，足够大，几乎不可能生成非唯一值。

- 创建具有外键约束的表

外键约束保证列只使用它引用的列中已经存在的值，这些值必须来自另一个表。此约束强制两个表之间的引用完整性。

```
create  table users (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        city STRING,
        name STRING,
        address STRING,
        credit_card STRING,
        dl STRING UNIQUE CHECK (LENGTH(dl) < 8)
);
```

```
create  table orders (
        id UUID NOT NULL DEFAULT gen_random_uuid(),
        city STRING NOT NULL,
        types STRING,
        order_id UUID REFERENCES users(id) ON DELETE CASCADE,
        creation_time TIMESTAMP,
        status STRING,
        current_location STRING,
        ext JSONB,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC)
);
```

- 创建带有检查约束的表

在此示例中，我们创建了`custs_info`表，但带有一些列约束。一列是主键，另一列被赋予唯一约束和限制字符串长度的检查约束。主键列和具有唯一约束的列会自动建立索引。

```
create  table custs_info (
        cust_id UUID PRIMARY KEY,
        city STRING,
        cust_name STRING,
        address STRING,
        credit_card STRING,
        dl STRING UNIQUE CHECK (LENGTH(dl) > 4)
);
```

- 创建一个映射键值存储的表

Hubble 是一个分布式 SQL 数据库，建立在事务性和强一致性键值存储之上。尽管无法直接访问键值存储，但您可以使用包含两列的简单的表来镜像直接访问，其中一组作为主键：

```
create  table kv (k INT PRIMARY KEY, v BYTES);
```

- 创建具有计算列的表

该full_name列是从first_name和last_name列计算的

```
create  table users_info (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        city STRING,
        first_name STRING,
        last_name STRING,
        full_name STRING AS (CONCAT(first_name, ' ', last_name)) STORED
);
```

6.3.2.2.5 CREATE TABLE AS

该CREATE TABLE ... AS语句选择查询创建一个新表。

语法图

```
CreateTableAsStmt ::=
'CREATE' opt_temp_table 'TABLE' 'IF' 'NOT' 'EXISTS' table_name AsSelopt
```

```
AsSelopt ::=
column_name create_as_col_qual_list (column_name create_as_col_qual_list |
    ↪ family_def | create_as_constraint_def)
```

参数介绍

参数	详情
if not exists	仅当数据库中不存在同名表时，才创建新表；如果存在，不返回错误。请注意，if not exists仅检查表名，不检查现有表是否具有新表的相同列，索引，约束等。
table_name	要创建的表的名称，与其数据库中必须是唯一的，并遵循标识符规则。如果未设置默认父数据库，则必须将名称格式设置为database.name UPSERT和INSERT ON CONFLICT语句使用名为excluded的临时表来处理执行期间的唯一性冲突。因此，建议不要使用表名excluded。
column_name	您要使用的列的名称，而不是来自的列的名称select_stmt
create_as_col_qual_list ↪	一个可选的列定义，它可能包括主键约束和列族分配
family_def	一个可选的列族定义，列族名称在表中必须是唯一的，但可以与列、约束或索引同名

> 使用 create table ... as 创建的表的主键不是从查询结果派生的；与其他表一样，创建后无法添加或更改主键；而且这些表不与其他表交错；列族的默认规则适用。

- SELECT查询创建表

例如：

```
create table logtable (
    user_id INT PRIMARY KEY,
    user_name STRING UNIQUE,
```

```
    log_date DATE NOT NULL
);

create table logtable_copy AS TABLE logtable;
```

```
show create table logtable_copy;
```

```
   table_name    |                      create_statement
----------------+-----------------------------------------------------------------
  logtable_copy | CREATE TABLE public.logtable_copy (
                |     user_id INT8 NULL,
                |     user_name STRING NULL,
                |     log_date DATE NULL,
                |     rowid INT8 NOT VISIBLE NOT NULL DEFAULT unique_rowid(),
                |     CONSTRAINT "primary" PRIMARY KEY (rowid ASC),
                |     FAMILY "primary" (user_id, user_name, log_date, rowid)
                | )
```

该示例说明主键，唯一和非空约束不会派生到新的表。

不过可以在CREATE TABLE ... AS 之后创建二级索引:

```
create index logtable_copy_id_idx ON logtable_copy ( user_id );
show create table logtable_copy;
```

```
   table_name    |                      create_statement
----------------+-----------------------------------------------------------------
  logtable_copy | CREATE TABLE public.logtable_copy (
                |     user_id INT8 NULL,
                |     user_name STRING NULL,
                |     log_date DATE NULL,
                |     rowid INT8 NOT VISIBLE NOT NULL DEFAULT unique_rowid(),
                |     CONSTRAINT "primary" PRIMARY KEY (rowid ASC),
                |     FAMILY "primary" (user_id, user_name, log_date, rowid)
                | )
```

- 更改列名

此语句创建现有表的副本，但列名已更改:

```
create table t2 (user_id, user_name) AS SELECT id, name FROM t1;
```

- 指定主键

可以指定从选择查询创建的新表的主键:

```
CREATE TABLE t2 (id, city, name PRIMARY KEY) AS SELECT id, city, name FROM t1
    ↪ WHERE city = 'shanghai';
```

- 定义列族

可以定义从选择查询创建的新表的列族:

```
CREATE TABLE t2 (id PRIMARY KEY FAMILY ids, name, city FAMILY locs, address,
    ↪ credit_card FAMILY payments) AS SELECT id, name, city, address,
    ↪ credit_card FROM t1 WHERE city = 'beijing';
```

6.3.2.2.6 CREATE INDEX

CREATE INDEX语句为表创建索引，通过帮助 SQL 定位数据而无需查看表的每一行来提高数据库的性能。

其中，PRIMARY KEY和UNIQUE列将会自动创建索引。查询表时，Hubble 使用最快的索引。

以下类型不能包含在索引键中:

- JSONB
- ARRAY
- 计算类型

所需权限

要求拥有表的CREATE权限。

语法图

```
CreateIndexStmt ::=
    'CREATE' IndexTypeOpt 'INDEX' IfNotExists Identifier IndexTypeOpt 'ON'
       ↪ TableName IndexLimit

IndexLimit  ::=
    IndexOptList

IndexTypeOpt ::=
    ( 'UNIQUE' | 'INVERTED')?

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

IndexTypeOpt ::=
    IndexType?


IndexOptList ::=
    IndexOption*

IndexType ::=
    ( 'USING' | 'TYPE' ) IndexTypeName


ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?

IndexNameList ::=
```

```
    ( Identifier | 'PRIMARY' )? ( ',' ( Identifier | 'PRIMARY' ) )*

KeyOrIndex ::=
    'Key' | 'Index'
```

参数介绍

参数	说明
UNIQUE	将UNIQUE约束应用于索引列
INVERTED	对指定列中的无模式数据创建索引，您还可以使用与 PostgreSQL 兼容的语法。
IF NOT EXISTS	仅当不存在同名索引时才创建新索引；如果确实存在，则不返回错误。
index_name	要创建的索引的名称，必须对其表唯一并遵循这些标识符规则
TableName	要在其上创建索引的表的名称

创建表

```
create table cust_info (
    id int default unique_rowid(),
    name string,
    core_id int,
    region string,
    address jsonb
);
```

创建索引

```
create index on cust_info (core_id);
```

创建多列索引

```
create index on cust_info (name, region);
```

创建唯一索引

```
create unique index on cust_info (name);
```

或者

```
alter table cust_info add constraint cust_name_id_key unique (name, id);
```

JSONB类型的字段可以创建倒排索引

```
create inverted index on cust_info (address);
```

```
show columns from cust_info;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
    ↪  |                                                                  indices
    ↪                                                                 | is_hidden
```

```
--------------+-----------+-------------+----------------+-----------------------+---------
  ↪
 id           | INT8      |    true     | unique_rowid() |
    ↪                      | {cust_name_id_key}
    ↪
    ↪ |   false
 name         | STRING    |    true     | NULL           |
    ↪                      | {cust_info_name_key,cust_info_name_region_idx,
    ↪ cust_name_id_key}                                                    |
    ↪   false
 core_id      | INT8      |    true     | NULL           |
    ↪                      | {cust_info_core_id_idx}
    ↪
    ↪ |   false
 region       | STRING    |    true     | NULL           |
    ↪                      | {cust_info_name_region_idx}
    ↪
    ↪ |   false
 address      | JSONB     |    true     | NULL           |
    ↪                      | {cust_info_address_idx}
    ↪
    ↪ |   false
 rowid        | INT8      |    false    | unique_rowid() |
    ↪                      | {cust_info_address_idx,cust_info_core_id_idx,
    ↪ cust_info_name_key,cust_info_name_region_idx,cust_name_id_key,primary} |
    ↪    true
```

- 更改列排序顺序

要按降序对列进行排序，必须在创建索引时显式设置该选项。

```
create index on cust_info (name, region desc);
```

6.3.2.2.7 CREATE VIEW

CREATE VIEW语句创建一个新视图，它是一个表示为虚拟表的存储查询。

所需权限

用户必须对父数据库具有CREATE权限，并且对视图引用的表具有SELECT权限。

语法图

```
CreateViewStmt ::=
    'CREATE' (opt_temp 'VIEW' 'IF' 'NOT' 'EXISTS'|OrReplace opt_temp 'VIEW')
       ↪ ViewName  'AS' ViewSelectStmt

OrReplace ::=
    ( 'OR' 'REPLACE' )?

ViewAlgorithm ::=
```

```
    ( MATERIALIZED )?

ViewSelectStmt  ::=
    Sel_Stmt
```

参数介绍

参数	详情
ViewName	要创建的视图的名称，该视图的名称在其数据库中必须是唯一的，并遵循标识符规则。如果未设置默认父数据库，则必须将名称格式设置为database.name。
MATERIALIZED	创建物化视图
IF NOT EXISTS	仅当不存在同名视图时才创建新视图，请注意，IF NOT EXISTS仅检查视图名称。
OR REPLACE	如果不存在同名视图，则创建一个新视图。如果已存在同名视图，请替换该视图
Sel_Stmt	在请求视图时执行的选择查询。请注意，目前无法使用*来从引用的表或视图中选择所有列，你必须指定特定的列。

建表语句

```
create table cust_info (
    id int ,
    name string,
    core_id int,
    region string
);
insert into cust_info values(100,'assir',130234,'beijing');
```

创建视图

```
create view cust_view as select id, name from cust_info;
```

查看视图

```
select * from cust_view ;
```

```
  id  | name
------+--------
  100 | assir
```

> 目前暂不支持CREATE VIEW .. AS SELECT * FROM ...，需要手动指定字段名。

查看创建视图

```
show create view cust_view;
```

```
  table_name |                                    create_statement
-------------+-------------------------------------------------------------------------------------
    ↪
```

```
  cust_view  | CREATE VIEW public.cust_view (id, name) AS SELECT id, name FROM
     ↪ ora.public.cust_info
```

6.3.2.2.8 CREATE SEQUENCE

该CREATE SEQUENCE语句在数据库中创建一个新序列，使用序列自动递增表中的整数。

所需权限

执行此操作需要用户拥有CREATE权限。

语法图

```
CreateSequenceStmt ::=
    'CREATE' Opt_Temp 'SEQUENCE' IfNotExists TableName
       ↪ CreateSequenceOptionListOpt

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

CreateSequenceOptionListOpt ::=
    SequenceOption*


SequenceOption ::=
    ( 'INCREMENT' ( '='? | 'BY' ) | 'START' ( '='? | 'WITH' ) | ( 'MINVALUE' | '
       ↪ MAXVALUE' | 'CACHE' ) '='? ) SignedNum
|   'NOMINVALUE'
|   'NO' ( 'MINVALUE' | 'MAXVALUE' | 'CACHE' | 'CYCLE' )
|   'NOMAXVALUE'
|   'NOCACHE'
|   'CYCLE'
|   'NOCYCLE'
```

参数介绍

参数	详情
seq_name	要创建的序列的名称，在其数据库中必须是唯一的，并且遵循标识符规则。当父数据库未设置为默认值时，名称的格式必须为database.seq_name
INCREMENT	序列递增的值，负数创建一个降序，正数创建升序，默认1
MINVALUE	序列的最小值，如果未指定或输入，则应用默认值NO MINVALUE。升序1，默认值：降序默认值MININT
START	序列的第一个值升序 1，默认值：降序默认值-1
NO CYCLE	目前，所有序列都设置为NO CYCLE并且序列不会换行
CACHE	要在内存中缓存以在会话中重用的序列值的数量。缓存大小为1表示没有缓存，小于缓存大小的缓存1无效。

参数	详情
OWNED BY ↪ column_name	将序列与特定列相关联，如果删除该列或其父表，则该序列也将被删除。
Opt_Temp	将序列定义为会话范围的临时序列。

序列函数

SEQUENCE支持以下函数操作:

- nextval('seq_name')
- currval('seq_name')
- lastval()
- setval('seq_name',value,is_called)

临时序列

Hubble 支持会话范围的临时序列。与持久序列不同，临时序列只能从创建它们的会话中访问，并且在会话结束时被删除。可以在持久表和临时表上创建临时序列。

细节:

- 临时序列会在会话结束时自动删除。
- 临时序列只能从创建它的会话中访问。
- 临时序列在同一会话中的事务中持续存在。
- 临时序列不能转换为持久序列。
- 创建临时序列

将TEMP/TEMPORARY添加到CREATE SEQUENCE语句中。

```
SET experimental_enable_temp_tables=on;
```

```
CREATE TEMP SEQUENCE temp_seq START 1 INCREMENT 1;
```

- 查询当前数据库中的SEQUENCES情况

```
select * from information_schema.sequences;
```

- 创建SEQUENCE

```
create sequence userid_seq;
```

```
show create userid_seq;
```

```
table_name |    create_statement
-----------+-------------------------------
userid_seq | CREATE SEQUENCE userid_seq
           | MINVALUE 1
           | MAXVALUE 9223372036854775807
           | INCREMENT 1
           | START 1
```

建表时指定主键根据sequence自动增长

```
create table user_table(
  user_id int PRIMARY KEY DEFAULT nextval('userid_seq'),
  phone_number string,
  create_date date
);
```

插入几条测试数据

```
insert into user_table (phone_number, create_date)
values
('13900010002','2018-10-01'),
('13654559807','2018-12-01'),
('18800001111','2016-10-01');
```

```
select * from user_table;
```

```
  user_id | phone_number |          cre_date
+---------+--------------+---------------------------+
        1 | 13900010002  | 2018-10-01 00:00:00+00:00
        2 | 13654559807  | 2018-12-01 00:00:00+00:00
        3 | 18800001111  | 2016-10-01 00:00:00+00:00
```

> 使用SEQUENCE比使用内置函数gen_random_uuid()、uuid_v4()、unique_rowid()自动生成的唯一 ID 要慢。

当数据被删除之后，仍然可以从删除的位置的序列值自增

```
truncate table user_table;
```

```
select * from user_table;
```

```
  user_id | phone_number | create_date
----------+--------------+--------------
(0 rows)
```

```
insert into user_table (phone_number, create_date)
                          values
                          ('13900010002','2018-10-01'),
                          ('13654559807','2018-12-01'),
                          ('18800001111','2016-10-01');
```

```
select * from user_table;
```

```
  user_id | phone_number | create_date
----------+--------------+--------------
        4 | 13900010002  | 2018-10-01
```

```
        5 | 13654559807   | 2018-12-01
        6 | 18800001111   | 2016-10-01
```

- 查看序列的当前值

要查看当前值而不增加序列，请使用：

```
select * from userid_seq;
```

- 设置序列的下一个值

在项目中，往往从导入后数据的最大值的下一个值开始新增数据，比如说现在一个表test有 1000 条数据，则从第 1001 条数据开始递增。

```
create sequence userid_seq
           MINVALUE 1
           MAXVALUE 9223372036854775807
           INCREMENT 1
           START 1001;
```

```
create table test(
  user_id int PRIMARY KEY DEFAULT nextval('userid_seq'),
  phone_number string,
  create_date date
);
```

- 使用用户定义的设置创建序列

在此示例中，我们创建了一个从1开始并以2为增量递减的序列。

```
create sequence desc_customer_list START 1 INCREMENT -2;
```

- 列出所有序列

```
show sequences;
```

- 在内存中缓存序列值

为了提高性能，使用CACHE关键字在内存中缓存序列值。

例如，在内存中缓存 5 个序列值：

```
create sequence cust_seq_id CACHE 5;
```

6.3.2.2.9 CREATE TYPE

该CREATE TYPE语句在数据库中创建一个新的枚举数据类型。

所需权限

要创建类型，用户必须具有数据库的CREATE权限。

语法图

```
CreateTypeStmt ::=
'CREATE' 'TYPE' ('IF' 'NOT' 'EXISTS')? type_name AS ENUM Enum_List

Enum_List  ::=
(opt_enum_val_list)?
```

参数介绍

参数	详情
type_name	类型的名称。您可以使用数据库和模式名称来限定名称db.typename，但是在创建类型之后，它只能从包含该类型的数据库中引用。
IF NOT EXISTS	仅当数据库中不存在同名类型时才创建新类型；如果确实存在，则不返回错误。
opt_enum_val_list	构成类型枚举集的值列表。

创建类型

```
create type status as enum ('open', 'closed', 'inactive');
```

创建与类型相关的表，并插入数据

```
create table account (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        balance DECIMAL,
        status status
);
```

```
insert into account(balance,status) values (500.50,'open'), (0.00,'closed'),
    ↪ (1.25,'inactive');
```

查看数据

```
select * from account;
```

```
                   id                   | balance |  status
----------------------------------------+---------+-----------
  3848e36d-ebd4-44c6-8925-8bf24bba957e |  500.50 | open
  60928059-ef75-47b1-81e3-25ec1fb6ff10 |    0.00 | closed
  71ae151d-99c3-4505-8e33-9cda15fce302 |    1.25 | inactive
```

查看建表结构

```
show create table account;
```

```
 table_name |                create_statement
-------------+-----------------------------------------------------
  account    | CREATE TABLE public.account (
             |     id UUID NOT NULL DEFAULT gen_random_uuid(),
             |     balance DECIMAL NULL,
```

```
            |     status public.status NULL,
            |     CONSTRAINT account_pkey PRIMARY KEY (id ASC)
            | )
```

6.3.2.2.10 DROP DATABASE

DROP DATABASE语句会删除 Hubble 中的一个数据库和数据库中的所有对象。

所需权限

用户必须拥有要删除数据库和数据库中所有表的DROP 权限。

- 删除数据库与数据库中的对象（CASCADE）

对于 client 端会话，DROP DATABASE默认使用CASCADE，会删除数据库中的所有表与视图，同时也会删除所有依赖表的对象，例如CONSTRAINT和view。

- 禁止删除一个非空的数据库（RESTRICT）

语法图

```
DropDatabaseStmt ::=
    'DROP' 'DATABASE' IfExists Name LimitConditions

IfExists ::= ( 'IF' 'EXISTS' )?

LimitConditions  ::=
    CASCADE ?
|   RESTRICT
```

参数介绍

参数	详情
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
NAME	您要删除的数据库的名称。如果将数据库设置为当前数据库或如果sql_safe_updates = true
CASCADE	删除数据库中的所有表和视图以及依赖于这些表的所有对象（例如约束和视图）
RESTRICT	如果数据库包含任何表或视图，请不要删除它。

- 防止删除非空数据库

当一个数据库不是空的时候，RESTRICT会阻止数据库的删除。

```
drop  database ora restrict;
```

```
ERROR: database "ora" is not empty and RESTRICT was specified
```

- 删除数据库及其对象

对于非交互式会话 (例如，客户端应用程序)，默认DROP DATABASE应用该选项，这将删除数据库中的所有表和视图以及依赖于这些表的所有对象 (例如约束和视图)。

```
drop  database ora CASCADE;
```

6.3.2.2.11 DROP TABLE

从数据库中删除一张表，包括表的数据与索引。

所需权限

用户必须对要删除的表拥有drop权限。

语法图

```
DropTableStmt ::=
    'DROP'  'Table' IfExists TableName ('CASCADE'?|'RESTRICT')

IfExists  ::=
    'IF' 'EXISTS'
```

参数介绍

参数	详情
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
TableName	以逗号分隔的表名列表。要查找表名，请使用SHOW TABLES
CASCADE	删除数据库中的所有表和视图以及依赖于这些表的所有对象（例如约束和视图）
RESTRICT	如果数据库包含任何表或视图，请不要删除它。

展示当前数据库的表

```
show tables;
```

```
  schema_name |          table_name          | type  | owner |
    ↪ estimated_row_count | locality
--------------+------------------------------+-------+-------+---------------------+-----
  ↪
  public      | a                            | table | root  |
    ↪                   1 | NULL
  public      | aa                           | table | root  |
    ↪                   1 | NULL
  public      | b                            | table | root  |
    ↪                   1 | NULL
  public      | customers                    | table | root  |
    ↪                   2 | NULL
  public      | orders                       | table | root  |
    ↪                   0 | NULL
```

删除表示例（无依赖关系）

```
drop table b;
```

删除表示例（有依赖关系）

```
create table customers (
      id int primary key,
```

```
      name string,
      email string
  );

create table shipments (
      tracking_number uuid default gen_random_uuid() primary key,
      carrier string,
      status string,
      customer_id int
    );

alter table shipments add constraint fk_customers foreign key (customer_id)
   ↪ references customers(id);
```

```
drop table customers ;
```

```
"customers" is referenced by foreign key from table "shipments"
```

此时，我们要用到CASCADE强制删除

```
drop table customers CASCADE;
```

> CASCADE删除依赖于表的所有对象（例如外键约束和视图），为强制性删除语句，且CASCADE不会列出它掉落的对象，因此应谨慎使用。

6.3.2.2.12 DROP COLUMN

DROP COLUMN 语句是ALTER TABLE的一部分，会删除一张表中某些列。

所需权限

用户必须拥有CREATE权限。

语法图

```
DropColumnStmt
         ::= 'ALTER'  'TABLE' TableName DropColumnSpec

DropColumnSpec
         ::= 'DROP' 'COLUMN'? 'IF EXISTS'? ColumnName ( 'RESTRICT' | 'CASCADE' )
            ↪ ?
```

参数介绍

参数	详情
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
TableName	以逗号分隔的表名列表。要查找表名，请使用SHOW TABLES
ColumnName	要删除的列的名称。当删除带有CHECK约束的列时，该CHECK约束也将被删除
CASCADE	删除数据库中的所有表和视图以及依赖于这些表的所有对象（例如约束和视图）
RESTRICT	如果数据库包含任何表或视图，请不要删除它。

当我们有变更表结构时，Hubble 会将这个操作注册成一个job，可以通过SHOW JOBS查看。

删除列示例（表中有数据）

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );

insert into cust_info values('14435550','王吉','12022519960321531X
    ↪ ',15122511874,'天津武清','抵押');
insert into cust_info values('14435551','张贺
    ↪ ','431256197306265320',15534343555,'山西临汾','质押');
insert into cust_info values('14435552','刘明
    ↪ ','371452199303034312',18967756743,'陕西延安','信用');
insert into cust_info values('14435553','李华','52112119860621421X
    ↪ ',15833355455,'湖北武汉','抵押');
insert into cust_info values('14435554','郑青
    ↪ ','213456199102275341',13054546567,'江西南昌','质押');
```

- 删除cust_name

```
alter table cust_info drop column cust_name;
```

```
pq: rejected: ALTER TABLE DROP COLUMN will remove all data in that column (
    ↪ sql_safe_updates = true)
```

- 先将sql_safe_updates会话变量必须设置为false，在执行删除列语句

```
set sql_safe_updates=false;
alter table cust_info drop column cust_name;
```

```
set sql_safe_updates=true;
```

> 注意：删除完成后请还原sql_safe_updates为true

- 删除时有依赖：

```
create view cust_view as select cust_no,cust_card_no from cust_info;
```

```
alter table cust_info drop column cust_card_no RESTRICT;
```

```
cannot drop column "cust_card_no" because view "cust_view" depends on it
```

- 删除有对象依赖列（CASCADE）

如果要删除有对象依赖的列，使用CASCADE

```
alter table cust_info drop column cust_card_no CASCADE;
```

> 不要忘记删除完成后，还原sql_safe_updates为true

6.3.2.2.13 DROP INDEX

DROP INDEX语句会删除表中的索引。

所需权限

用户必须拥有表的CREATE权限。

语法图

```
DropIndexStmt ::=
    'DROP' 'INDEX' CONCURRENTLY IfExists  TableName  index_name ( 'RESTRICT' | '
       ↪ CASCADE' )?


IfExists ::=
    ( 'IF' 'EXISTS' )?
```

参数介绍

参数	详情
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
TableName	以逗号分隔的表名列表。要查找表名，请使用SHOW TABLES
CASCADE	删除数据库中的所有表和视图以及依赖于这些表的所有对象（例如约束和视图）
RESTRICT	如果数据库包含任何表或视图，请不要删除它。
index_name	要删除的索引的名称。使用查找索引名称SHOW INDEX
CONCURRENTLY	用于 PostgreSQL 兼容性的可选无操作语法。

- 建表并查看索引

```
create table shipments (
      tracking_number uuid default gen_random_uuid() primary key,
      carrier string,
      status string,
      customer_id int,
      index(customer_id)
    );

create table customers (
      id int primary key,
      name string,
      email string,
```

```
      index(name)
  );

alter table shipments add constraint fk_customers foreign key (customer_id)
   ↪ references customers(id);

show index from customers;
```

```
  table_name |      index_name      | non_unique | seq_in_index | column_name |
     ↪ direction | storing | implicit
-------------+--------------------+------------+--------------+-------------+-----------+-
   ↪
  customers  | customers_name_idx |    true    |            1 | name        |
     ↪ ASC       |  false  |  false
  customers  | customers_name_idx |    true    |            2 | id          |
     ↪ ASC       |  false  |   true
  customers  | primary            |   false    |            1 | id          |
     ↪ ASC       |  false  |  false
```

- 删除索引

格式：DROP INDEX table_name@index_name ，默认生成的索引会有表名作为前缀，如果自定义的索引名，则需要按此格式

```
drop  index customers@customers_name_idx;
```

- 如果一个表字段是外键类型时，删除索引时要加关键字CASCADE。

```
drop  index shipments@shipments_customer_id_idx CASCADE;
```

6.3.2.2.14 DROP VIEW

DROP VIEW语句删除数据库中的视图。

所需权限

用户必须拥有删除视图的权限。

语法图

```
DropViewStmt ::=
    'DROP' 'VIEW' ( 'IF' 'EXISTS' )? TableName CheckOpt

CheckOpt ::=
   CASCADE ?
|   RESTRICT
```

参数介绍

HUBBLE

参数	详情
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
TableName	以逗号分隔的表名列表。要查找表名，请使用SHOW TABLES
CASCADE	删除数据库中的所有表和视图以及依赖于这些表的所有对象（例如约束和视图）
RESTRICT	如果数据库包含任何表或视图，请不要删除它。

- 创建表并创建视图

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );

create view user_info
  as select  cust_no, cust_name,cust_card_no
  from  cust_info;
```

- 查看视图

```
SELECT * FROM information_schema.tables WHERE table_type = 'VIEW';
```

```
  table_catalog | table_schema | table_name | table_type | is_insertable_into |
     ↪ version
----------------+--------------+------------+------------+--------------------+----------
   ↪
  ora            | public       | user_info  | VIEW       | NO                 |
     ↪        1
```

- 删除视图 (无依赖)

```
drop view user_info;
```

- 强制删除（CASCADE）

> **警告**:
>
> CASCADE删除所有依赖视图而不列出它们，这可能会导致意外和难以恢复的损失。

```
drop view user_info CASCADE;
```

6.3.2.2.15 DROP SEQUENCE

删除数据库中的序列。

所需权限

用户必须拥有要删除序列的DROP权限。

语法图

```
DropSequenceStmt ::=
    'DROP' 'SEQUENCE' IfExists TableNameList ('CASCADE'?|'RESTRICT')

IfExists ::= ( 'IF' 'EXISTS' )?

TableNameList ::=
    TableName ( ',' TableName )*
```

参数介绍

参数	详情
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
sequence_name	您要删除的序列的名称。SHOW CREATE在使用序列的表上找到序列名称
CASCADE	删除数据库中的所有表和视图以及依赖于这些表的所有对象（例如约束和视图）
RESTRICT	如果数据库包含任何表或视图，请不要删除它。

- 删除有依赖的序列

> 如果有表依赖此序列不允许删除，且不建议用CASCADE强制删除

```
drop  sequence userid_seq;
```

```
pq: cannot drop sequence userid_seq because other objects depend on it
```

- 删除没有依赖的序列

```
create sequence cust_id;
```

```
drop  sequence cust_id;
```

6.3.2.2.16 DROP TYPE

该DROP TYPE语句从当前数据库中删除指定的枚举数据类型。

所需权限

用户必须是该类型的所有者。

```
DropTYPEStmt ::=
   'DROP' 'TYPE' ('IF' 'EXISTS')? type_name_list
```

参数介绍

参数	详情
IF EXISTS	如果存在则删除数据库；如果不存在，则不返回错误
type_name_list	要删除的类型名称或逗号分隔的类型名称列表

- 建表并创建类型

```
create type sta_type AS enum ('open');

create table cust (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        balance DECIMAL,
        status sta_type
);
```

如果有表依赖所在的类型，则无法删除，需要先删除表

```
drop type sta_type;
```

```
ERROR: cannot drop type "sta_type" because other objects ([ora.public.cust])
   ↪ still depend on it
SQLSTATE: 2BP01
```

首先删除相关的表

```
drop table accounts;
```

删除类型

```
drop type status;
```

6.3.2.2.17 DROP CONSTRAINT

DROP CONSTRAINT 语法是ALTER TABLE 的一部分，会删除列CHECK和FOREIGN KEY约束。

所需权限

用户必须拥有CREATE权限。

语法图

```
DropConstraintStmt ::=
   'ALTER' 'TABLE' IfExists table_name DropConstraint

DropConstraint     ::=
   'DROP' 'CONSTRAINT' IfExists Name ('CASCADE'?|'RESTRICT')

IfExists ::=
      ( 'IF' 'EXISTS' )?
```

参数介绍

参数	详情
table_name	具有要删除的约束的表的名称。
name	要删除的约束的名称。

- 建表语句示例：

```
create table customers (
      id int primary key,
      name string,
      email string
  );

create table shipments (
      tracking_number uuid default gen_random_uuid() primary key,
      carrier string,
      status string,
      customer_id int
    );

alter table shipments add constraint fk_customers foreign key (customer_id)
   ↪ references customers(id);
```

```
show constraints from shipments;
```

```
  table_name | constraint_name | constraint_type |                      details
     ↪                        | validated
-------------+-----------------+-----------------+-------------------------------------
   ↪
  shipments  | fk_customers    | FOREIGN KEY     | FOREIGN KEY (customer_id)
     ↪ REFERENCES customers(id) |   true
  shipments  | primary         | PRIMARY KEY     | PRIMARY KEY (tracking_number
     ↪ ASC)                   |   true
```

删除外键约束

```
alter table shipments drop constraint fk_customers;
```

```
show constraints from shipments;
```

```
  table_name | constraint_name | constraint_type |                      details
     ↪                        | validated
-------------+-----------------+-----------------+-------------------------------------
   ↪
  shipments  | primary         | PRIMARY KEY     | PRIMARY KEY (tracking_number
     ↪ ASC)                   |   true
```

- 删除唯一约束

唯一约束不允许使用ALTER TABLE [tablename] DROP CONSTRAINT语句进行删除，通过删除索引及依赖对象方式来下降唯一约束。

```
create table login_info (
    login_id       int primary key,
    customer_id    int,
    login_date     timestamp,
    unique (customer_id)
  );

show constraints from login_info;
```

```
  table_name |       constraint_name        | constraint_type |           details
     ↪           | validated
-------------+----------------------------+-----------------+----------------------------+
   ↪
  login_info | login_info_customer_id_key | UNIQUE          | UNIQUE (
     ↪ customer_id ASC)    |    true
  login_info | primary                    | PRIMARY KEY     | PRIMARY KEY (
     ↪ login_id ASC) |    true
```

```
drop  index login_info@login_info_customer_id_key cascade;
```

6.3.2.2.18 RENAME DATABASE

用于更改数据库的名称

所需权限

要重命名数据库，用户必须是admin角色的成员，并且先关闭会话变量set sql_safe_updates = false，改名完成后再次set sql_safe_updates = true。

语法图

```
RenameDatabaseStmt ::=
   'ALTER' 'DATABASE' database_name 'RENAME' 'TO' database_name
```

参数介绍

参数	详情
database_name	database_name代表的是数据库名称。

注意：如果数据库被视图引用，则无法重命名数据库

```
alter database db1 rename to db2;
```

6.3.2.2.19 RENAME TABLE

该RENAME TO语句是ALTER TABLE的一部分，用于修改表名称。

所需权限

用户必须具有表和数据库的DROP权限。用户必须同时拥有源数据库和目标数据库的CREATE权限。

语法图

```
RenameTableStmt ::=
   'ALTER' 'TABLE' ('IF' 'EXISTS')? cur_name 'RENAME' 'TO' new_name
```

参数介绍

参数	详情
IF EXISTS	仅当具有当前名称的表存在时才重命名该表
cur_name	表的当前名称
new_name	表的新名称，在其数据库中必须是唯一的并遵循这些标识符规则。

- 展示数据库的表

```
show tables;
```

```
 schema_name | table_name   | type  | owner | estimated_row_count | locality
--------------+-------------+-------+-------+---------------------+-----------
  public      | cust_info   | table | root  |                   0 | NULL
```

- 重命名表名

```
alter table  cust_info rename to cust_user_info;
```

```
show tables;
```

```
 schema_name | table_name        | type  | owner | estimated_row_count |
    ↪ locality
--------------+------------------+-------+-------+---------------------+----------
   ↪
  public      | cust_user_info   | table | root  |                   0 | NULL
```

为避免表不存在时出现错误，语句可以包括IF EXISTS:

```
alter table if exists cust_info rename to cust_user_info;
```

6.3.2.2.20 RENAME COLUMN

RENAME COLUMN语句用于修改表的字段的名称。

所需权限

要求拥有表的CREATE权限。

语法图

```
RenameColumnStmt ::=
   'ALTER' 'TABLE' ('IF' 'EXISTS')? table_name 'RENAME' 'COLUMN' name 'TO' name
```

参数介绍

参数	详情
IF EXISTS	仅当具有当前名称的表存在时才重命名该表
table_name	表的当前名称
name	表中列的名称，在其数据库中必须是唯一的并遵循这些标识符规则。

- 建表并更改列名

```
create table user_info (
    id int PRIMARY KEY,
    first_name string,
    last_name string
  );
```

```
alter  table user_info rename column last_name to name;
```

```
show create table user_info;
```

```
  table_name |                create_statement
-------------+-------------------------------------------------
  user_info  | CREATE TABLE public.user_info (
             |     id INT8 NOT NULL,
             |     first_name STRING NULL,
             |     name STRING NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (id ASC),
             |     FAMILY "primary" (id, first_name, name)
             | )
```

- 以原子方式添加和重命名列

一些子命令可以在单个ALTER TABLE语句中组合使用。例如，假设创建了一个cust包含 2 列的表：

```
create table cust (
    id INT PRIMARY KEY,
    name STRING
  );
```

要为每个用户的名字、姓氏和全名使用不同的列，因此您执行一条ALTER TABLE语句，将重命名name为last_name
↪ 和first_name连接的计算列和：

```
ALTER TABLE cust
    RENAME COLUMN name TO last_name,
    ADD COLUMN first_name STRING,
    ADD COLUMN name STRING
      AS (CONCAT(first_name, ' ', last_name)) STORED;
```

6.3.2.2.21 RENAME SEQUENCE

修改数据库的SEQUENCE名称。

所需权限

要求拥有库的CREATE权限。

语法图

```
RenameSequenceStmt ::=
   'ALTER'  'SEQUENCE' name 'RENAME' 'TO' name
```

参数介绍

参数	详情
name	表中序列的名称。

- 创建序列

```
show create user_seq;
```

- 查看当前库中的SEQUENCE:

```
select * from information_schema.sequences;
```

- 修改SEQUENCE:

```
alter sequence user_seq rename to cust_seq;
```

> 需要确认是否有表依赖于此 SEQUENCE，如果依赖不允许重命名。只有没有依赖的 SEQUENCE 才可以重命名。

6.3.2.2.22 RENAME CONSTRAINT

该RENAME CONSTRAINT语句用于修改表中约束名称。

所需权限

要求拥有表的CREATE权限。

语法图

```
RenameConstraintStmt ::=
   'ALTER' 'TABLE' ('IF' 'EXISTS')? table_name 'RENAME' 'CONSTRAINT' name 'TO'
      ↪ name
```

参数介绍

参数	详情
IF EXISTS	仅当具有当前名称的表存在时才重命名该表
table_name	表的当前名称
name	表中约束的名称。

- 建表并查看约束:

```
create table login_info (
    login_id  int primary key,
    cust_id   int,
    login_date   timestamp,
    unique (cust_id)
  );
```

```
show constraints from login_info;
```

```
 table_name |     constraint_name      | constraint_type |           details
    ↪           | validated
-------------+-------------------------+-----------------+---------------------------+----
   ↪
  login_info | login_info_cust_id_key | UNIQUE          | UNIQUE (cust_id ASC)
     ↪        |    true
  login_info | primary                 | PRIMARY KEY     | PRIMARY KEY (login_id
     ↪ ASC) |    true
```

- 修改表的约束名称:

```
alter table  login_info rename constraint login_info_cust_id_key TO unique_id;
```

```
show constraints from login_info;
```

```
  table_name | constraint_name | constraint_type |          details           |
     ↪ validated
-------------+-----------------+-----------------+----------------------------+----------
   ↪
  login_info | primary         | PRIMARY KEY     | PRIMARY KEY (login_id ASC) |
     ↪    true
  login_info | unique_id       | UNIQUE          | UNIQUE (cust_id ASC)       |
     ↪    true
```

6.3.2.2.23 SHOW DATABASES

SHOW DATABASES语句查看集群包含的所有数据库。

所需权限

用户必须被授予特定数据库的CONNECT权限才能在 Hubble 集群中列出这些数据库。

语法图

```
ShowDatabasesStmt ::=
  'SHOW'  'DATABASES' ('WITH' 'COMMENT')?
```

- 查看数据库

```
show databases;
```

或者

```
\l;
```

显示带有注释的数据库

可以使用COMMENT ON在数据库上添加注释

```
comment on database hdb is '这是一个测试库';
```

- 查看数据库的注释:

```
show databases with comment;
```

```
  database_name |                                comment
+---------------+-------------------------------------------------------------------+
  ↪
  defaultdb     | NULL
  hdb           | 这是一个测试库
  postgres      | NULL
  startrek      | NULL
  system        | NULL
```

6.3.2.2.24 SHOW TABLES

该SHOW TABLES语句列出了模式或数据库中表或视图的模式、表名、表类型、所有者和行数。

所需权限

需要有相关表的数据库的CONNECT权限。

语法图

```
ShowTablesStmt ::=
  'SHOW'  'TABLES' ('FROM' database_name)? ('WITH' 'COMMENT')?
```

参数介绍

参数	详情
database_name	要为其显示表的数据库的名称

表现

要优化SHOW TABLES语句的性能可以执行以下操作:

- 通过在执行SHOW TABLES语句之前将sql.show_tables.estimated_row_count.enabled=false来禁用表行计数显示。
- 避免SHOW TABLES在具有大量表（例如超过 10000 个表）的数据库上运行。
- 查看当前数据库的表

```
show tables;
```

或者

```
\dt;
```

- 查看指定数据库的表

```
show tables from ora;
```

- 显示带有注释的表

```
show tables from ora with comment;
```

6.3.2.2.25 SHOW COLUMNS

SHOW COLUMNS语句显示有关表中列的详细信息，包括每列的名称、类型、默认值以及它是否可以为空。

所需权限

需要对相关表的数据库的CONNECT权限。

语法图

```
ShowColumnsStmt ::=
  'SHOW'  'COLUMNS' 'FROM' table_name ('WITH' 'COMMENT')?
```

参数介绍

参数	详情
table_name	要为其显示列的表的名称

返回参数

参数	详情
column_name	列的名称
data_type	列的数据类型
is_nullable	该列是否接受NULL 可能的值：true或false
column_default	列的默认值，或计算结果为默认值的表达式。
generation_expression ↪	用于计算列的表达式
indices	列所涉及的索引列表，作为一个数组
is_hidden	列是否隐藏。可能的值：true或false

- 查看一张表中列的详细信息，包括列名、类型、默认值以及是否非空

```
show columns from  cust_info with comment;
```

```
  column_name  |  data_type  | is_nullable | column_default |
     ↪ generation_expression |                indices                | is_hidden
     ↪  |   comment
---------------+-------------+-------------+----------------+-----------------------+------
   ↪
```

```
  cust_no       | STRING       |    false     | NULL            |
    ↪                          | {cust_info_cust_card_no_idx,primary} |    false
    ↪    | 客户号
  cust_name     | VARCHAR(30) |    false     | NULL            |
    ↪                          | {}                                   |    false
    ↪    | 客户姓名
  cust_card_no | VARCHAR(18) |    true      | NULL            |
    ↪                          | {cust_info_cust_card_no_idx}         |    false
    ↪    | 客户身份证号
  cust_phoneno | DECIMAL(15) |    true      | NULL            |
    ↪                          | {}                                   |    false
    ↪    | 客户手机号
  cust_address | VARCHAR(30) |    true      | NULL            |
    ↪                          | {}                                   |    false
    ↪    | 客户所在地
  cust_type     | VARCHAR(10) |    true      | NULL            |
    ↪                          | {}                                   |    false
    ↪    | 客户授信类型
```

- 显示表中的列

```
show columns from  cust_info;
```

或者

```
\d  cust_info;
```

6.3.2.2.26 SHOW TYPES

该SHOW TYPES语句列出了当前数据库中用户定义的数据类型。

所需权限

需要对相关表的数据库的权限CONNECT。

语法图

```
ShowTypesStmt ::=
  'SHOW'  'TYPES'
```

- 列出了当前数据库中用户定义的数据类型。

```
show types;
```

```
 schema |  name   | owner
---------+--------+--------
  public | status | root
```

6.3.2.2.27 SHOW GRANTS

该SHOW GRANTS声明列出了以下内容之一：

- 授予集群中用户的角色。
- 授予用户对数据库、模式、表或用户定义类型的权限。

所需权限

查看授予用户的权限不需要任何权限。

对于SHOW GRANTS ON ROLES，用户必须具有系统表的SELECT权限

语法图

```
ShowGrantsStmt ::=
  'SHOW' 'GRANTS' 'ON' ('DATABASE' | 'SCHEMA' | 'TABLE' | 'TYPE') targets 'FOR'
     ↪ users
```

参数介绍

参数	详情
targets	以逗号分隔的数据库、模式、表或用户定义的类型名称列表
schema_name	要为其显示表的模式的名称

- 列出当前数据库及其表上所有用户和角色的所有授权

```
show grants;
```

- 显示特定用户或角色的授权

```
create user roc with password roc;
```

```
grant all on database ora to roc with grant option;
```

```
show grants for roc;
```

- 显示数据库的授权

```
show grants on database ora;
```

- 显示所有角色的所有成员

```
show grants on role;
```

- 展示具体表，所有用户和角色

```
show grants on table cust;
```

- 展示所有表、所有用户和角色

```
show grants on table *;
```

HUBBLE

6.3.2.2.28 SHOW INDEX

该SHOW INDEX语句返回表或数据库的索引信息。

所需权限

用户必须对目标表或数据库具有任何特权。

别名

在 Hubble 中，以下是SHOW INDEX的别名：

- SHOW INDEXES
- SHOW KEYS

语法图

```
ShowIndexsStmt ::=
   'SHOW' ('INDEX'|'INDEXES'|'KEYS') 'FROM' (table_name| 'DATABASE' name) ('WITH
      ↪ ' 'COMMENT')?
```

参数介绍

参数	详情
table_name	要为其显示索引的表的名称。
name	要为其显示表的数据库的名称

返回响应

参数	详情
table_name	表的名称。
index_name	索引的名称
non_unique	索引列中的值是否唯一。可能的值：true或false
seq_in_index	列在索引中的位置，以1开头
column_name	索引列
direction	列在索引中的排序方式。可能的值：ASC或DESC用于索引列；N/A对于存储的列
storing	该子句是否在索引创建STORING期间用于索引列。可能的值：true或false

- 查看表cust_info的索引信息

```
show index from cust_info;
```

- 显示数据库的索引

```
show index from database ora;
```

6.3.2.2.29 SHOW SCHEMAS

该SHOW SCHEMAS语句列出了数据库中的所有模式。

所需权限

需要数据库CONNECT权限才能列出数据库中的模式。

HUBBLE

语法图

```
ShowSchemasStmt ::=
    'SHOW' 'SCHEMAS' 'FROM' name
```

参数介绍

参数	详情
name	要为其显示架构的数据库的名称

- 查看一个数据库中所有的schemas信息

```
show schemas;
```

6.3.2.2.30 SHOW SEQUENCE

该SHOW SEQUENCES语句列出了数据库中的所有序列。

所需权限

列出数据库中的序列不需要任何特权。

语法图

```
ShowSequencesStmt ::=
    'SHOW' 'SEQUENCE' ('FROM' name)?
```

参数介绍

参数	详情
name	要为其显示架构的数据库的名称，省略时，将列出当前数据库中的序列

显示当前可用的SEQUENCE

```
show  sequences;
```

6.3.2.2.31 SHOW CONSTRAINT

该SHOW CONSTRAINTS语句列出了表上的所有命名约束。

所需权限

用户必须对目标表具有任何特权。

别名

SHOW CONSTRAINT的别名是SHOW CONSTRAINTS。

语法图

```
ShowConstraintsStmt ::=
    'SHOW' ('CONSTRAINTS'|'CONSTRAINT') 'FROM' table_name
```

参数介绍

参数	详情
table_name	要显示约束的表的名称

返回响应

参数	详情
table_name	要显示约束的表的名称
constraint_name	约束的名称
constraint_type	约束的类型
details	约束的定义，包括它适用的列
validated	列中的值是否与约束匹配

- 建表语句如下

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );
```

- 显示表的约束情况

```
show  constraint from cust_info;
```

6.3.2.2.32 SHOW PARTITIONS

使用该SHOW PARTITIONS语句查看有关现有分区的详细信息。

所需权限

列出分区不需要特权。

语法图

```
ShowPartitionsStmt ::=
    'SHOW' 'PARTITIONS' 'FROM' ('TABLE' table_name|'DATABASE' database_name|'
       ↪ INDEX' index_name)
```

参数介绍

参数	详情
database_name	要为其显示分区的数据库的名称
table_name	要显示分区的表的名称
index_name	要显示分区的索引的名称

- 分区表的建立

```
CREATE TABLE cust_info (
       cust_id INT NOT NULL,
       city VARCHAR(10) NOT NULL,
       cust_name VARCHAR(10) NULL,
       address VARCHAR(10) NULL,
       cust_card VARCHAR(10) NULL,
       CONSTRAINT "primary" PRIMARY KEY (city ASC, cust_id ASC),
       FAMILY "primary" (cust_id, city, cust_name, address, cust_card)
        ) PARTITION BY LIST (city) (
          PARTITION p1 VALUES IN (('shanghai'), ('nanjing')),
          PARTITION p2 VALUES IN (('beijing'), ('tangshan') ),
          PARTITION p3 VALUES IN (('xian'), ('xining')),
          PARTITION DEFAULT VALUES IN (default)
        );
```

- 展示分区表

```
show partitions from table cust_info;
```

- 按数据库显示分区

```
show partitions  from database ora;
```

6.3.2.2.33 SHOW DEFAULT PRIVILEGES

显示默认权限

SHOW DEFAULT PRIVILEGES语句列出了当前数据库中用户或角色创建的对象的默认权限。

所需权限

要显示默认权限，用户、角色必须对当前数据库具有任何权限。

语法图

```
ShowPrivilegesStmt ::=
    'SHOW' 'DEFAULT' 'PRIVILEGES' 'FOR' ('ROLE'|'USER'|'ALL' 'ROLES') role_list
```

参数介绍

参数	详情
FOR ROLE name/FOR ↪ USER name	列出特定用户、角色创建的对象的默认权限，或用户、角色列表
FOR ALL ROLES	列出任何用户、角色创建的对象的默认权限
role_list	角色名称

- 查看默认权限 (当前用户)

```
show default privileges;
```

- 显示由任何用户角色创建的对象的默认权限

```
show default privileges for all roles;
```

- 显示由特定用户角色创建的对象的默认权限

```
show default privileges for  role role_test;
```

- 显示特定架构中对象的默认权限

```
CREATE SCHEMA role_test;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA test GRANT SELECT ON TABLES TO role_test;
```

```
SHOW DEFAULT PRIVILEGES IN SCHEMA role_test;
```

6.3.2.2.34 SHOW ENUMS

该SHOW ENUMS语句列出了当前数据库中的枚举数据类型。

语法图

```
ShowEnumsStmt ::=
    'SHOW' 'ENUMS' ('FROM' name '.' name)?
```

参数介绍

参数	详情
name/name.name	从中显示枚举数据类型的模式的名称，或数据库和模式的名称

- 查看枚举

```
show  enums from ora.public;
```

6.3.2.2.35 SHOW RANGES

所需权限

要使用该SHOW RANGES语句，用户必须是角色的成员admin(用户默认root属于该角色) 或已定义权限。

语法图

```
ShowRangesStmt ::=
    'SHOW' 'RANGES' 'FROM' ('TABLE' table_name|'INDEX' table_index_name|'
        ↪ DATABASE' database_name)
```

参数介绍

参数	详情
table_name	您想要范围信息的表的名称
table_index_name	您想要范围信息的索引的名称
database_name	您想要范围信息的数据库的名称

查看表的range分区情况

```
show ranges from table customers_by_range;
```

6.3.2.2.36 SHOW CREATE

该SHOW CREATE语句显示CREATE现有数据库、表、视图或序列的语句。

所需权限

用户必须对目标数据库、表、视图或序列具有任何特权。

语法图

```
ShowCreateStmt ::=
    'SHOW' 'CREATE' 'FROM' (object_name|AllObj)?

AllObj ::=
    'ALL' ('TABLES'|'SCHEMAS'|'TYPES')
```

参数介绍

参数	详情
object_name	CREATE要为其显示语句的数据库、表、视图或序列的名称。
ALL TABLES	显示CREATE当前数据库中所有表、视图和序列的语句。
ALL SCHEMAS	显示当前数据库中所有模式CREATE的语句。
ALL TYPES	显示当前数据库中所有类型CREATE的语句。

- SHOW CREATE TABLE，显示创建表的语句

```
show create table <table_name>;
```

- SHOW CREATE VIEW，显示创建view的create view语句

```
show create view <viewname>;
```

- SHOW CREATE SEQUENCE，显示特定序列的创建语句

```
show create sequence <sequencesName>;
```

- 显示在当前数据库中重新创建所有表、视图和序列所需的语句

```
show create all tables;
```

- 显示CREATE DATABASE数据库的语句

```
show create database ora;
```

- 显示CREATE SCHEMA数据库中所有模式的语句

```
show  all schemas;
```

6.3.2.2.37 ALTER TABLE

子命令

HUBBLE

参数	详情
ADD COLUMN	向表中添加列
ADD CONSTRAINT	向列添加约束
ALTER COLUMN	更改现有列
DROP COLUMN	您想要范围信息的表的名称
DROP CONSTRAINT	从表中删除列
EXPERIMENTAL_AUDIT	从列中删除约束
OWNER TO	更改表的所有者
PARTITION BY	对表进行分区、重新分区或取消分区
RENAME COLUMN	更改列的名称
RENAME CONSTRAINT	更改约束列
RENAME TO	更改表的名称
SET SCHEMA	更改表的架构
SPLIT AT	强制在表中的指定行进行范围拆分
UNSPLIT AT	删除表中指定行的范围拆分强制
VALIDATE CONSTRAINT	检查列中的值是否与列上的约束匹配
SET (storage parameter)	在表上设置存储参数

6.3.2.3 DML

(Data Manipulation Language)：数据操作语言，用来操作和业务相关的记录。

6.3.2.3.1 INSERT

该INSERT语句将一行或多行插入到表中。

- 执行一个单独的多行INSERT语句比多个单行的INSERT语句更快。往一张现有表中插入大量的数据时，建议使用多行INSERT语句代替多个单行INSERT语句。
- 往一张新表写入数据时，IMPORT语句性能比INSERT语句更好。
- 在传统的 SQL 数据库中，生成和回收唯一性的 ID 需要使用INSERT和SELECT。在 Hubble 中，用RETURNING ↪ 分句和INSERT代替。

所需权限

用户必须拥有该表的INSERT权限，要使用ON CONFLICT，用户还必须具有该表的SELECT权限。要使用ON CONFLICT ↪ DO UPDATE，用户必须另外拥有该表的UPDATE权限。

语法图

```
InsertIntoStmt ::=
    'INSERT'  IntoOpt TableName 'AS' AliasName 'VALUES' InsertValues 'RETURNING'
       ↪  ReturnList

IntoOpt  ::=
    'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

AliasName ::=
```

```
    'table_alias_name'?

InsertValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '('
        ↪ SelectStmt ')' | UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   Column_Name
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*

ReturnList  ::=
    (returning target_list)?

Target_List ::=

    nothing ?
|   target_elem
```

参数介绍

参数	简介
TableName	写入数据的表
table_alias_name	表的别名
column_name	插入期间要填充的列的名称
RETURNING	根据插入的行返回值
target_elem	其中target_elem可以是表中的特定列名

示例

插入单行

```
insert into accounts (id, name, balance, flag, is_disabled) values (100, 'manus'
    ↪ , 2000.00, 'false', '1');
```

```
select * from accounts;
```

```
  id  | name  | balance | flag  | is_disabled | is_status
------+-------+---------+-------+-------------+------------
   11 | mike  | 1000.00 | false | 1           | 1
   12 | jack  | 2000.00 | true  | 0           | 1
   13 | allen | 5555.30 | false | 1           | 0
   14 | oslwn | 6321.00 | true  | 0           | 0
   15 | ding  | 6984.00 | true  | 1           | 1
   16 | mark  | 9840.50 | true  | 0           | 0
   17 | jamms | 6530.20 | false | 0           | 0
   18 | haden | 1520.00 | true  | 1           | 1
```

```
  100 | manus | 2000.00 | false | 1            | NULL
```

插入多行

```
insert into accounts (id, name, balance, flag, is_disabled) values
  (999, 'sinal', 7600.20, 'false', '0'), (19, 'feina', 5500.20, 'false', '1');
```

```
select * from accounts where id in (999, 19);
```

```
  id   | name   | balance | flag   | is_disabled | is_status
------+-------+---------+-------+-------------+------------
   19 | feina | 5500.20 | false | 1            | NULL
  999 | sinal | 7600.20 | false | 0            | NULL
```

ON CONFLICT更新值

当遇到唯一的冲突时，Hubble 将该行存储在称为excluded的临时表中。以下示例演示在发生冲突时，如何使用临时表excluded中的列来应用更新。

```
insert into accounts (id, name, balance, flag, is_disabled)
  values (999, 'sinal', 7600.20, 'false', '0')
  on conflict (id)
  do update set id =excluded.id;
```

使用现有值更新

```
insert into  accounts (id, name,balance, flag, is_disabled)
  values (999, 'sinal', 7600.20, 'false', '0')
  on conflict (id)
  do update set balance=accounts.balance+excluded.balance;
```

```
select * from accounts where id=999;
```

```
  id   | name   | balance | flag   | is_disabled | is_status
------+-------+---------+-------+-------------+------------
  999 | sinal | 7600.20 | false | 0            | NULL
```

ON CONFLICT语句下使用WHERE条件

```
insert into accounts (id, name,balance, flag, is_disabled)
  values (999, 'sinal', 7600.20, 'false', '0')
  on conflict (id)
  do update set balance=excluded.balance
  where excluded.balance < accounts.balance;
```

```
select * from  accounts where id=999;
```

```
  id   | name   | balance | flag   | is_disabled | is_status
------+-------+---------+-------+-------------+------------
  999 | sinal | 7600.20 | false | 0            | NULL
```

ON CONFLICT不更新值

- 多行插入时，不冲突的值可被插入，冲突的不被执行

```
insert into accounts (id, name, balance, flag, is_disabled )
    values (321, 'alian', 76, 'false', '1'), (999, 'sinal', 7600.20, 'false', '0
        ↪ ')
    on conflict (id)
    do nothing;
```

性能最佳实践

现有表

表已经存在的条件下

- 在事务中的一个语句中执行多行INSERT。
- 不要使用 100000 行或更多行的大批量，这可能会导致长时间运行的事务，从而导致事务重试错误。如果多行INSERT导致40001带有消息"transaction deadline exceeded" 的错误代码，Hubble 建议将INSERT分成更小的批次。
- 通过监控隐式事务中不同批处理大小（1、10、100、1000）行的性能，通过实验确定应用程序的最佳批处理大小。
- 可以使用该IMPORT INTO语句批量插入CSV数据。

新表

表不存在

Hubble 建议您使用该IMPORT语句，因为它的性能优于INSERT。

6.3.2.3.2 DELETE

DELETE语句删除一个表中行的数据

- 如果你要删除由foreign key constraint引用的行，而且有on delete action，所有依赖这行数据的行将被删除或更新。

所需权限

要执行DELETE操作，用户必须拥有DELETE与SELECT权限

语法图

```
DeleteFromStmt ::=
    'DELETE'  ( 'FROM' ( TableName TableAsName  WhereClause OrderByOpt  |
        ↪ TableList 'USING' TableRefs WhereClauseOptional ) | TableAliasRefList
        ↪ 'FROM' TableRefs WhereClauseOptional )  ReturnList

WhereClause ::=
     (where a_expr)?

OrderByOpt  ::=
     (order by a_expr)?

ReturnList  ::=
    (returning target_list)?
```

```
Target_List ::=

    nothing ?
|   target_elem
```

参数介绍

参数	简介
TableName	包含要更新的行的表的名称
TableAName	包含要更新的行的表的别名
a_expr	要使用的值或要使用的标量表达式
RETURNING target_list	根据删除的行返回值，其中target_list可以是表中的特定列名、所有列或使用标量表达式的计算

删除后的磁盘空间使用情况

删除一行不会立即释放磁盘空间。这是因为 Hubble 保留了历史查询表的能力。

如果磁盘使用是一个问题，解决方案是通过设置gc.ttlseconds为较低的值来减少区域的生存时间（TTL），这将导致垃圾收集更频繁地清理已删除的对象（行、表）。

选择已删除行的性能

扫描具有大量已删除行的表的查询将不得不扫描尚未被垃圾收集的删除。某些频繁扫描并删除大量行的数据库使用模式将希望减少生存时间值以更频繁地清理已删除的行。

对删除的输出进行排序

要对DELETE语句的输出进行排序，请使用：

```
WITH a AS (DELETE ... RETURNING ...)
  SELECT ... FROM a ORDER BY ...
```

示例

返回删除数据，为了知道删除了哪些数据，可以使用RETURNING字段取回已删除的数据。

- 返回所有列数据

```
delete from accounts where id < 12 returning * ;
```

```
  id | name | balance | flag  | is_disabled | is_status
-----+------+---------+-------+-------------+------------
  11 | mike | 1000.00 | false | 1           | 1
```

- 返回特定列的数据

```
delete from accounts where id < 13 returning id,name;
```

```
  id | name
-----+-------
  12 | jack
```

- 使用主键或者唯一列删除行

```
delete from accounts where id = 7369;
```

- 使用非唯一列删除行

```
delete from accounts where id < 1000;
```

6.3.2.3.3 UPDATE

UPDATE语句用于更新表中的数据。

所需权限

用户需要拥有表的SELECT和UPDATE权限。

语法图

```
UpdateTableStmt ::=
    'UPDATE'  TableRef  'SET' column_name  SelectOpt WhereClause LimitClause
       ↪ ReturnList

TableRef       ::=
     tableName

SelectOpt      ::=
     select_stmt ?
|    a_expr

WhereClause    ::=
    ( WHERE a_expr)?

ReturnList  ::=
    (returning target_list)?

Target_List ::=

    nothing ?
|   target_elem
```

参数介绍

参数	简介
TableName	包含要更新的行的表的名称
column_name	要更新其值的列的名称
a_expr	要使用的新值、要执行的聚合函数或要使用的标量表达式

参数	简介
select_stmt	选择查询，每个值必须与其左侧列的数据类型=相匹配
WHERE a_expr	a_expr必须是使用列，例如: 返回布尔值的标量表达式<column> = <value>
limit_clause	一个LIMIT子句。有关更多详细信息，请参阅限制查询结果。
RETURNING target_list	根据更新的行返回值，其中target_list可以是表中的特定列名、所有列或使用标量表达式的计算

示例

- 根据单个字段进行更新

```
update accounts set balance=9856 where id =19;
```

```
select * from accounts where id=19;
```

```
  id | name   | balance | flag   | is_disabled | is_status
-----+-------+---------+-------+-------------+------------
  19 | feina | 9856.00 | false | 1           | NULL
```

- 根据多个字段进行更新

```
update accounts set balance=9856,name='anna' where id =10;
```

- 使用SELECT语句更新

```
update accounts
  set (name, balance)=
  (select name, balance from accounts where  id=100)
  where id=18;
```

```
select * from accounts where id=18;
```

```
  id | name   | balance | flag | is_disabled | is_status
-----+-------+---------+------+-------------+------------
  18 | manus | 2000.00 | true | 1           | 1
```

- 更新后返回值

```
update  accounts set balance=9986.8 where  name='jamms' returning id,name,
    ↪ balance;
```

```
  id | name   | balance
-----+-------+----------
  17 | jamms | 9986.80
```

在更新中引用其他表

要引用正在更新的表以外的表中的值，请添加一个FROM子句，指定一个或多个表。子句中指定的表中的值FROM可用于UPDATE表达式和WHERE子句中。

当使用子句执行UPDATE时，Hubble 将目标表（即正在更新的表）连接到子句中引用的表。此连接的输出应该与目标表中正在更新的行具有相同的行数，因为 Hubble 使用连接输出中的单行来更新目标表中的给定行。如果连接产生的行多于目标表中正在更新的行，则无法预测连接输出中的哪一行将用于更新目标表中的行。

- 使用来自不同表的值进行更新

```
update cust c set name ='mike' from cust_info ci where c.id=ci.id ;
```

批量更新数据

要更新大量行 (即数万行或更多行)，我们建议迭代更新要更新的行的子集，直到所有行都已更新。可以编写一个脚本来执行此操作，或者您可以在您的应用程序中编写一个循环。

6.3.2.3.4 TRUNCATE

TRUNCATE语句用于从表中删除所有行。在高层次上，它通过删除表并重新创建具有相同名称的新表来工作，可认为TRUNCATE 语句同DROP TABLE + CREATE TABLE 组合在语义上相同。

TRUNCATE TABLE tableName 和 TRUNCATE tableName 均为有效语法

所需权限

用户需要拥有表的DROP权限

语法图

```
TruncateTableStmt ::=
  'TRUNCATE' OptTable  TableName  Constrints

OptTable  ::=
  'Table'  ?

Constrints ::=
  CASCADE  ?
 |RESTRICT
```

参数介绍

参数	简介
TableName	包含要更截断的表的名称
CASCADE	在被截断的表上截断所有具有外键依赖项的表
RESTRICT	默认）如果任何其他表具有外键依赖项，则不要截断该表

清空一张客户表

```
truncate dates;
```

等同于

```
truncate table dates;
```

表与emp表具有外键关系dept。因此，只能在截断dept表的同时截断从属emp表

对于含有外键的表，需要使用CASCADE关键字删除数据

```
truncate dept ;
```

```
pq: "dept" is referenced by foreign key from table "emp"
```

引入CASCADE关键字，进行清表

```
truncate dept cascade;
```

6.3.2.3.5 UPSERT

UPSERT语句在语义上等效于INSERT ON CONFLICT，但是两者的性能特征可能略有不同。该UPSERT语句在指定值不违反唯一性约束的情况下插入行，并在值确实违反唯一性约束的情况下更新行。

同INSERT类似，单个多行UPSERT语句比多个单行UPSERT语句快。尽可能使用多行UPSERT而不是多个单行UPSERT
↪ 语句。

所需权限

用户需要拥有表的 INSERT ， SELECT 和 UPDATE 权限。

语法图

```
UpsertIntoStmt ::=
    'UPSERT'  IntoOpt TableName 'AS' AliasName NameList UpdateValues ReturnList

IntoOpt  ::=
    'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

AliasName ::=
    'table_alias_name'?

NameList ::=
    column_name ( '.' column_name )?

UpdateValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '('
       ↪ SelectStmt ')' | UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   DEFAULT VALUES
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*

ReturnList  ::=
    (returning target_list)?

Target_List ::=
```

```
    nothing ?
|   target_elem
```

参数介绍

参数	简介
TableName	写入数据的表
table_alias_name	表的别名
column_name	插入期间要填充的列的名称
SelectStmt	选择查询，每个值必须与其列的数据类型匹配。
DEFAULT VALUES	要使用默认值填充所有列
RETURNING	根据插入的行返回值
target_elem	其中target_elem可以是表中的特定列名

示例数据：

```
create table a (id int primary key, name string);
insert into  a  values (1,'zhangsan'),(2,'mike');
```

id 列是主键，因为插入的 id 值不与 id 任何现有行的值冲突，所以该UPSERT语句将新行插入到表中，如下语句：

```
upsert into a (id, name) values (3, 'lisi');
```

```
select  * from a;
```

```
 id |   name
-----+-----------
   1 | zhangsan
   2 | mike
   3 | lisi
```

若插入的 id 值不是唯一的，所以该UPSERT语句使name更新，语句如下：

```
upsert into a (id, name) values (3, 'liuli');
```

```
select * from a;
```

```
  id |   name
-----+-----------
   1 | zhangsan
   2 | mike
   3 | liuli
```

当唯一性冲突不在主键中的列上时，UPSERT将不会更新。

下表中id列是主键，但是aml列具有唯一约束，UPSERT将不会生效。

```
create table b (id int primary key, aml int, unique (aml));
insert into  b  values (1,1000),(2,2000);
```

```
upsert into  b (id, aml) values (3, 1000);
```

```
duplicate key value violates unique constraint "b_aml_key"
```

由于aml字段创建了唯一性约束，所以更新失败

> 如果非主键字段有唯一约束，不能使用UPSERT INTO语法，使用INSERT INTO ... ON CONFLICT ... DO UPDATE ...语句进行操作。

6.3.2.4 DQL

数据查询语言，用来查询经过条件筛选的记录。

6.3.2.4.1 SELECT

SELECT语句用于读取和处理现有数据。

选择查询读取和处理 Hubble 中的数据。它们比简单的SELECT子句更通用：它们可以使用集合操作对一个或多个选择子句进行分组，并且可以请求特定的排序或行限制。

可能会发生选择查询：

- 括号之间作为子查询。
- 作为将表格数据作为输入的其他语句的操作数，例如INSERT、UPSERT、CREATE TABLE AS等。

所需权限

要求用户拥有表的SELECT权限。

语法图

```
SelectStmt ::=
    SelectStmtFromTable ?
|   SelectStmtFromDualTable

SelectConditionStmt ::=

OrderByOptional SelectStmtLimit SelectLockOpt

FromDual   ::=
    'FROM' 'DUAL'
```

```
WhereClauseOptional  ::=
    WhereClause  ?

SelectStmtGroup      ::=
    GroupByClause  ?
```

```
HavingClause   ::=
    'HAVING' Expression ?

OrderByOptional  ::=

     OrderBy ?

SelectStmtLimit  ::=
('LIMIT' LimitOption | 'LIMIT' LimitOption 'OFFSET' LimitOption )?

SelectLockOpt    ::=
    'FOR' 'UPDATE'  'OF' TableList ?

TableList ::=
    TableName ( ',' TableName )*

WindowClauseOptional  ::=
    WindowDefinition ?
```

参数介绍

参数	简介
DISTINCT	指定时DISTINCT，将消除重复行
ALL	默认情况下或ALL指定时，返回所有选定的行
ORDER BY	用于排序的函数，一般情况下默认升序
GROUP BY	在一列或多列上对结果进行分组
HAVING	它必须是使用聚合函数
windowDefinition	窗口定义列表
limit	用于分页的关键字，后面跟数字

条款

TABLE 条款

```
TableStmt ::=
   'TABLE'  tab_ref
```

子句从指定的TABLE表中读取表格数据。

TABLE x相当于SELECT * FROM x。

SELECT 条款

有关详细信息，请参阅下文select示例语句。

查询示例

- 建表并插入数据

```
create table accounts (
id int primary key,
```

```
name varchar(20) not null,
balance  DECIMAL(10,2),
flag string,
is_disabled string,
is_status string,
index(name)
);

insert into accounts values (11,'mike',1000,'false','1','1');
insert into accounts values (12,'jack',2000,'true','0','1');
insert into accounts values (13,'allen',5555.3,'false','1','0');
insert into accounts values (14,'oslwn',6321,'true','0','0');
insert into accounts values (15,'ding',6984,'true','1','1');
insert into accounts values (16,'mark',9840.5,'true','0','0');
insert into accounts values (17,'jamms',6530.2,'false','0','0');
insert into accounts values (18,'haden',1520,'true','1','1');
```

- 展示建表语句

```
show create table accounts;
```

```
  table_name |                              create_statement
-------------+-------------------------------------------------------------------------------
  ↪
  accounts   | CREATE TABLE public.accounts (
             |     id INT8 NOT NULL,
             |     name VARCHAR(20) NULL,
             |     balance DECIMAL(10,2) NULL,
             |     flag STRING NULL,
             |     is_disabled STRING NULL,
             |     is_status STRING NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (id ASC),
             |     FAMILY "primary" (id, name, balance, flag, is_disabled,
               ↪ is_status)
```

- 查询所有列

```
select * from accounts;
```

```
  id | name   | balance | flag  | is_disabled | is_status
-----+-------+---------+-------+-------------+------------
  11 | mike   | 1000.00 | false | 1           | 1
  12 | jack   | 2000.00 | true  | 0           | 1
  13 | allen  | 5555.30 | false | 1           | 0
  14 | oslwn  | 6321.00 | true  | 0           | 0
  15 | ding   | 6984.00 | true  | 1           | 1
  16 | mark   | 9840.50 | true  | 0           | 0
  17 | jamms  | 6530.20 | false | 0           | 0
```

```
  18 | haden | 1520.00 | true  | 1            | 1
```

- 查询特定列

```
select name from accounts;
```

```
  name
---------
  mike
  jack
  allen
  oslwn
  ding
  mark
  jamms
  haden
```

- 单个条件过滤

```
select * from accounts where name='mark';
```

```
  id | name | balance | flag | is_disabled | is_status
-----+------+---------+------+-------------+------------
  16 | mark | 9840.50 | true | 0           | 0
```

- 多个条件过滤

```
select * from accounts where name='mark' or is_disabled='0';
```

```
  id | name  | balance | flag  | is_disabled | is_status
-----+-------+---------+-------+-------------+------------
  12 | jack  | 2000.00 | true  | 0           | 1
  14 | oslwn | 6321.00 | true  | 0           | 0
  16 | mark  | 9840.50 | true  | 0           | 0
  17 | jamms | 6530.20 | false | 0           | 0
```

- 列表过滤值

```
select name from accounts where balance in ('1000', '6530.20');
```

```
   id | name  | balance | flag  | is_disabled | is_status
-----+-------+---------+-------+-------------+------------
  11 | mike  | 1000.00 | false | 1           | 1
  17 | jamms | 6530.20 | false | 0           | 0
```

- DISTINCT查询

```
select distinct name from accounts;
```

```
  name
---------
  mike
  jack
  allen
  oslwn
  ding
  mark
  jamms
  haden
```

- ALL查询 (默认情况下也是返回所有)

```
select all * from accounts;
```

```
  id | name  | balance | flag  | is_disabled | is_status
-----+-------+---------+-------+-------------+------------
  11 | mike  | 1000.00 | false | 1           | 1
  12 | jack  | 2000.00 | true  | 0           | 1
  13 | allen | 5555.30 | false | 1           | 0
  14 | oslwn | 6321.00 | true  | 0           | 0
  15 | ding  | 6984.00 | true  | 1           | 1
  16 | mark  | 9840.50 | true  | 0           | 0
  17 | jamms | 6530.20 | false | 0           | 0
  18 | haden | 1520.00 | true  | 1           | 1
```

- GROUP BY聚合函数查询

```
select name, sum(balance) from accounts group by name;
```

```
  name  |   sum
--------+----------
  mike  | 1000.00
  jack  | 2000.00
  allen | 5555.30
  oslwn | 6321.00
  ding  | 6984.00
  mark  | 9840.50
  jamms | 6530.20
  haden | 1520.00
```

- HAVING分组过滤查询 (通常用于分组后数据的过滤)

```
select is_status, sum(balance) from accounts group by is_status having is_status
    ↪ ='0';
```

```
  is_status |   sum
------------+-----------
```

```
  0          | 28247.00
```

- 分页查询

```
select * from accounts offset 0 limit 3;
```

```
  id | name   | balance | flag   | is_disabled | is_status
-----+-------+---------+-------+-------------+------------
  11 | mike   | 1000.00 | false | 1           | 1
  12 | jack   | 2000.00 | true  | 0           | 1
  13 | allen  | 5555.30 | false | 1           | 0
```

- 重命名输出的列，用AS

```
select id as cust_id,name as cust_name from accounts limit 3;
```

```
  cust_id | cust_name
----------+------------
       11 | mike
       12 | jack
       13 | allen
```

使用搜索列中的部分字符串匹配LIKE，它支持以下通配符运算符：

- % 匹配 0 个或多个字符。
- _ 恰好匹配 1 个字符。

```
select * from accounts where name like 'm%';
```

```
  id | name | balance | flag   | is_disabled | is_status
-----+------+---------+-------+-------------+------------
  11 | mike | 1000.00 | false | 1           | 1
  16 | mark | 9840.50 | true  | 0           | 0
(2 rows)
```

- 按别名分组，给子便标签增加AS

```
select name as cust_name,sum(balance) as aml from accounts group by cust_name;
```

```
  cust_name |    aml
------------+----------
  mike      | 1000.00
  jack      | 2000.00
  allen     | 5555.30
  oslwn     | 6321.00
  ding      | 6984.00
  mark      | 9840.50
  jamms     | 6530.20
  haden     | 1520.00
```

集合运算符

SQL 允许您比较多个选择子句的结果。您可以将每个集合运算符视为表示布尔运算符:

- UNION=OR
- INTERSECT=AND
- EXCEPT=NOT

默认情况下，这些比较中的每一个仅显示每个值的一个副本（类似于SELECT DISTINCT）。

建表及其数据准备:

```
create table cust_infobak(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );

insert into cust_infobak values('14435551','张贺','431256197306265320'
    ↪ ,15534343555,'山西临汾','质押');
insert into cust_infobak values('14435552','刘明','371452199303034312'
    ↪ ,18967756743,'陕西延安','信用');
insert into cust_infobak values('14435553','李华','52112119860621421X'
    ↪ ,15833355455,'湖北武汉','抵押');
insert into cust_infobak values('14435554','郑青','213456199102275341'
    ↪ ,13054546567,'江西南昌','质押');
```

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );


insert into cust_info values('14435550','王吉','12022519960321531X',15122511874,
    ↪ '天津武清','抵押');
insert into cust_info values('14435551','张贺','431256197306265320',15534343555,
    ↪ '山西临汾','质押');
insert into cust_info values('14435552','刘明','371452199303034312',18967756743,
    ↪ '陕西延安','信用');
```

UNION: 合并两个查询

UNION将两个查询的结果合并为一个结果。

```
select * from cust_info
union
select * from cust_infobak;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435551 | 张 贺                    | 431256197306265320 |  15534343555 | 山 西 临
     ↪ 汾                        | 质 押
  14435552 | 刘 明                    | 371452199303034312 |  18967756743 | 陕 西 延
     ↪ 安                        | 信 用
  14435550 | 王 吉                    | 12022519960321531X |  15122511874 | 天 津 武
     ↪ 清                        | 抵 押
  14435553 | 李 华                    | 52112119860621421X |  15833355455 | 湖 北 武
     ↪ 汉                        | 抵 押
  14435554 | 郑 青                    | 213456199102275341 |  13054546567 | 江 西 南
     ↪ 昌                        | 质 押
```

要显示重复的行，您可以使用UNION ALL

```
select * from cust_info
union all
select * from cust_infobak;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435550 | 王 吉                    | 12022519960321531X |  15122511874 | 天 津
     ↪ 武 清                        | 抵 押
  14435551 | 张 贺                    | 431256197306265320 |  15534343555 | 山 西
     ↪ 临 汾                        | 质 押
  14435552 | 刘 明                    | 371452199303034312 |  18967756743 | 陕 西
     ↪ 延 安                        | 信 用
  14435551 | 张 贺                    | 431256197306265320 |  15534343555 | 山 西
     ↪ 临 汾                        | 质 押
  14435552 | 刘 明                    | 371452199303034312 |  18967756743 | 陕 西
     ↪ 延 安                        | 信 用
  14435553 | 李 华                    | 52112119860621421X |  15833355455 | 湖 北
     ↪ 武 汉                        | 抵 押
  14435554 | 郑 青                    | 213456199102275341 |  13054546567 | 江 西
     ↪ 南 昌                        | 质 押
```

INTERSECT: 检索两个查询的交集

INTERSECT仅选择两个查询操作数中都存在的值。

```
select * from cust_info
intersect
select * from cust_infobak;
```

```
  cust_no   | cust_name |     cust_card_no     | cust_phoneno | cust_address |
     ↪ cust_type
-----------+-----------+----------------------+--------------+--------------+-------------
   ↪
  14435551 | 张 贺                      | 431256197306265320 |  15534343555 | 山 西
     ↪ 临 汾                        | 质 押
  14435552 | 刘 明                      | 371452199303034312 |  18967756743 | 陕 西
     ↪ 延 安                        | 信 用
```

EXCEPT：从另一个查询中排除一个查询的结果

EXCEPT选择存在于第一个查询操作数中但不存在于第二个查询操作数中的值。

```
select cust_no from cust_info
except
select cust_no from cust_infobak;
```

```
  cust_no
------------
  14435550
```

6.3.2.4.2 SELECT FOR UPDATE

用于并发控制的行级锁定SELECT FOR UPDATE

该SELECT FOR UPDATE语句用于通过控制对表的一行或多行的并发访问来对事务进行排序。

它通过锁定选择查询返回的行来工作，这样试图访问这些行的其他事务被迫等待锁定这些行的事务完成。这些其他事务根据尝试读取锁定行的值的时间有效地放入队列。

因为这种排队发生在读取操作期间，所以如果多个并发执行的事务尝试访问相同的数据，会阻止该选择的结果，Hubble 还可以防止可能发生的事务重试

使用SELECT FOR UPDATE导致竞争操作的吞吐量增加和尾部延迟减少。

所需权限

用户必须对用作操作数的表具有SELECT和UPDATE权限

语法图

```
ForUpdateStmt   ::=
'FOR'  'UPDATE'  of  tablename
```

有关完整的SELECT语句语法文档，请参阅选择查询

参数介绍

参数	简介
FOR UPDATE	锁定SELECT语句返回的行，以便尝试访问这些行的其他事务必须等待事务完成

查询示例

此示例用于SELECT FOR UPDATE锁定事务中的一行，强制其他想要更新同一行的事务等待第一个事务完成。想要更新同一行的其他事务根据它们第一次尝试读取该行的值的时间被有效地放入队列中。

首先，连接到正在运行的集群（称为终端 1），并创建一个表插入一些行

```
create table test (a int primary key, b int);
insert into  test (a, b) values (1, 3), (20,0), (2, 6);
```

接下来，启动一个事务并锁定要操作的行：

```
begin;
select  * from  test where a = 1 for  update;
```

在客户端中按两次 Enter 以发送到目前为止的输入以进行评估。这将导致以下输出：

```
  a | b
+---+----+
  1 | 3
```

现在打开另一个终端并从第二个客户端连接到数据库（称为终端 2），开始一个事务并尝试锁定同一行以获取我们在终端 1 中打开的事务已经访问的更新

```
begin;
select  * from  test where a = 1 for  update;
```

按两次 Enter 以发送到目前为止的输入以进行评估。因为终端 1 经锁定了这一行，所以SELECT FOR UPDATE来自终端 2 的语句将显示为“等待”。

回到终端 1，更新行并提交事务：

```
update test set b = b + 7 where  a = 1;
```

```
commit;
```

现在终端 1 中的交易已经提交，终端 2 中的交易将被“解锁”，生成以下输出

```
  a | b
+---+----+
  1 | 10
```

终端 2 中的事务现在可以接收输入，因此再次更新有问题的行

```
update test set b = b + 3 where  a = 1;
```

```
commit;
```

终端 2 中的交易留下的值

```
  a | b
+---+----+
  1 | 13
```

6.3.2.5 SYSTEM

6.3.2.5.1 SHOW SESSIONS

显示会话的详细信息：包括session_id、用户名、登陆方式、ip 地址、连接时长、活跃查询等等。

root用户可以查看所有当前登陆用户的session信息；其他用户只能看到自己的session信息。

也可以使用SHOW CLUSTER SESSIONS展示相应的信息。

返回参数说明

参数	含义
node_id	节点 ID
session_id	会话 ID
user_name	连接的用户名
client_adress	客户端地址和端口
application_name	客户端连接方式
active_queries	活动状态的 SQL 查询
last_active_query	最近完成的 SQL 查询
session_start	会话开始时间
oldest_query_start	当前运行时间最长的 SQL 查询
kv_txn	事务 ID

列出集群中活动会话

```
SHOW SESSIONS;
```

```
SHOW CLUSTER SESSIONS;
```

```
  node_id |             session_id             | user_name |    client_address    |
     ↪ application_name |     active_queries      | last_active_query |
     ↪ session_start            |         oldest_query_start
+---------+------------------------------------+-----------+----------------------+-----------
   ↪
        1 | 16b20c35abefaec10000000000000001 | root      | 192.168.1.11:11373 |
           ↪ $ hubble sql      | SHOW CLUSTER SESSIONS | SHOW database      |
           ↪ 2021-10-28 01:03:59.346244+00:00 | 2021-10-28
           ↪ 01:23:34.070655+00:00
(1 row)
```

查询指定用户的session信息

```
SELECT * FROM [SHOW CLUSTER SESSIONS] WHERE user_name = 'beagledata';
```

6.3.2.5.2 SHOW QUERIES

可以使用SHOW QUERIES或SHOW CLUSTER QUERIES查询当前活跃的 SQL 的执行信息。

返回参数说明

列名	说明
query_id	查询 ID
node_id	当前连接的节点 ID
username	当前连接的用户名
start	查询开始的时间
query	查询的 SQL 语句
client_adress	发起这条查询的客户端地址与端口
application_name	客户端使用的具体名称
distributed	true：这条查询会被分布式 SQL 执行引擎（DistSQL）执行。false：查询会通过本地的 SQL 引擎执行。null：表示该条查询正在准备，不知道会通过哪种引擎执行。
phase	查询执行的阶段。如果是if preparing，表示这条查询正在被解析和生成执行计划。如果是executing，表示这条语句正在执行。

```
SHOW QUERIES;
```

```
              query_id              | node_id | user_name |              start
                ↪                   |         query         |   client_address   |
                ↪ application_name | distributed |   phase
+----------------------------------+---------+-----------+-----------------------------
  ↪
  16b20ecf05fae0a60000000000000001 |       1 | root      | 2021-10-28
    ↪ 01:51:37.010185+00:00 | SHOW CLUSTER QUERIES | 192.168.1.11:11373 | $
    ↪ hubble sql     |    false    | executing
(1 row)
```

查看整个集群的 SQL

```
SHOW CLUSTER QUERIES;
```

```
SHOW queries;
```

```
              query_id              | node_id | user_name |              start
                ↪                   |         query         |   client_address   |
                ↪ application_name | distributed |   phase
+----------------------------------+---------+-----------+-----------------------------
  ↪
  16b20f390f56a7a30000000000000001 |       1 | root      | 2021-10-28
    ↪ 01:59:12.433748+00:00 | SHOW CLUSTER QUERIES | 192.168.1.11:11373 | $
    ↪ hubble sql     |    false    | executing
(1 row)
```

查看当前节点 SQL

```
SHOW LOCAL queries;
```

```
              query_id              | node_id | user_name |              start
                ↪                   |         query         |    client_address    |
                ↪ application_name | distributed |    phase
+----------------------------------+---------+-----------+-----------------------------
   ↪
  16b20f60f8928e3f0000000000000001 |       1 | root      | 2021-10-28
     ↪ 02:02:03.850485+00:00 | SHOW LOCAL QUERIES | 192.168.1.11:11373 | $
     ↪ hubble sql     |    false    | executing
(1 row)
```

查看具体的查询任务，可将SHOW QUERIES作为SELECT语句的数据源用于过滤 SQL，通过WHERE条件查找。

```
SELECT * FROM [ SHOW QUERIES ] WHERE node_id=1;
```

```
SELECT * FROM [ SHOW QUERIES ] WHERE START < (now()-INTERVAL '3 hours');
```

```
SELECT * FROM [ SHOW QUERIES ]
  WHERE client_address='192.168.100.113:32135'
  AND user_name='beagledata';
```

```
SELECT * FROM [ SHOW QUERIES ] WHERE application_name='hubbleapplication';
```

查询指定用户的query信息

```
SELECT * FROM [SHOW CLUSTER QUERIES] WHERE user_name = 'beagledata';
```

6.3.2.5.3 CANCEL QUERY

用于取消正在运行的 SQL 查询

所需权限

> root用户可以取消任何当前状态为active的查询，而非root用户只取消他们自己当前状态为active的查询。

```
CANCEL QUERY '15d611a6c5a6307f0000000000000001';
```

```
CANCEL QUERY (
    SELECT query_id FROM [SHOW CLUSTER QUERIES]
    WHERE client_address = '192.168.100.113:32135'
    AND user_name = 'beagledata'
    AND query = 'SELECT * FROM test.beagledata ORDER BY k' limit 1 );
```

> CANCEL QUERY只能用于单个查询 query id。如果使用的子查询并返回多个 query id，CANCEL
> ↪ QUERY语句将失败。

6.3.2.5.4 SHOW ALL

使用SHOW ALL语句可以查看当前session中所有的变量的值:

```
SHOW ALL;
```

查看一个具体变量的值时，可以使用SHOW SESSION varableName;

例如，查看编码格式:

```
SHOW SESSION client_encoding;
```

```
  client_encoding
+-----------------+
  UTF8
(1 row)
```

6.3.2.5.5 SET

使用SET语句可以修改客户端session中变量的值

参数说明

变量名称	默认值	说明
application_name	$ hubble sql	应用名称
database	defaultdb	数据库名
default_transaction_priority	normal	默认事务优先权
default_transaction_use_follower_reads	off	用于允许事务使用跟随者读取。
disable_partially_distributed_plans	off	禁用部分分发状态
disallow_full_table_scans	off	执行全表扫描的语句将被记录到慢查询日志中，即使它们不满足延迟阈值。必须启用慢查询日志才能使此设置生效。
enable_copying_partitioning_when_deinterleaving_table	off	启用复制分区取消表
enable_drop_enum_value	off	启用删除枚举值
enable_experimental_alter_column_type_general	off	启用实验更改列类型
enable_experimental_stream_replication	off	启用实验流复制，使能够集群流式传输
enable_implicit_select_for_update	on	启用隐式查询锁
enable_insert_fast_path	on	是否将使用专门的执行运算符来插入表。建议保留此设置 on。
enable_seqscan	on	启用序列扫描
escape_string_warning	on	避免字符警告
experimental_distsql_planning	off	分布式 sql 计划
experimental_enable_hash_sharded_indexes	off	启用 hash 分片索引
experimental_enable_implicit_column_partitioning	off	允许使用隐式列分区

变量名称	默认值	说明
experimental_enable_temp_tables	off	默认允许使用临时表
experimental_enable_unique_without_index_constraints	off	默认禁用没有唯一性的索引约束
experimental_use_new_schema_changer	off	是否对支持的语句使用声明性模式更改器。
foreign_key_cascades_limit	10000	限制作为单个查询的一部分运行的级联操作的数量
idle_in_session_timeout	0	自动终止超过指定阈值的空闲会话。
locality_optimized_partitioned_index_scan	on	启用在搜索远程区域之前搜索当前区域中的行
optimizer_improve_disjunction_selectivity	off	更准确地估计查询过滤器的选择性
optimizer_use_histograms	on	优化器使用收集的直方图进行基数估计。
optimizer_use_multicol_stats	on	优化器使用收集的多列统计信息进行基数估计。
override_multi_region_zone_config	off	允许覆盖多区域表或数据库的区域配置
prefer_lookup_joins_for_fks	off	使外键操作使用查找连接
require_explicit_primary_keys	off	在 CREATE TABLE 语句中显式主键的默认值
serial_normalization	rowid	表定义中 SERIAL 的默认处理
session_id		当前会话的 ID
stub_catalog_tables	on	stub_catalog_tables 会话设置的默认值
synchronous_commit	on	同步提交
testing_vectorize_inject_panics	off	用于测试矢量化注入

指定当前数据库：

```
SET database=database_0922;
```

- `idle_in_session_timeout`实例

自动终止超过指定阈值的空闲会话。当设置为 0 时，会话不会超时。

```
show session idle_in_session_timeout;
```

```
  idle_in_session_timeout
---------------------------
  0
(1 row)
```

```
select current_time;
```

```
    current_time
----------------------
  11:41:31.810453+08
(1 row)
```

```
select current_time;
```

```
     current_time
----------------------
  11:42:26.815024+08
(1 row)
```

设置 10s 超时

```
set idle_in_session_timeout='10s';
```

```
select current_time;
```

```
    current_time
---------------------
  11:43:43.89054+08
(1 row)
```

```
select current_time;
```

```
invalid syntax: statement ignored: unexpected error: write tcp
    ↪ 192.168.100.134:27604->192.168.100.134:35432: write: broken pipe
warning: error retrieving the transaction status: write tcp
    ↪ 192.168.100.134:27604->192.168.100.134:35432: write: broken pipe
warning: error retrieving the database name: write tcp
    ↪ 192.168.100.134:27604->192.168.100.134:35432: write: broken pipe
root@ty-bj03-test04:35432/? ?>
```

- 关于时区

默认时区为 UTC, 如果需要修改客户端显示的时区，设置后显示的时间即为中国时间。

```
SET timezone='Asia/Shanghai';
```

- session变量传参设置功能options

```
jdbc:postgresql://xx:35432/defaultdb?ssl=true&sslmode=require&options=--
    ↪ statement_timeout=1000
```

HUBBLE

6.3.2.5.6 SHOW CLUSTER SETTING

- 集群变量

变量名称	默认值	说明
changefeed.experimental_poll_interval	1s	实现的轮询间隔（可能会损害集群的稳定性或正确性；请勿在没有监督的情况下进行编辑）
cloudstorage.timeout	10ms	导入/导出存储操作的超时
schemachanger.backfiller.max_sst_size	16MiB	读取文件的目标大小
diagnostics.reporting.enabled	true	允许向实验室报告诊断指标
diagnostics.reporting.send_crash_reports	true	发送崩溃和报告
external.graphite.interval	10s	指标推的送时间间隔
jobs.registry.leniency	1ms	推迟任何重新安排工作的尝试的时间
kv.allocator.lease_rebalancing_aggressiveness	1	设置大于 1 以更积极地重新平衡租约，或设置在 0 和 1.0 之间以更保守地重新平衡租约
kv.allocator.load_based_lease_rebalancing.enabled	true	设置以启用基于负载和延迟的范围租约的重新平衡
kv.allocator.qps_rebalance_threshold	0.25	与 QPS 的平均值相差的最小分数可以被认为是过满或过满
kv.bulk_ingest.buffer_increment	32MiB	批量添加器在刷新之前尝试增加其缓冲区的大小
kv.bulk_ingest.index_buffer_size	32MiB	处理二级索引导入的缓冲区的初始大小
kv.bulk_ingest.max_index_buffer_size	51 MiB	处理二级索引导入的缓冲区的最大大小
kv.bulk_ingest.max_pk_buffer_size	128MiB	处理主索引导入的缓冲区最大大小
kv.bulk_ingest.pk_buffer_size	32MiB	处理主索引导入的缓冲区的初始大小
kv.bulk_io_write.concurrent_addsstable_requests	1	存储在排队之前将同时处理的请求数
kv.bulk_io_write.concurrent_export_requests	3	排队之前将同时处理的导出请求数
kv.bulk_io_write.concurrent_import_requests	1	排队之前将同时处理的导入请求数

HUBBLE

变量名称	默认值	说明
kv.bulk_io_write.experimental_incremental_export_enabled	false	在 BACKUP 中导出时使用实验性的时间限制文件过滤器
kv.bulk_io_write.max_rate	1.0TiB	代表批量用于写入磁盘的速率限制（字节/秒）
kv.bulk_io_write.small_write_size	400KiB	低于该大小的批量写入将作为普通写入执行
kv.closed_timestamp.close_fraction	0.2	关闭时间戳目标持续时间的分数
kv.closed_timestamp.follower_reads_enabled	true	允许副本根据封闭的时间戳信息提供一致的历史读取
kv.raft_log.disable_synchronization_unsafe	false	设置 true 以禁用 Raft 日志写入持久存储的同步。设置为 true 会导致服务器崩溃时数据丢失或数据损坏。该设置仅用于内部测试，不应在生产中使用。
kv.range_descriptor_cache.size	1000000	范围描述符和租用者缓存中的最大条目数
kv.range_merge.queue_enabled	true	是否启用自动合并队列
kv.range_merge.queue_interval	1s	合并队列在处理副本之间等待多长时间
kv.range_split.by_load_enabled	true	允许根据负载集中的位置自动分割范围
kv.snapshot_rebalance.max_rate	8MiB	用于重新平衡和复制快照的速率限制（字节/秒）
kv.snapshot_recovery.max_rate	8MiB	用于恢复快照的速率限制（字节/秒）
kv.transaction.max_intents_bytes	262144	用于跟踪事务中写入意图的最大字节数
kv.transaction.max_refresh_spans_bytes	256000	用于跟踪可序列化事务中的刷新跨度的最大字节数
kv.transaction.parallel_commits_enabled	true	如果启用，事务提交将与事务写入并行
kv.transaction.write_pipelining_enabled	true	如果启用，事务写入通过 Raft 共识流水线化

HUBBLE

变量名称	默认值	说明
kv.transaction.write_pipelining_max_batch_size	128	如果非零，则定义将通过 Raft 共识流水线化的最大批量
kv.transaction.write_pipelining_max_outstanding_size	256 KiB	在禁用流水线之前用于跟踪正在进行的流水线写入的最大字节数
schemachanger.backfiller.buffer_increment	32MiB	加载器在刷新之前尝试增加其缓冲区的大小
schemachanger.backfiller.buffer_size	32MiB	缓冲区处理索引回填的初始大小
schemachanger.backfiller.max_buffer_size	512MiB	缓冲区处理索引回填的最大大小
schemachanger.backfiller.max_sst_size	16 MiB	读取文件的目标大小
server.consistency_check.interval	24h	范围一致性检查间隔的时间；设置 0 以禁用一致性检查
server.failed_reservation_timeout	5s	预留调用失败后考虑限制存储进行复制的时间量
server.heap_profile.max_profiles	5	要保留的最大配置文件数。分数较低的配置文件被 GC 处理，但始终保留最新的配置文件。
server.web_session_timeout	168h	新创建的 Web 会话有效的持续时间
sql.defaults.default_int_size	8	INT 类型的大小（以字节为单位）
sql.defaults.distsql	auto	默认分布式 SQL 执行模式 [off = 0,auto = 1,on = 2]
sql.defaults.experimental_optimizer_foreign_keys.enabled	false	默认启用优化器驱动的外键检查
sql.defaults.results_buffer.size	16 KiB	在将语句或一批语句发送到客户端之前累积结果的缓冲区的默认大小。
sql.defaults.vectorize	on	默认矢量化模式 [off = 0，on = 1]
sql.defaults.vectorize_row_count_threshold	1000	默认矢量化行计数值
sql.distsql.flow_stream_timeout	10s	输入流在出错之前等待设置流的时间量

HUBBLE

变量名称	默认值	说明
sql.distsql.max_running_flows	500	一个节点上可以运行的最大并发流数
sql.distsql.temp_storage.workmem	64MiB	处理器在回退到临时存储之前可以使用的最大内存量（以字节为单位）
sql.metrics.statement_details.dump_to_logs	false	定期清除时将收集的语句统计信息转储到节点日志
sql.metrics.statement_details.enabled	true	收集每个语句的查询统计信息
sql.metrics.statement_details.plan_collection.enabled	true	定期为每个指纹保存一个逻辑计划
sql.metrics.statement_details.plan_collection.period	5ms	收集新逻辑计划的时间
sql.metrics.statement_details.threshold	0s	收集统计信息的最短执行时间
sql.metrics.transaction_details.enabled	true	收集每个应用程序的事务统计信息
sql.query_cache.enabled	true	启用查询缓存
sql.stats.automatic_collection.enabled	true	自动统计收集模式
sql.stats.histogram_collection.enabled	true	直方图采集模式
sql.tablecache.lease.refresh_limit	50	定期刷新租约的最大数
sql.trace.log_statement_execute	false	设置为 true 以启用已执行语句的日志记录
sql.trace.session_eventlog.enabled	false	设置为 true 以启用会话跟踪
sql.trace.txn.enable_threshold	0s	跟踪所有事务的持续时间（设置为 0 以禁用）
timeseries.storage.enabled	true	如果设置 true，则将周期性时间序列数据存储在集群中

查看一个或多个`clister setting`的值，也可以通过`SET CLUSTER SETTING`进行配置。

> 查看所有`Cluster Setting`的值，需要 admin 权限

```
SHOW ALL CLUSTER SETTINGS;
```

使用`SET CLUSTER SETTING`语句可以修改集群中变量的默认值。

需要用户拥有 admin 权限

```
SET CLUSTER SETTING <variable>=<value>;
```

例如：执行语句超时时间设置，默认值 0

```
SET CLUSTER SETTING sql.defaults.statement_timeout = '0s';
```

6.3.2.5.7 EXPERIMENTAL_AUDIT

语句用来打开或关闭表的 SQL 审计。

所需权限

只有 root 用户可以打开表的审计日志

审计日志包含有关针对你的系统执行查询的详细信息，包括

- 查询语句全文
- 时间
- 客户端地址
- 应用名

```
ALTER TABLE customers experimental_audit SET READ WRITE;
```

开启后，将会把所有关于这个表的操作记录在日志中。

```
ALTER TABLE customers EXPERIMENTAL_AUDIT SET OFF;
```

6.3.2.6 JOBS

6.3.2.6.1 EXPORT

使用EXPORT导出表格数据到CSV文件中。

所需权限

需要用户拥有admin权限。

注意：

此语句不支持事务。

建议导出为每个节点都可以访问到的公共文件。

从 hubble 中导出数据

以下示例 nodelocal://1/ ，其中数字 1 代表在第一个节点进行操作，且推荐。

HUBBLE

```
export into
csv 'nodelocal://1/cust'
with delimiter ='|',nullas=''
from
select * from tt_image1;
```

常用的分隔符有\t 或者\001 等，以\001 为例

数据准备：

```
create table cust_info(
    cust_no        string primary key,
    cust_name      varchar(30) not null,
    cust_card_no   varchar(18),
    cust_phoneno   decimal(15),
    cust_address   varchar(30),
    cust_type      varchar(10),
    index(cust_card_no)
 );

insert into cust_info values('14435550','王吉','12022519960321531X',15122511874,
    ↪ '天津武清','抵押');
insert into cust_info values('14435551','张贺','431256197306265320',15534343555,
    ↪ null,'质押');
...
```

nullas 参数来定义字段表示过滤 null，比如：以上第二条 insert 语句中地址字段为空值

```
export into
csv 'nodelocal://1/cust'
with delimiter =e'\001',nullas=''
from
select * from cust_info;
```

```
                    filename                                          | rows |
                        ↪ bytes
-------------------------------------------------------------------+------+-------------
    ↪
  export17665471189f3d580000000000000001-n871817396986347523.0.csv |  115 |
      ↪ 121347951
```

对应的导入语句见下文import栏目

> 注意：在处理更多行之前上传文件大小的默认值为 32M，为减少导出的 csv 数量，加入参数chunk_size，建议chunk_size='512M'

```
export into
csv 'nodelocal://1/cust'
```

```
with delimiter ='|' ,chunk_size='512M'
from
select * from tt_image1;
```

返回结果：文件数据大，可能会导出多个 csv

```
                    filename                        | rows |    bytes
----------------------------------------------------+------+------------
  export16d82d3d6101475c0000000000000001-n3.0.csv |   15 | 121347951
  export16d82d3d6101475c0000000000000001-n4.0.csv |   44 | 355954016
```

6.3.2.6.2 IMPORT

- 用于将表格数据导入到单个表中，无法在事务中使用。
- 成功启动导入后，它将导入注册为 job，你可以使用SHOW JOBS查看。
- 导入开会后，你可以使用PAUSE JOB,RESUME JOB和CANCEL JOB来控制它。
- 暂停然后恢复IMPORT作业将导致它从头开始重新启动。

所需权限

> 只有root用户才能运行IMPORT任务。

> **注意** csv 的导入：因为目录结尾是 *.csv，所以目录下只能放本张表导出的 csv

在导入数据前一定要先创建表

创建表 img，并将数据导入到 img

```
import into img ( a , b
) csv data ( 'nodelocal://1/cust/*.csv' )
with delimiter = '|'
;
```

返回结果：

```
       job_id        |  status   | fraction_completed | rows | index_entries |
          ↪ system_records |   bytes
+--------------------+-----------+--------------------+------+---------------+------------
   ↪
  739023580166127617 | succeeded |                  1 |   59 |             0 |
     ↪              0 | 238652345
```

分隔符\001 为例，将数据进行导入

nullif 参数防止导入的 null 值都变成空字符串

```
import into cust_info1 CSV DATA(
'nodelocal://1/cust/*.csv')WITH delimiter = e'\001',nullif = e'';
```

```
       job_id          |  status    | fraction_completed | rows | index_entries |
          ↪    bytes
+-------------------
   ↪ +-----------+--------------------+------+---------------+-----------+
  871818819657760769 | succeeded |                  1 |  115 |             5 |
     ↪ 238652345
```

在数据导入的过程中，字段中有可能存在特殊字符，如果字段存在有双引号 (例如:” 北京”) 或者字段有转义字符 (例如:shanghai\pudong)，导入时候可以参考以下参数:

- fields_terminated_by用于单个字符的分割
- fields_enclosed_by 用于将字段的内容包装起来的符号的情况，例如: 双引号
- fields_escaped_by用于字段中包含的转义字符的情况，例如:\

如下一张表:

```
create table test (id int,cust_address string,aml string);
```

文件test.txt里面的样例数据如下:

```
1,"beijing",qwe\123
2,"shanghai",env\456
```

```
import into  test delimited data  ('nodelocal://1/test.txt')
with fields_terminated_by = e',', fields_enclosed_by='"', fields_escaped_by='\';
```

```
        job_id          |  status    | fraction_completed | rows | index_entries |
           ↪ bytes
---------------------+-----------+--------------------+------+---------------+--------
   ↪
  779219645387079681 | succeeded |                  1 |    2 |             0 |
     ↪     78
```

6.3.2.6.3 SHOW JOBS

查看集群中所有长时间运行的任务，用于获取到影响集群性能的关键任务，从而帮助用户来控制任务状态。

所需权限

> 仅root用户执行SHOW JOBS语句

```
SHOW JOBS;
```

可将SHOW JOBS作为SELECT语句的数据源用于过滤 job，通过WHERE条件来查找需要的 job 信息。

```
SELECT * FROM [ SHOW JOBS ]
WHERE
    job_type = 'RESTORE'
    AND status IN ( 'running', 'succeeded' )
ORDER BY
```

```
    created DESC;
```

6.3.2.6.4 CANCEL JOB

用于停止长时间运行的 job，包括IMPORT、BACKUP与RESTORE任务。取消RESTORE后，将清理部分还原的数据。可能会对集群性能产生短暂的较小的影响。

所需权限

> 默认情况下，只有root用户可以使用CANCEL JOB

停止单个任务

```
CANCEL JOB <job_id>;
```

停止多个任务

```
CANCEL JOBS (SELECT job_id FROM [SHOW JOBS] WHERE user_name = 'beagledata');
```

6.3.2.6.5 PAUSE JOB

PAUSE JOB语句可以暂停的IMPORT、BACKUP、RESTORE、USER-CREATED TABLESTATISTICS JOBS
↪ 和AUTOMATICTABLE STATISTICS JOBS任务。

> **注意**
>
> 无法暂停 schema 的修改

所需权限

> 用户需要 root 权限

暂停单个任务

```
PAUSE JOB <job_id>;
```

暂停多个任务

```
PAUSE JOBS (SELECT job_id FROM [SHOW JOBS] WHERE user_name = 'beagledata');
```

6.3.2.6.6 RESUME JOB

RESUME JOB语句可以恢复已暂停的IMPORT、BACKUP或RESTORE任务。

所需权限

> 用户需要root权限

恢复单个JOB

```
RESUME job <job_id>;
```

恢复多个JOB

```
RESUME JOBS (SELECT job_id FROM [SHOW JOBS] WHERE user_name = 'beagledata');
```

6.3.2.6.7 EXPLAIN

EXPLAIN 说明

语句返回指定语句的查询计划，用户可以根据explain语句优化查询。

查询优化

使用EXPLAIN输出，可以按如下方式优化查询：

- 具有较少数据的查询条件执行得更快。
- 避免扫描整个表，这是访问数据最慢的方式。创建至少包含查询在其子句中过滤的列之一的索引。

可以通过以下方式了解查询是否正在执行整个表扫描EXPLAIN：

- 查询使用的索引；显示为 table 的值。
- 正在扫描索引中的键值；显示为 spans 的值。

参数介绍

参数	详情
VERBOSE	尽可能多地显示有关声明计划的信息。
TYPES	选择评估中间 SQL 表达式的中间数据类型。
OPT	显示由基于成本的优化器生成的语句计划树。
VEC	显示有关查询的矢量化执行计划的详细信息。
DISTSQL	生成一个指向分布式 SQL 物理语句计划图的 URL。
preparable_stmt	想要了解详细信息的语句。所有可准备的陈述都是可以解释的。

返回信息

描述	详情
全局属性	适用于整个查询计划的属性。全局属性包括 distribution 和 vectorized。
报表计划树属性	报表计划层次结构的树表示。
指数推荐：N	索引建议的数量，后跟索引操作列表和执行操作的 SQL 语句。
时间	查询的时间详细信息，总时间是查询的计划和执行时间。执行时间是最终语句计划完成所用的时间，网络时间是在集群中的相关节点上分发查询所花费的时间。有些查询不需要分布式，所以网络时间为 0ms。

定位索引和键范围

可以使用EXPLAIN来了解查询使用哪些索引和键范围，这可以确定查询没有执行全表扫描。

```
create table shops (aml int primary key, num int);
```

- 用非索引列查询

```
explain select  * from shops where num between 4 and 6;
```

```
                                        info
-------------------------------------------------------------------------------------
  ↪
  distribution: full
  vectorized: false

  • filter
    estimated row count: 1
    filter: (num >= 4) AND (num <= 6)

      • scan
        estimated row count: 1 (100% of the table; stats collected 2 seconds ago
          ↪ )
        table: shops@primary
        spans: FULL SCAN
(11 rows)
```

说明：重点查看spans值，因为列 num 不是索引列，进行过滤的查询会扫描整个表。

- 用索引列查询

```
explain select  * from shops where aml between 80 and 100;
```

```
                                    info
-----------------------------------------------------------------------------
  distribution: local
  vectorized: false

  • scan
    estimated row count: 1 (100% of the table; stats collected 6 hours ago)
    table: shops@primary
    spans: [/80 - /100]
```

说明：重点查看spans值，因为列 aml 是索引列，进行过滤的查询语句避免了全表扫描。

单表执行样例

```
explain select * from bptfhist   where tx_tm>151111  order by tx_tm asc;
```

```
                                                 info
-------------------------------------------------------------------------------------------
  ↪
  distribution: full
  vectorized: true

  • sort
    estimated row count: 40,854
    order: +tx_tm
```

```
    • filter
      estimated row count: 40,854
      filter: tx_tm > 151111

        • scan
          estimated row count: 100,000 (100% of the table; stats collected 24
              ↪ days ago)
          table: bptfhist@primary
          spans: FULL SCAN
```

- distribution：full

计划者选择了一个分布式执行计划，其中查询的执行由多个节点并行执行，然后由网关节点返回结果。具有 full 分布的执行计划不会在集群中的所有节点上处理。local 仅在网关节点上执行具有分布的执行计划。即使执行计划是 local，也可能从远程节点获取行数据，但数据的处理由本地节点执行。

- vectorized：true

该计划将使用矢量化执行引擎执行。

- estimated row count：100,000 (100% of the table; stats collected 24 days ago)

查询扫描的估计行数，在本例中 100000 为数据行；查询跨越的表的百分比，在本例中为 100%；以及上次收集表的统计信息的时间。

- order：tx_tm

排序将在 revenue 列上按升序排列。

- filter：tx_tm > 151111

列上的扫描过滤器 revenue。

- table：bptfhist@primary

在索引上扫描表。

- spans：FULL SCAN

该表在索引的所有键范围上进行即全表扫描。

表联合执行样例

```
explain select  * from test_trans as t   join bptfhist as b on b.ac = t.ac;
```

```
 distribution: full
  vectorized: false

  • lookup join
    table: bptfhist@primary
    equality: (rowid) = (rowid)
    equality cols are key

      • lookup join
        estimated row count: 1
        table: bptfhist@idx_bptfhist3
```

```
        equality: (ac) = (ac)

          • scan
            estimated row count: 1 (100% of the table; stats collected 17
               ↪ minutes ago)
            table: test_trans@primary
            spans: FULL SCAN
```

备注：相比于单表查询equality输出显示查询将执行的连接条件。

6.3.2.6.8 EXPLAIN ANALYZE

EXPLAIN ANALYZE 说明

EXPLAIN ANALYZE语句的工作方式类似于EXPLAIN，主要区别在于前者实际上会执行语句。这样可以将查询计划中的估计值与执行时所遇到的实际值进行比较。

参数介绍

参数	详情
PLAN	执行该语句并返回一个带有可解释语句的计划和执行时间的语句计划。
DISTSQL	执行语句并返回语句计划和性能统计信息以及生成的指向图形分布式 SQL 物理语句计划树的链接。
DEBUG	执行该语句并生成一个 ZIP 文件，其中包含有关查询和查询中引用的数据库对象的详细信息的文件。
preparable_stmt	您要执行和分析的语句。所有可准备的陈述都是可以解释的。

备注：默认情况下，EXPLAIN ANALYZE使用该PLAN选项，EXPLAIN ANALYZE与EXPLAIN ANALYZE (PLAN)产生相同的输出，也是常用的。

返回信息

一条成功的EXPLAIN ANALYZE语句会返回一个表，列中包含以下详细信息：

参数	详情
全局属性	适用于整个语句计划的属性和统计信息。
报表计划树属性	报表计划层次结构的树表示。
节点详细信息	树中当前语句计划节点的属性、列和排序详细信息。
时间	声明的时间详细信息。总时间是语句的计划和执行时间。执行时间是最终语句计划完成所用的时间。网络时间是在集群中的相关节点之间分发语句所花费的时间。有些语句不需要分发，所以网络时间为 0ms。

语句分析重点是全局属性和报表计划树属性两类信息的查看

全局属性

参数	详情
planning time	计划者创建报表计划所花费的总时间。
execution time	完成最终执行计划所需的时间。
distribution	声明是分发的还是本地的。如果 distribution 是 full，则语句的执行由多个节点并行执行，则结果由网关节点返回。如果 local，则仅在网关节点上执行执行计划。即使执行计划是 local，也可能从远程节点获取行数据，但数据的处理由本地节点执行
vectorized	指示此语句中是否使用了向量化执行引擎
rows read from KV	从存储层读取的行数
cumulative time spent ↪ in KV	在存储层中花费的总时间
maximum memory usage	此语句在执行期间随时使用的最大内存量
network usage	执行语句时通过网络传输的数据量。如果值为 0 B，则该语句在单个节点上执行并且没有使用网络
regions	受影响节点所在的区域
max sql temp disk ↪ usage	(仅DISTSQL参数) 执行查询时发生多少磁盘溢出。仅当磁盘使用率大于零时才显示此属性

报表计划树属性

参数	详情
processor	语句计划层次结构中的每个处理器都有一个节点，其中包含有关语句该阶段的详细信息。如有GROUP BY子句的语句有一个group处理器，其中包含有关集群节点、行和与操作相关的GROUP BY操作的详细信息
nodes	受此阶段语句影响的集群节点名称
regions	受影响节点所在的区域
actual row count	执行期间受此处理器影响的实际行数
KV time	此阶段语句在存储层中的总时间
KV contention time	在此声明阶段中存储层争用的时间
KV rows read	在扫描期间，此阶段语句读取的存储层中的行数。
KV bytes read	在扫描期间，在语句的这个阶段从存储层读取的数据量
estimated max ↪ memory allocated	为语句估计的最大分配内存
estimated max sql ↪ temp disk usage	语句的估计最大临时磁盘使用量
estimated row ↪ count	根据语句计划器估计受此处理器影响的行数、查询跨越的表的百分比以及上次收集表的统计信息的时间
table	语句中的扫描操作中使用的表和索引，格式为{table name}@{index name}
spans	处理器读取密钥空间的间隔，如果spans是FULL SCAN，则在索引的所有键范围上扫描该表。如果spans是，则处理器仅读取 [/1 - /1] 具有值的键

EXPLAIN ANALYZE

EXPLAIN ANALYZE语句执行一个查询

```
explain analyze  select jrnno, avg(tx_tm) from bptfhist group by jrnno;
```

以下展示统计的信息：

```
                                        info
-----------------------------------------------------------------------------------------
  ↪
  planning time: 614µs
  execution time: 307ms
  distribution: full
  vectorized: true
  rows read from KV: 100,000 (76 MiB)
  cumulative time spent in KV: 235ms
  maximum memory usage: 16 MiB
  network usage: 384 B (3 messages)
  regions: ch-beijin

  • group
    nodes: n4
    regions: ch-beijin
    actual row count: 4
    estimated row count: 4
    group by: jrnno

      • scan
        nodes: n4
        regions: ch-beijin
        actual row count: 100,000
        KV rows read: 100,000
        KV bytes read: 76 MiB
        estimated row count: 100,000 (100% of the table; stats collected 24 days
          ↪ ago)
        table: bptfhist@primary
        spans: FULL SCAN
```

说明：主要关注以下信息：

- execution time

完成最终执行计划所需的时间

- distribution

计划者选择了一个分布式执行计划，其中查询的执行由多个节点并行执行，然后由网关节点返回结果。具有 full 分布的执行计划不会在集群中的所有节点上处理。local 仅在网关节点上执行具有分布的执行计划。即使执行计划是 local，也可能从远程节点获取行数据，但数据的处理由本地节点执行。

- vectorized

该执行是否使用矢量化执行引擎执行。

- rows read from KV

从存储层读取的行数

- cumulative time spent in KV

在存储层中花费的总时间

- maximum memory usage

此语句在执行期间随时使用的最大内存量

- network usage

执行语句时通过网络传输的数据量。如果值为 0 B，则该语句在单个节点上执行并且没有使用网络

- actual row count

执行期间受此处理器影响的实际行数

- KV rows read

在扫描期间，此阶段语句读取的存储层中的行数

- KV bytes read

在扫描期间，在语句的这个阶段从存储层读取的数据量

- estimated max memory allocated

为语句估计的最大分配内存

- estimated max sql temp disk usage

语句的估计最大临时磁盘使用量

- table

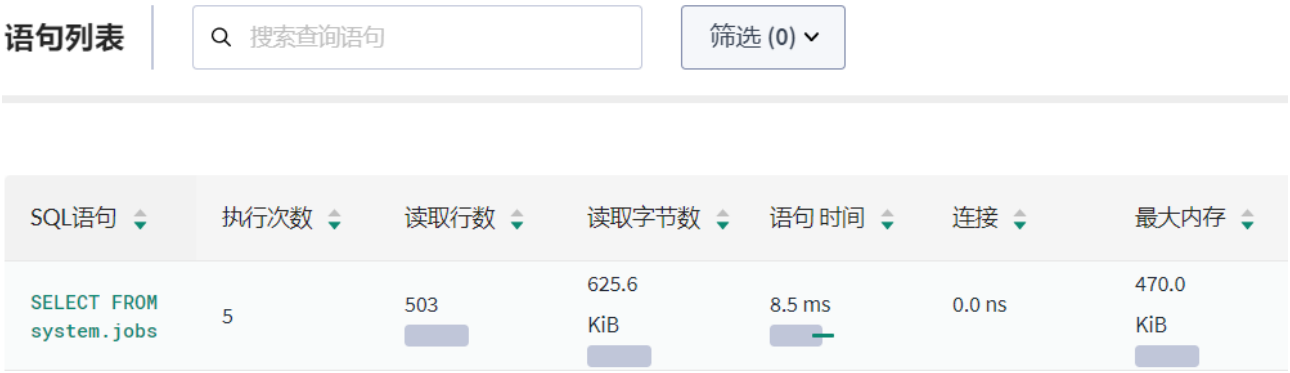
语句中的扫描操作中使用的表和索引，格式为 {table name}@{index name}

- spans

处理器读取密钥空间的间隔。如果 spans 是 FULL SCAN，则在索引的所有键范围上扫描该表。如果 spans 是 [/1 - /1]，则处理器读取范围键的值是 [/1 - /1]

通过监控页面分析 sql 执行情况

1. 登录页面后，点击语句“语句列表”，页面可以清晰的查看到每个 SQL 的基本



2. 选中执行的 SQL 语句，点击进去，可以具体查看 SQL 的概况、诊断、逻辑计划及其执行状态

← SQL语句

概况 诊断 逻辑计划 执行状态

```
SELECT * FROM bptfhist LIMIT _
```

6.3.3 数据类型

6.3.3.1 支持的类型

类型	描述	样例	向量量化执行
ARRAY	存储了非数组类型的一维度，一索引的相同类型元素的数组	{"sky","road","car"}	不支持
BIT	二进制数字（位）	B'10010101'	不支持
BOOL	布尔值	true	支持
BYTES	二进制字符	b ↪ '\141\061\142\062\143\063' ↪	支持
COLLATE	COLLATE功能使您可以根据特定于语言和国家/地区的规则（称为归类）对 STRING 值进行排序。	'a1b2c3' COLLATE en	不支持
DATE	日期	DATE '2019-01-25'	支持
DECIMAL	精确的定点数	1.2345	部分支持
FLOAT	64 位浮点数	1.2345	支持
INET	IPv4 或 IPv6 地址	192.168.0.1	不支持
INT	带符号整数，最大 64 位	12345	支持
INTERVAL	时间跨度/间隔	INTERVAL '2h30m30s'	不支持
JSONB	JSON (JavaScript Object Notation)	'{"first_name": "Lola", ↪ "last_name": "Dog", " ↪ location": "NYC", " ↪ online" : true, " ↪ friends" : 547}'	不支持
SERIAL	将整数类型与默认表达式组合在一起的伪类型	148591304110702593	不支持
STRING	Unicode 字符串	'a1b2c3'	支持
TIME	UTC 时间	TIME '01:23:45.123456'	不支持
TIMESTAMP，TIMESTAMPTZ	UTC 中的日期和时间对	TIMESTAMP '2016-01-25 ↪ 10:10:10' TIMESTAMPTZ '2016-01-25 ↪ 10:10:10-05:00'	TIMESTAMP 支持 TIMESTAMPTZ 不支持
UUID	128 位十六进制值	7f9c24e8-3b12-4fef-91e0 ↪ -56a2d5a246ec	支持

HUBBLE

类型	描述	样例	向量量化执行
ENUM	枚举类型	('open', 'closed')	不支持

6.3.3.2 数据类型转换

使用以下方法进行显式类型转换：

- <type> 'string literal'，用于将值的字符串形式转换为该类型的值。例如：DATE '2008-12-21'，INT '123'，或 BOOL 'true'。
- <value>::<data type>或等效的更长格式 CAST(<value> AS <data type>)，它将一个内置类型的任意表达式转换为另一种（也称为类型强制或“转换”）如：NOW()::DECIMAL，VARIANCE(a+2)::INT。

> 注意：要创建常量值，请考虑使用类型注释而不是强制类型转换，因为它可提供更可预测的结果。

不是 SQL 类型时，其他内置转换函数，例如from_ip()，to_ip()，可在STRING和BYTES值之间转换 IP 地址。

您可以在每种数据类型详情的支持的转换部分的找到对应类型的转换信息。

6.3.3.3 ARRAY

ARRAY 数据类型存储了非数组类型的一维的相同类型元素的数据。

ARRAY 数据类型对于确保与 ORM 和其他工具的兼容性很有用。但是，如果不考虑这种兼容性，则可以更灵活的使用规范化的表。

> 注意：数据库不支持嵌套数组，同时不支持在数组上创建数据库索引以及不支持按数组排序

6.3.3.3.1 语法

数据类型 ARRAY 的值可以通过以下方式表示：

- 将方括号 [] 附加到任何非数组数据类型。
- 将术语 ARRAY 添加到任何非数组数据类型。

6.3.3.3.2 大小

ARRAY 的值大小是可变的，为确保性能最好小于 1MB。超过该阈值，写放大和其他考虑因素可能导致性能显着下降。

6.3.3.3.3 示例

通过添加方括号创建数组列

```
CREATE TABLE a (b STRING[]);
```

```
INSERT INTO a VALUES (ARRAY['sky', 'road', 'car']);
```

```
SELECT * FROM a;
```

```
+----------------------+
|          b           |
+----------------------+
| {"sky","road","car"} |
+----------------------+
(1 row)
```

通过添加 `ARRAY` 来创建数组列

```
CREATE TABLE c (d INT ARRAY);
```

```
INSERT INTO c VALUES (ARRAY[10,20,30]);
```

```
SELECT * FROM c;
```

```
+------------+
|     d      |
+------------+
| {10,20,30} |
+------------+
(1 row)
```

使用数组索引访问数组元素

```
SELECT * FROM c;
```

```
+------------+
|     d      |
+------------+
| {10,20,30} |
+------------+
(1 row)
```

```
SELECT d[2] FROM c;
```

```
+------+
| d[2] |
+------+
|   20 |
+------+
(1 row)
```

将元素追加到数组

使用array_append函数

```
SELECT * FROM c;
```

HUBBLE

```
+------------+
|     d      |
+------------+
| {10,20,30} |
+------------+
(1 row)
```

```
UPDATE c SET d = array_append(d, 40) WHERE d[3] = 30;
```

```
SELECT * FROM c;
```

```
+---------------+
|       d       |
+---------------+
| {10,20,30,40} |
+---------------+
(1 row)
```

使用 append（||）运算符

```
SELECT * FROM c;
```

```
+---------------+
|       d       |
+---------------+
| {10,20,30,40} |
+---------------+
(1 row)
```

```
UPDATE c SET d = d || 50 WHERE d[4] = 40;
```

```
SELECT * FROM c;
```

```
+------------------+
|        d         |
+------------------+
| {10,20,30,40,50} |
+------------------+
(1 row)
```

6.3.3.3.4 支持的转换

当数组的数据类型支持转换时，支持在ARRAY值之间进行转换。例如，可以将BOOL数组强制转换为INT数组，而不能将BOOL数组强制转换为TIMESTAMP数组：

```
SELECT ARRAY[true,false,true]::INT[];
```

HUBBLE

```
   array
+---------+
  {1,0,1}
(1 row)
```

```
SELECT ARRAY[true,false,true]::TIMESTAMP[];
```

```
pq: invalid cast: bool[] -> TIMESTAMP[]
```

您可以将数组强制转换为STRING值，与 PostgreSQL 兼容：

```
SELECT ARRAY[1,NULL,3]::string;
```

```
    array
+------------+
  {1,NULL,3}
(1 row)
```

```
SELECT ARRAY[(1,'a b'),(2,'c"d')]::string;
```

```
                array
+----------------------------------+
  {"(1,\"a b\")","(2,\"c\"\"d\")"}
(1 row)
```

6.3.3.4 BIT

BIT和VARBIT数据类型存储BIT数组。使用BIT，长度是固定的；使用VARBIT，长度可以可变。

6.3.3.4.1 别名

名称BIT VARYING是VARBIT的别名

6.3.3.4.2 语法

BIT数组常量表示为字面量串。例如，B'100101'表示 6 位的数组。

6.3.3.4.3 大小

BIT值中的位数确定如下：

类型	逻辑大小
BIT	1 bit
BIT(N)	N bits
VARBIT	没有最大值
VARBIT(N)	最大值为 N bits

BIT值的有效大小比其逻辑位数大一个有界的常数因子。在内部以 64 位为增量存储位数组，外加一个额外的整数值来编码长度。

BIT值的总大小可以任意大，但建议将值保持在 1MB 以下以确保性能。超过该阈值，写放大和其他考虑因素可能导致性能显着下降。

6.3.3.4.4 示例

通过添加方括号创建数组列

```
CREATE TABLE b (x BIT, y BIT(3), z VARBIT, w VARBIT(3));
```

```
SHOW COLUMNS FROM b;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |  indices  | is_hidden
+--
   ↪ -----------+-----------+-------------+----------------+-----------------------+-----
   ↪
  x           | BIT       |    true     | NULL           |
     ↪                     | {}          |   false
  y           | BIT(3)    |    true     | NULL           |
     ↪                     | {}          |   false
  z           | VARBIT    |    true     | NULL           |
     ↪                     | {}          |   false
  w           | VARBIT(3) |    true     | NULL           |
     ↪                     | {}          |   false
  rowid       | INT8      |    false    | unique_rowid() |
     ↪                     | {primary}   |   true
(5 rows)
```

```
INSERT INTO b(x, y, z, w) VALUES (B'1', B'101', B'1', B'1');
```

```
SELECT * FROM b;
```

```
  x |  y  | z | w
+---+-----+---+---+
  1 | 101 | 1 | 1
```

对于BIT类型，该值必须与指定大小完全匹配：

```
INSERT INTO b(x) VALUES (B'101');
```

```
pq: bit string length 3 does not match type BIT
```

```
INSERT INTO b(y) VALUES (B'10');
```

```
pq: bit string length 2 does not match type BIT(3)
```

对于VARBIT类型，该值不得大于指定的最大大小：

```
INSERT INTO b(w) VALUES (B'1010');
```

```
pq: bit string length 4 too large for type VARBIT(3)
```

6.3.3.4.5 支持的转换

BIT值可以强制转换为以下任何数据类型：

类型	细节
INT	将位数组转换为相应的数值，将这些位解释为该值是使用二进制补码编码的。如果位数组大于整数类型，则将忽略左侧多余的位。例如，B'1010':: INT等于 10。
STRING	将二进制数字打印为字符串。这将恢复文字表示形式。例如，B'1010':: STRING等于'1010'。

6.3.3.5 BOOL

BOOL数据类型存储布尔值false或true。

6.3.3.5.1 别名

BOOLEAN是BOOL的别名。

6.3.3.5.2 语法

BOOL有两个预定义的命名常量：TRUE和FALSE（名称不区分大小写）

或者，可以通过强制数值来获得布尔值：将零强制为FALSE，任何非零值为TRUE。

- CAST(0 AS BOOL) (false)
- CAST(123 AS BOOL) (true)

6.3.3.5.3 大小

BOOL值的宽度为 1 个字节，但由于 Hubble 库元数据的因素，总存储大小可能会更大。

6.3.3.5.4 示例

```
CREATE TABLE bool (a INT PRIMARY KEY, b BOOL, c BOOLEAN);
```

```
SHOW COLUMNS FROM bool;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
    ↪  |  indices  | is_hidden
+--
  ↪ -----------+-----------+-------------+----------------+-----------------------+------
  ↪
  a           | INT8      |    false    | NULL           |
    ↪                     | {primary} |   false
  b           | BOOL      |    true     | NULL           |
    ↪                     | {}        |   false
```

```
  c            | BOOL       |     true      | NULL            |
     ↪                        | {}          |    false
(3 rows)
```

```
INSERT INTO bool VALUES (12345, true, CAST(0 AS BOOL));
```

```
SELECT * FROM bool;
```

```
+-------+------+-------+
    a    |  b   |   c
+-------+------+-------+
  12345 | true | false
(1 row)
```

6.3.3.5.5 支持的转换

BOOL值可以强制转换为以下任何数据类型：

类型	细节
INT	将true转换为1，将false转换为0
DECIMAL	将true转换为1，将false转换为0
FLOAT	将true转换为1，将false转换为0
STRING	--

6.3.3.6 BYTES

BYTES数据类型存储可变长度的二进制字符串。

6.3.3.6.1 别名

BYTEA和BLOB是BYTES的别名。

6.3.3.6.2 语法

以下三个是相同字节数组的等效文字：`b'abc'`，`b'\141 \142 \143'`，`b'\x61 \x62 \x63'`。

除此语法外，还支持使用字符串文字，包括在上下文中的`'...'`，`e'...'`和`x'....'`。

6.3.3.6.3 大小

BYTES值的大小是可变的，但建议将值保持在 1 MB 以下以确保性能。超过该阈值，写放大和其他考虑因素可能导致性能显着下降。

6.3.3.6.4 示例

```
CREATE TABLE bytes (a INT PRIMARY KEY, b BYTES);

-- explicitly typed BYTES literals
```

```
INSERT INTO bytes VALUES (1, b'\141\142\143'), (2, b'\x61\x62\x63'), (3, b'\141\
    ↪ x62\c');

-- string literal implicitly typed as BYTES
INSERT INTO bytes VALUES (4, 'abc');


SELECT * FROM bytes;
```

```
+---+-----+
| a |  b  |
+---+-----+
| 1 | abc |
| 2 | abc |
| 3 | abc |
| 4 | abc |
+---+-----+
(4 rows)
```

6.3.3.6.5 支持的转换

可以将BYTES值显式转换为STRING。此转换总是成功。

支持两种转换模式，由会话变量bytea_output控制：

- hex（默认）：转换的输出以两个字符\，x开头，字符串的其余部分由输入中每个字节的十六进制编码组成。例如，x'48AA'::STRING产生'H\xaa'。
- escape：转换的输出包含输入中的每个字节，如果是 ASCII 字符则保持原样，否则使用八进制转义格式\NNN进行编码。例如，x'48AA'::STRING产生'H\xaa'。

可以将STRING值显式转换为BYTES。如果十六进制数字无效或十六进制数字为奇数，则此转换将失败。支持两种转换模式：

- 如果字符串以两个特殊字符\和x开头（例如\xAABB），则字符串的其余部分将被解释为十六进制数字序列。然后将字符串转换为字节数组，其中每对十六进制数字都转换为一个字节。
- 否则，该字符串将转换为包含其 UTF-8 编码的字节数组。

STRING与BYTES

尽管在许多情况下，STRING和BYTES似乎都具有相似的行为，但在将它们转换为另一种之前，应先了解它们的细微差别。

STRING 将其所有数据视为字符，或更具体地说，将其视为 Unicode 代码点。BYTES将其所有数据视为字节字符串。在实现上的这种差异可能导致行为发生显着不同。例如，让我们采用一个复杂的 Unicode 字符，例如 （雪人表情符号）：

```
SELECT length(' '::string);
```

```
  length
+--------+
       1
```

```
SELECT length(' '::bytes);
```

```
  length
+--------+
       3
```

在这种情况下，LENGTH(string)测量字符串中存在的 Unicode 代码点的数量，而LENGTH(bytes)测量存储该值所需的字节数。每个字符（或 Unicode 代码点）可以使用多个字节进行编码，因此两者之间的输出差异。

将文字转换为STRING与BYTES

过 SQL 客户端输入的文字将根据类型转换为其他值：

- BYTES在开头对\x赋予特殊含义，并通过将十六进制数字对替换为单个字节来转换其余部分。例如，\xff等效于值为 255 的单个字节。
- STRING没有给\x赋予特殊含义，因此所有字符都被视为不同的 Unicode 代码点。例如，\xff被视为长度为 4（\，x，f，和f）的STRING。

6.3.3.7 COLLATE

COLLATE功能使您可以根据特定的语言和国家地区的规则（称为排序规则）对STRING值进行排序。

排序后的字符串很重要，因为不同的语言对于字母顺序（尤其是重音字母）有不同的规则。例如，在德语中，重音字母与非重音字母一起排序，而在瑞典语中，带重音字母则放在字母的末尾。排序规则是用于排序的一组规则，通常对应于一种语言，尽管某些语言具有多个排序不同的排序规则；例如，葡萄牙语对巴西和欧洲方言分别使用排序规则（分别为pt-BR和pt-PT）。

6.3.3.7.1 细节

对整理字符串进行的操作，不能涉及排序规则不同的字符串或不具有排序规则的字符串。但是，可以动态添加或覆盖排序规则。

仅在需要按特定排序规则对字符串排序时才使用排序规则功能。我们之所以建议这样做，是因为每次构造或加载排序后的字符串到内存中时，数据库都会计算排序键，其大小与排序后的字符串的长度呈线性关系，此过程还需要额外的资源。

排序后的字符串可能比相应的未排序后的字符串大得多，具体取决于语言和字符串内容。例如，包含字符é的字符串在法语语言环境中比在中文语言中产生更大的排序键。

与未排序的字符串相比，为排序后字符串创建索引需要更多的磁盘空间。如果是索引排序后字符串，则除了要存储排序后的字符串之外，还必须存储排序键。

6.3.3.7.2 支持的排序

支持 Go 语言包提供的排序规则。<collation>参数是每行末尾的 BCP47 语言标记，紧跟前面的//。例如，支持南非荷兰语作为AF进行排序

6.3.3.7.3 SQL 句法

归类的字符串在 SQL 中用作常规字符串，但在其后附加了COLLATE子句。

- 列语法：STRING COLLATE <collation>。例如：

```
CREATE TABLE foo (a STRING COLLATE en PRIMARY KEY);
-- Note: STRING 可用任意别名
```

- 值语法：<STRING value> COLLATE <collation>。例如：

```
INSERT INTO foo VALUES ('dog' COLLATE en);
```

6.3.3.7.4 示例

指定列的排序规则

您可以在STRING列中为所有值设置默认排序规则。

```
CREATE TABLE de_names (name STRING COLLATE de PRIMARY KEY);
```

将值插入此列时，必须为每个值指定排序规则：

```
INSERT INTO de_names VALUES ('Backhaus' COLLATE de), ('Bär' COLLATE de), ('Baz'
    ↪ COLLATE de);
```

排序现在将按照de处理，在字母排序中将 $\ddot{a}$ 视为 a ：

```
SELECT * FROM de_names ORDER BY name;
```

```
    name
+----------+
  Backhaus
  Bär
  Baz
(3 rows)
```

通过非默认排序规则排序

您可以使用特定的排序规则而不是默认排序规则对列进行排序。

例如，如果您按德语（de）和瑞典语（sv）排序结果，则会收到不同的结果：

```
SELECT * FROM de_names ORDER BY name COLLATE sv;
```

```
    name
+----------+
  Backhaus
  Baz
  Bär
(3 rows)
```

点对点排序规则转换

您可以随时将任何字符串转换为排序规则。

```
SELECT 'A' COLLATE de < 'Ä' COLLATE de;
```

```
  ?column?
+----------+
    true
(1 row)
```

但是，您不能比较具有不同归类的值：

```
SELECT 'Ä' COLLATE sv < 'Ä' COLLATE de;
pq: unsupported comparison operator: <collatedstring{sv}> < <collatedstring{de}>
```

您还可以使用强制转换从值中删除排序规则。

```
SELECT CAST(name AS STRING) FROM de_names ORDER BY name;
```

```
    name
+----------+
  Backhaus
  Baz
  Bär
(3 rows)
```

显示字符串的排序规则

您可以使用pg_collation_for内置函数或其替代语法形式COLLATION FOR来返回整理后的字符串的语言环境名称。

```
SELECT pg_collation_for('Bär' COLLATE de);
```

```
  pg_collation_for
+------------------+
  "de"
(1 row)
```

这等效于：

```
SELECT COLLATION FOR ('Bär' COLLATE de);
```

```
  pg_collation_for
+------------------+
  "de"
(1 row)
```

6.3.3.8 DATE

该DATE 数据类型储存年月日信息。

6.3.3.8.1 语法

DATE类型的常量值可以使用解释后的文字或用DATE类型注释或强制为DATE类型的字符串文字表示。

日期的字符串格式为YYYY-MM-DD。例如：DATE'2016-12-23'。

支持在预期DATE值的上下文中使用未解释的字符串文本。

6.3.3.8.2 大小

DATE列支持最大宽度为 16 个字节的值，但是由于数据库元数据因素，总存储大小可能会更大。

6.3.3.8.3 示例

```
CREATE TABLE dates (a DATE PRIMARY KEY, b INT);
```

```
SHOW COLUMNS FROM dates;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |  indices  | is_hidden
+--
   ↪ -----------+-----------+-------------+----------------+----------------------+-----
   ↪
  a           | DATE      |    false    | NULL           |
     ↪                     | {primary} |   false
  b           | INT8      |    true     | NULL           |
     ↪                     | {}        |   false
(2 rows)
```

显式写入DATE文字：

```
INSERT INTO dates VALUES (DATE '2016-03-26', 12345);
```

隐式写入DATE文字：

```
INSERT INTO dates VALUES ('2016-03-27', 12345);
```

```
SELECT * FROM dates;
```

```
+------------+-------+
      a       |   b
+------------+-------+
  2016-03-26 | 12345
  2016-03-27 | 12345
+------------+-------+
(2 rows)
```

6.3.3.8.4 支持的转换

DATE值可以强制转换为以下任何数据类型：

类型	细节
DECIMAL	转换为 Unix 纪元（1970 年 1 月 1 日）以来的天数
FLOAT	转换为 Unix 纪元（1970 年 1 月 1 日）以来的天数
TIMESTAMP	将结果时间戳记中的时间设置为 00:00:00（午夜）。
INT	转换为 Unix 纪元（1970 年 1 月 1 日）以来的天数
STRING	--

6.3.3.9 DECIMAL

DECIMAL数据类型存储精确的小数。当保留精确的精度很重要时可以使用此类型，例如，使用货币数据。

6.3.3.9.1 别名

DEC和NUMRIC是DECIMAL的别名

6.3.3.9.2 精度和刻度

使用DECIMAL（precision，scale）限制DECIMAL列，其中precision是小数点左边和右边的最大位数，而scale
↪ 是小数点右边的精确位数。precision不能小于scale。使用DECIMAL（precision）等效于DECIMAL（
↪ precision，0）。

插入十进制值时：

- 如果小数点右边的数字超过该列的小数位数scale，则将四舍五入到scale。
- 如果小数点右边的数字小于列的小数位数scale，则填充 0 直到scale。
- 如果小数点左右的数字超过该列的精度precision，则会报误。
- 如果列的precision和scale相同，则插入的值必须四舍五入到小于 1。

6.3.3.9.3 语法

可以将DECIMAL类型的常量值输入为数字文字。例如：1.414或-1234。

无法使用数字文字直接输入正无穷大，负无穷大和 NaN(Not-a-Number) 的特殊 IEEE754 值，必须使用解释后的文字或字符串文字的显式转换来进行转换。

可以识别以下值：

语法	值
inf，infinity，+inf，+infinity	+∞
-inf，-infinity	-∞
nan	(Not-a-Number)

举例：

- DECIMAL '+Inf'
- '-Inf'::DECIMAL
- CAST('NaN' AS DECIMAL)

6.3.3.9.4 大小

DECIMAL值的大小是可变的，从 9 个字节开始。建议将值保持在 64 KB 以下，以确保性能。超过该阈值，写放大和其他因素可能导致性能显着下降。

6.3.3.9.5 示例

```
CREATE TABLE decimals (a DECIMAL PRIMARY KEY, b DECIMAL(10,5), c NUMERIC);
```

```
SHOW COLUMNS FROM decimals;
```

```
  column_name |    data_type    | is_nullable | column_default |
     ↪ generation_expression |  indices   | is_hidden
+--
   ↪ -----------+---------------+-------------+----------------+-----------------------+-
   ↪
  a           | DECIMAL        |    false    | NULL           |
     ↪                         | {primary} |   false
  b           | DECIMAL(10,5) |    true     | NULL           |
     ↪                         | {}        |   false
  c           | DECIMAL        |    true     | NULL           |
     ↪                         | {}        |   false
(3 rows)
```

```
INSERT INTO decimals VALUES (1.01234567890123456789, 1.01234567890123456789,
   ↪ 1.01234567890123456789);
```

```
SELECT * FROM decimals;
```

```
            a             |    b    |            c
+-------------------------+---------+-------------------------+
  1.01234567890123456789 | 1.01235 | 1.01234567890123456789
(1 rows)

# The value in "a" matches what was inserted exactly.
# The value in "b" has been rounded to the column's scale.
# The value in "c" is handled like "a" because NUMERIC is an alias.
```

6.3.3.9.6 支持的转换

DECIMAL值可以强制转换为以下任何数据类型：

类型	细节
BOOL	0 转换为false；所有其他值都转换为true
FLOAT	丢失精度，如果值的太大，则可能会四舍五入到 +/-无穷大；如果值的大小，则可能会四舍五入到 +/- 0
INT	截断小数的精度
STRING	--

6.3.3.10 FLOAT

支持各种不精确的浮点数数据类型，最高可达 17 位小数精度。它们在内部使用标准的双精度（64 位二进制编码）IEEE754 格式进行处理。

6.3.3.10.1 别名

Name	Aliases
FLOAT	None
REAL	FLOAT4
DOUBLE PRECISION	FLOAT8

6.3.3.10.2 语法

数值文本可以作为浮点类型的输入。例如：`1.414`或`-1234`。

无法使用数字文字直接输入正无穷大，负无穷大和 NaN(Not-a-Number) 的特殊 IEEE754 值，必须使用解释后的文字或字符串文字的显式转换来进行转换。

可以识别以下值：

语法	值
inf, infinity, +inf, +infinity	$+\infty$
-inf, -infinity	$-\infty$
nan	(Not-a-Number)

举例：

- `DECIMAL '+Inf'`
- `'-Inf'::DECIMAL`
- `CAST('NaN' AS DECIMAL)`

6.3.3.10.3 大小

`FLOAT`列最多支持 8 个字节的值，由于库元数据因素，总存储大小可能会更大。

6.3.3.10.4 示例

```
CREATE TABLE floats (a FLOAT PRIMARY KEY, b REAL, c DOUBLE PRECISION);
```

```
SHOW COLUMNS FROM floats;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
    ↪  |  indices  | is_hidden
+--
  ↪ -----------+-----------+-------------+----------------+-----------------------+------
  ↪
  a           | FLOAT8    |    false    | NULL           |
    ↪                      | {primary} |   false
```

```
  b            | FLOAT4     |     true      | NULL            |
     ↪                        | {}          |    false
  c            | FLOAT8     |     true      | NULL            |
     ↪                        | {}          |    false
(3 rows)
```

```
INSERT INTO floats VALUES (1.012345678901, 2.01234567890123456789, CAST('+Inf'
   ↪ AS FLOAT));
```

```
SELECT * FROM floats;
```

```
        a         |      b      |     c
+----------------+-----------+----------+
  1.012345678901 | 2.0123458 | Infinity
(1 row)
# Note that the value in "b" has been limited to 7 digits.
```

6.3.3.10.5 支持的转换

FLOAT值可以强制转换为以下任何数据类型：

类型	细节
BOOL	0 转换为false；所有其他值都转换为true
DECIMAL	如果值为 NaN 或 +/- Inf，则会报告错误。
INT	截断小数精度，并要求值介于-2 ^ 63 和 2 ^ 63-1 之间
STRING	--

6.3.3.11 INET

INET数据类型存储 IPv4 或 IPv6 地址。

6.3.3.11.1 语法

INET类型的常量值可以使用解释的文字或用INET类型注释或强制为INET类型的字符串文字表示。

INET常数可以使用以下格式表示：

格式	描述
IPv4	标准 RFC791 规定的 4 个八位位组的格式，分别用十进制数表示，并用句点分隔。地址之后可选择性添加子网掩码。Examples: '190.0.0.0', '190.0.0.0/24'
IPv6	标准 RFC8200 指定的格式的 8 个冒号分隔的 4 个十六进制数字组。IPv6 地址可以映射到 IPv4 地址。地址之后可选择性添加子网掩码。Examples: '2001:4f8:3:ba:2e0:81ff:fe22:d1f1', '2001:4f8:3:ba:2e0:81ff:fe22:d1f1/120', '::ffff:192.168.0.1/24'

举例：

- 显式 INET '192.168.0.1'
- 隐式 '192.168.0.1' 数据插入表时，数据类型隐式转换为 INET

> 注意: IPv4 地址将在 IPv6 地址（包括 IPv4 映射的 IPv6 地址）之前排序

6.3.3.11.2 大小

使用INET值存储 IPv4 占用 32 位，存储 IPv6 占用 128 位。

6.3.3.11.3 示例

```
CREATE TABLE computers (
    ip INET PRIMARY KEY,
    user_email STRING,
    registration_date DATE
  );
```

```
SHOW COLUMNS FROM computers;
```

```
     column_name     | data_type | is_nullable | column_default |
        ↪ generation_expression |  indices   | is_hidden
+--
   ↪ -----------------+-----------+-------------+----------------+-----------------------
   ↪
  ip                 | INET      |    false    | NULL           |
     ↪                       | {primary} |   false
  user_email         | STRING    |    true     | NULL           |
     ↪                       | {}        |   false
  registration_date | DATE      |    true     | NULL           |
     ↪                       | {}        |   false
(3 rows)
```

```
INSERT INTO computers
  VALUES
    ('192.168.0.1', 'info@google.com', '2018-01-31'),
    ('192.168.0.2/10', 'lauren@google.com', '2018-01-31'),
    ('2001:4f8:3:ba:2e0:81ff:fe22:d1f1/120', 'test@google.com', '2018-01-31');
```

```
SELECT * FROM computers;
```

```
                   ip                   |     user_email      | registration_date
+---------------------------------------+---------------------+-------------------+
  192.168.0.2/10                        | lauren@google.com | 2018-01-31
  192.168.0.1                           | info@google.com   | 2018-01-31
  2001:4f8:3:ba:2e0:81ff:fe22:d1f1/120 | test@google.com   | 2018-01-31
(3 rows)
```

6.3.3.11.4 支持的转换

INET值可以强制转换为以下数据类型：

- STRING-转换为'Address/subnet'格式。

6.3.3.12 INT

支持各种有符号整数数据类型。

6.3.3.12.1 名称和别名

名称	允许的宽度	别名	范围
INT	64-bit	INTEGER INT8 INT64 BIGINT	-9223372036854775807 到 +9223372036854775807
INT2	16-bit	SMALLINT	-32768 到 +32767
INT4	32-bit	None	-2147483648 到 +2147483647
INT8	64-bit	INT	-9223372036854775807 到 +9223372036854775807

6.3.3.12.2 语法

数值文本可以作为INT类型的输入。例如：42，-1234或0xCAFE

6.3.3.12.3 大小

不同的整数类型对允许值的范围设置了不同的约束，但是无论类型如何，所有整数都以相同的方式存储。较小的值比较大的值占用更少的空间（基于数值，而不是数据类型）。

64 位有符号整数的注意事项

默认情况下，INT是INT8的别名，它将创建 64 位带符号整数。这与 Postgres 的INT默认值（32 位默认值）不同，如果未将其编写为处理 64 位整数，无论是编写应用程序的语言，还是生成 SQL 时使用的 ORM 框架，则都可能会给应用程序带来问题。

例如，JavaScript 语言运行时将数字表示为 64 位浮点小数，这意味着 JS 运行时只能表示 53 位的数字精度，因此最大合理值为 2 的 53 次或 9007199254740992。这意味着默认INT的最大值远远大于 JavaScript 可以表示为整数的值。在视觉上，大小差异如下：

```
9223372036854775807 # INT default max value
   9007199254740991 # JS integer max value
```

鉴于上述情况，如果表包含具有默认大小的INT值的列，并且您正在从中读取或通过 JavaScript 对其进行写入，则将无法正确地将值读取和写入该列。如果您使用的是自动生成前端和后端代码的框架（例如 twirp），则可能会突然弹出此问题。后端代码可以处理 64 位带符号的整数，但是生成的客户端或是前端代码则不能。

如果您的应用程序要使用与默认值不同的整数大小，则可以更改以下一项或两项设置。例如，可以将以下任意一项设置为4，以使INT和SERIAL成为 32 位整数INT4和SERIAL4的别名。

1. default_int_size 会话变量。
2. sql.defaults.default_int_size 群集设置。

提示: 如果您的应用程序需要精度数字，请使用 DECIMAL 数据类型。

6.3.3.12.4 示例

```
CREATE TABLE ints (a INT PRIMARY KEY, b SMALLINT);
```

```
SHOW COLUMNS FROM ints;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |  indices  | is_hidden
+--
   ↪ -----------+-----------+-------------+----------------+-----------------------+-----
   ↪
  a           | INT8      |    false    | NULL           |
     ↪                     | {primary} |   false
  b           | INT2      |    true     | NULL           |
     ↪                     | {}        |   false
(2 rows)
```

```
INSERT INTO ints VALUES (1, 32);
```

```
SELECT * FROM ints;
```

```
  a | b
+---+----+
  1 | 32
(1 row)
```

6.3.3.12.5 支持的转换

INT值可以强制转换为以下数据类型：

Type	Details
DECIMAL	—
FLOAT	如果INT值的大小大于 2 ^ 53，则会失去精度。
BOOL	0 转换为 false；所有其他值都将转换为 true。
DATE	转换为与 Unix 时间（1970 年 1 月 1 日）间隔转换值的日期，转换值不可为负数
TIMESTAMP	转换为与 Unix 时间（1970 年 1 月 1 日）间隔转换值的时间戳，时间戳记中的时间设置为 00:00:00（午夜），转换值不可为负数
INTERVAL	转换为秒。
STRING	—

6.3.3.13 INTERVAL

INTERVAL数据类型存储一个表示时间跨度的值。

6.3.3.13.1 别名

INTERVAL(6)

> 注意: 支持 INTERVAL(6) 作为别名仅出于兼容性目的。此别名不能确定 INTERVAL 值的大小。

6.3.3.13.2 语法

INTERVAL类型的常量值可以使用解释后的文字或用INTERVAL类型注解的字符串或强制为INTERVAL类型的字符串文字来表示。

INTERVAL常数可以使用以下格式表示:

格式	描述
标准 SQL	INTERVAL 'Y-M D H:M:S' Y-M D: 使用单个值仅定义天数; 使用两个值定义年份和月份。值必须是整数。H:M:S: 使用单个值仅定义秒; 使用两个值定义小时和分钟。值可以是整数或浮点数。
ISO 8601	INTERVAL 'P1Y2M3DT4H5M6S'
传统 PostgreSQL	INTERVAL '1 year 2 months 3 days 4 hours 5 minutes 6 seconds'
简略 PostgreSQL	INTERVAL '1 yr 2 mons 3 d 4 hrs 5 mins 6 secs'

支持在INTERVAL值的上下文中使用未解释的字符串文本。

6.3.3.13.3 大小

INTERVAL列最多支持宽度为 24 个字节的值，但是由于元数据因素，总存储大小可能会更大。

间隔在内部存储为月，日和微秒。

6.3.3.13.4 示例

```
CREATE TABLE intervals (a INT PRIMARY KEY, b INTERVAL);
```

```
SHOW COLUMNS FROM intervals;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
    ↪  |  indices  | is_hidden
+--
  ↪ -----------+-----------+-------------+----------------+----------------------+-----
  ↪
  a           | INT8      |    false    | NULL           |
    ↪                     | {primary} |   false
  b           | INTERVAL  |    true     | NULL           |
    ↪                     | {}        |   false
(2 rows)
```

```
INSERT INTO
    intervals
```

```
    VALUES (1, INTERVAL '1 year 2 months 3 days 4 hours 5 minutes 6 seconds'),
           (2, INTERVAL '1-2 3 4:5:6'),
           (3, '1-2 3 4:5:6');
```

```
SELECT * FROM intervals;
```

```
  a |                b
+---+-------------------------------+
  1 | 1 year 2 mons 3 days 04:05:06
  2 | 1 year 2 mons 3 days 04:05:06
  3 | 1 year 2 mons 3 days 04:05:06
(3 rows)
```

6.3.3.13.5 支持的转换

INTERVAL值可以强制转换为以下数据类型：

Type	Details
INT	转换为秒数（秒精度）
DECIMAL	换为秒数（微秒精度）
FLOAT	转换为皮秒数
STRING	转换为如'1 year 2 mons 3 days 04:05:06'的格式
TIME	转换为HH：MM：SS，该时间等于午夜后的时间间隔

6.3.3.14 JSONB

JSONB数据类型将 JSON 数据存储为JSONB值的二进制存储，从而消除了空格，重复键和键顺序。JSONB支持倒排索引。

6.3.3.14.1 别名

JSON是 JSONB的别名。

> 注意：在 PostgreSQL 中，JSONB和JSON是两种不同的数据类型。这里的 JSONB / JSON数据类型与 PostgreSQL 中的JSONB数据类型相似。

6.3.3.14.2 注意事项

- 主键，外键和唯一约束不能在JSONB值上使用。
- 无法在JSONB列上创建标准索引，必须使用倒排索引。

6.3.3.14.3 语法

JSONB数据类型的语法遵循 RFC8259 中指定的格式。JSONB类型的常量值可以使用解释后的文字或带JSONB类型的字符串文字来表示。

JSONB值有六种类型：

- null
- Boolean
- String
- Number (如：decimal，非标准 int64)
- Array (如：有序的JSONB序列)
- Object (如：从字符串到JSONB值的映射)

样例：

```
'{"type": "account creation", "username": "hubble123"}'
```

```
'{"first_name": "Zhang", "status": "Looking for treats", "location" : "Beijing
    ↪ "}'
```

> 注意：如果输入中包含重复的键，则仅保留最后一个值。

6.3.3.14.4 大小

JSONB值的大小是可变的，但建议将值保持在 1MB 以下以确保性能。超过该阈值，写放大和其他考虑因素可能导致性能显着下降。

6.3.3.14.5 JSONB 函数

Function	Description
jsonb_array_elements()	将JSONB数组扩展为一组JSONB值。
jsonb_build_object(...)	从可变参数列表中构建一个JSONB对象，该参数列表在键和值之间交替。
jsonb_each()	将最外面的JSONB对象扩展为一组键值对。
jsonb_object_keys()	返回最外面的JSONB对象中的有序键集合。
jsonb_pretty()	返回给定的JSONB值作为缩进的STRING。请参见下面的示例。

有关受支持的JSONB函数的完整列表。

6.3.3.14.6 JSONB 操作符

Operator	Description	Example
->	访问JSONB字段，返回JSONB值	SELECT '[{"foo":"bar"}]'::JSONB->0->'foo' = ↪ '"bar"'::JSONB;
->>	访问JSONB字段，返回字符串	SELECT '{"foo":"bar"}'::JSONB->>'foo' = ' ↪ bar'::STRING;
@>	测试左侧的JSONB字段是否包含右侧的的JSONB字段。	SELECT ('{"foo": {"baz": 3}, "bar": 2}':: ↪ JSONB @> '{"foo": {"baz":3}}'::JSONB)= ↪ true;

有关受支持的JSONB函数的完整列表。

6.3.3.14.7 示例

创建带有 `JSONB` 列的表

```
CREATE TABLE users (
    profile_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    last_updated TIMESTAMP DEFAULT now(),
    user_profile JSONB
  );
```

```
SHOW COLUMNS FROM users;
```

```
  column_name   | data_type | is_nullable |  column_default    |
     ↪ generation_expression |  indices   | is_hidden
+--------------+-----------+-------------+--------------------+----------------------+----
   ↪
  profile_id    | UUID      |    false    | gen_random_uuid() |
     ↪                      | {primary} |   false
  last_updated | TIMESTAMP |    true     | now()::TIMESTAMP |
     ↪                      | {}        |   false
  user_profile | JSONB     |    true     | NULL              |
     ↪                      | {}        |   false
(3 rows)
```

```
INSERT INTO users (user_profile) VALUES
    ('{"first_name": "Zhang", "last_name": "San", "location": "Beijing", "online
       ↪ " : true, "friends" : 520}'),
    ('{"first_name": "Wang", "status": "Looking for treats", "location" : "
       ↪ ShangHai"}');
```

```
SELECT * FROM users;
```

```
                    profile_id              |          last_updated          |
                       ↪
                       ↪ user_profile
    +--------------------------------------+--------------------------------+------------
       ↪
      8322a77e-84e3-4855-8a7a-f5a4adf5c6da | 2019-03-25 06:27:05.95072+00:00 |
         ↪ {"first_name": "Zhang", "friends": 520, "last_name": "San", "
         ↪ location": "Beijing", "online": true}
      e6af0808-f4b3-4c6d-813b-f59553673711 | 2019-03-25 06:27:05.95072+00:00 |
         ↪ {"first_name": "Wang", "location": "ShangHai", "status": "Looking
         ↪ for treats"}
    (2 rows)
```

检索格式化的 JSONB 数据

要使用易于阅读的格式检索JSONB数据，请使用jsonb_pretty()函数。例如，从您在第一个示例中创建的表中检索数据：

```
SELECT profile_id, last_updated, jsonb_pretty(user_profile) FROM users;
```

```
                 profile_id              |           last_updated           |
                    ↪            jsonb_pretty
    +--------------------------------------+----------------------------------+-----------
      ↪
      1740d746-2b14-41f7-b733-99c64de2de8b | 2020-03-25 06:48:51.240539+00:00 |
         ↪ {
                                           |                                  |
                                              ↪     "first_name": "Zhang",
                                           |                                  |
                                              ↪     "friends": 520,
                                           |                                  |
                                              ↪     "last_name": "San",
                                           |                                  |
                                              ↪     "location": "Beijing",
                                           |                                  |
                                              ↪     "online": true
                                           |                                  |
                                              ↪ }
      47f6134b-7ae7-4078-9c55-811d12905c39 | 2020-03-25 06:48:51.240539+00:00 |
         ↪ {
                                           |                                  |
                                              ↪     "first_name": "Wang",
                                           |                                  |
                                              ↪     "location": "ShangHai",
                                           |                                  |
                                              ↪     "status": "Looking for
                                              ↪ treats"
                                           |                                  |
                                              ↪ }
    (2 rows)
```

从 `JSONB` 值检索特定字段

```
SELECT user_profile->'first_name' as first_name,user_profile->'location' as
   ↪ location FROM users;
```

```
  first_name |  location
+------------+------------+
  "Zhang"    | "Beijing"
  "Wang"     | "ShangHai"
(2 rows)
```

还可以使用 `->>` 运算符将 JSONB 字段值作为STRING值返回：

```
SELECT user_profile->>'first_name' as first_name,user_profile->>'location' as
    ↪ location FROM users;
```

```
  first_name | location
+------------+------------+
  Zhang      | Beijing
  Wang       | ShangHai
(2 rows)
```

创建具有 `JSONB` 列和计算列的表

在此示例中，创建一个具有JSONB列和计算列的表：

```
CREATE TABLE student_profiles (
    id STRING PRIMARY KEY AS (profile->>'id') STORED,
    profile JSONB
);
```

```
INSERT INTO student_profiles (profile) VALUES
    ('{"id": "d1001", "name": "Zhang San", "age": "15", "school": "RENDA", "
        ↪ sports": "none"}'),
    ('{"name": "Wang Wu", "age": "15", "id": "f3003", "school": "SIZHONG", "
        ↪ clubs": "Basketball"}'),
    ('{"name": "Li Si", "school" : "BAZHONG", "id": "t2002", "sports": "Track
        ↪ and Field", "clubs": "Chess"}');
```

```
SELECT * FROM student_profiles;
```

```
   id    |                                                    profile
+-------+-------------------------------------------------------------------------------------
    ↪
  d1001 | {"age": "15", "id": "d1001", "name": "Zhang San", "school": "RENDA", "
      ↪ sports": "none"}
  f3003 | {"age": "15", "clubs": "Basketball", "id": "f3003", "name": "Wang Wu",
      ↪  "school": "SIZHONG"}
  t2002 | {"clubs": "Chess", "id": "t2002", "name": "Li Si", "school": "BAZHONG
      ↪ ", "sports": "Track and Field"}
(3 rows)
```

主键id作为profile列中的字段进行计算。

6.3.3.14.8 支持的转换

JSONB值可以转换为以下数据类型：

- STRING

6.3.3.15 SERIAL

SERIAL并非实际的数据类型，在定义一个表的列时，可以代替实际数据类型。它大约等效于使用带有 DEFAULT 表达式的整数类型，该整数类型在每次求值时都会生成不同的值。该默认表达式进而确保未指定此列的插入将收到自动生成的值，而不是NULL。

> 注意：提供SERIAL仅是为了与 PostgreSQL 兼容。应用程序应使用实际数据类型和合适的DEFAULT ↪ 表达式。
>
> 大部分情况下，建议使用带有gen_random_uuid()函数的UUID数据类型作为默认值，该值生成 128 位值（大于 SERIAL 的最大 64 位），并将它们更均匀地分散在表的所有基础键值范围内。当在索引或主键中使用UUID列时，UUID可以确保多个节点共担负载。

6.3.3.15.1 操作方式

关键字SERIAL在CREATE TABLE时可以识别，并在创建表时自动转换为实际数据类型和DEFAULT表达式。转换结果为数据库内部进行，并且可以使用SHOW CREATE进行查看。

选择的DEFAULT表达式可确保在行插入期间自动为列生成不同的值。这些值不能保证单调递增，有关详细信息，请参阅下面的文档。

SERIAL有三种可能的翻译模式：

模式	描述
rowid (默认)	SERIAL表示默认为unique_rowid()。实际数据类型始终为INT
virtual_sequence	SERIAL创建一个虚拟序列，表示DEFAULT nextval。实际数据类型始终为INT。
sql_sequence	SERIAL创建常规 SQL 序列，表示DEFAULT nextval。实际数据类型取决于SERIAL变体。

模式 `rowid` 和 `virtual_sequence` 的生成值

在rowid和virtual_sequence两种模式下，都会使用unique_rowid()函数自动生成一个值。这将从执行INSERT ↪ 或UPSERT操作的节点的当前时间戳和 ID 生成 64 位整数。从统计上讲，此操作在全球范围内可能是唯一的，除非是极端情况。

另外，由于使用unique_rowid()生成的值不需要节点间的协调，因此当多个 SQL 客户端从不同节点向表写入数据时，它比下面讨论的其他模式sql_sequence快得多。

模式 `sql_sequence` 的生成值

在这种模式下，在指定SERIAL的表的同时，自动创建常规 SQL 序列。

实际数据类型确定如下：

SERIAL 变体	实际数据类型
SERIAL2, SMALLSERIAL	INT2
SERIAL4	INT4
SERIAL	INT
SERIAL8, BIGSERIAL	INT8

每当使用插入或更新插入时都将使用nextval()来递增序列并产生递增的值。

由于 SQL 序列将当前序列值保留在数据库中，因此当多个客户端通过不同节点同时使用序列时，需要进行节点间协调。这可能导致影响性能下降。

因此，应用程序应考虑使用unique_rowid()或gen_random_uuid()，尽可能不使用序列。

6.3.3.15.2 示例

使用 `SERIAL` 自动生成主键

在此示例中，我们创建了一个以SERIAL列作为主键的表，这样我们可以在插入时自动生成唯一的 ID。

```
CREATE TABLE serial (a SERIAL PRIMARY KEY, b STRING, c BOOL);
```

SHOW COLUMNS语句显示SERIAL类型只是INT的别名，默认值为unique_rowid () 。

```
SHOW COLUMNS FROM serial;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |  indices  | is_hidden
+--
   ↪ -----------+-----------+-------------+----------------+----------------------+-----
   ↪
  a           | INT8      |    false    | unique_rowid() |
     ↪                     | {primary} |   false
  b           | STRING    |    true     | NULL           |
     ↪                     | {}        |   false
  c           | BOOL      |    true     | NULL           |
     ↪                     | {}        |   false
(3 rows)
```

当我们在a列中插入空值的行并显示新行时，我们看到每一行在a列中都默认为唯一值。

```
INSERT INTO serial (b,c) VALUES ('red', true), ('yellow', false), ('pink', true)
   ↪ ;
```

```
INSERT INTO serial (a,b,c) VALUES (123, 'white', false);
```

```
SELECT * FROM serial;
```

```
          a           |   b    |   c
+--------------------+--------+-------+
                 123 | white  | false
  540856466654756865 | red    | true
  540856466654789633 | yellow | false
  540856466654822401 | pink   | true
(4 rows)
```

自动增量并非是有顺序的

常见的误解是 PostgreSQL 和 MySQL 中的自动递增类型会生成严格的顺序值。可能存在空白并且不能完全保证连续性：

- 即使未提交插入，每个插入也会将序列增加一个。这意味着自动递增类型可能会在序列中留下空白。
- 两个并发事务可以以与使用序列不同的顺序提交。事务自动提交会使其效果更加明显。

要亲自体验一下，请在 PostgreSQL 中运行以下示例：

1. 创建一个带有SERIAL列的表：

```
CREATE TABLE increment (a SERIAL PRIMARY KEY);
```

1. 运行四个事务以插入行：

```
BEGIN; INSERT INTO increment DEFAULT VALUES; ROLLBACK;
```

```
BEGIN; INSERT INTO increment DEFAULT VALUES; COMMIT;
```

```
BEGIN; INSERT INTO increment DEFAULT VALUES; ROLLBACK;
```

```
BEGIN; INSERT INTO increment DEFAULT VALUES; COMMIT;
```

1. 查看创建的行：

```
SELECT * from increment;
```

```
+---+
| a |
+---+
| 2 |
| 4 |
+---+
```

由于每个插入将 a 列中的序列加 1，因此第一个提交的插入的值为 2，第二个提交的插入的值为 4。如您所见，这些值不是严格顺序的，最后一个值不能准确计算表中的行数。

总之，PostgreSQL 和本数据库中的SERIAL类型以及 MySQL 中的AUTO_INCREMENT类型都表现一致，因为它们不创建严格的序列。与其他数据库相比，本数据库可能会产生更多的空白间隔，但生成这些值的速度会更快。

其他示例

如果同时发生两个事务，则本数据库无法保证 ID 单调递增（即第一次提交小于第二次提交）。下面是三个演示此情况的方案：

案例 1：

- 时间点 1，事务 T1 开始。
- 时间点 2，事务 T2 在同一节点（来自不同的客户端）开始。
- 时间点 3，事务 T1 创建一个SERIAL值x。
- 时间点 3 + 2 微秒，事务 T2 创建一个SERIAL值y。
- 时间点 4，事务 T1 提交。
- 时间点 5，事务 T2 提交。

如果发生这种情况，尽管T1和T2开始和提交时间并不同，但本数据库无法保证x<y还是x>y。在这个特定示例中，甚至可能是因为差异小于 10 微秒导致x=y，并且SERIAL值是根据当前时间的微秒数构造的。

案例 2：

- 时间点 1，事务 T1 开始。
- 时间点 1，事务 T2 在其他节点上的其他地方开始。
- 时间点 2，事务 T1 创建一个SERIAL值x。
- 时间点 3，事务 T2 创建一个SERIAL值y。
- 时间点 3，事务 T2 创建一个SERIAL值y。
- 时间点 4，事务 T1 提交。
- 时间点 4，事务 T2 提交。

如果发生这种情况，本数据库无法保证x<y或x>y。即使事务是同时开始和提交的，两者都可能发生。但是，明确的是x!=y，因为这些值是在不同的节点上生成的。

案例 3：

- 时间点 1，事务 T1 开始。
- 时间点 2，事务 T1 创建一个SERIAL值x。
- 时间点 3，事务 T1 提交。
- 时间点 4，事务 T2 在其他节点上的其他地方开始。
- 时间点 5，事务 T2 创建SERIAL值y。
- 时间点 6，事务 T2 提交。

两个节点的系统时钟之间的差异小于 250 微秒。

如果发生这种情况，本数据库无法保证x<y或x>y。即使事务明显地发生在另一个之后，也可能在两个节点之间存在时钟偏差，并且第二个节点的系统时间设置得比第一个节点稍早。

6.3.3.16 STRING

STRING数据类型存储一串 Unicode 字符。

6.3.3.16.1 别名

STRING的别名

- CHARACTER
- CHAR
- VARCHAR
- TEXT

STRING(N)的别名：

- CHARACTER(n)
- CHARACTER VARYING(n)
- CHAR(n)
- CHAR VARYING(n)
- VARCHAR(n)

6.3.3.16.2 长度

使用 STRING(n)来显示string列的长度，其中n是允许的最大 Unicode 代码点数（通常认为是字符）。

插入字符串时：

- 如果该值超过列的长度限制，则会报错。

- 如果该值被强制转换为具有长度限制的字符串（例如CAST('hello world'AS STRING(5))），则将截断到限制长度。
- 如果该值未达到列的长度限制，则不会添加填充。这适用于STRING(n)及其所有别名。

6.3.3.16.3 语法

STRING类型的值可以使用多种格式表示。

在 SQL Shell 中打印出STRING值时，如果该值不包含特殊字符，则 Shell 使用简单的 SQL 字符串文字格式，否则使用转义格式。

归类

STRING值接受归类，使您可以根据特定于语言和国家/地区的规则对字符串进行排序。

> 注意：当前，您不能在索引或主键中使用排序规则的字符串；这样做会导致数据库崩溃。

6.3.3.16.4 大小

STRING值的大小是可变的，但建议将值保持在 64KB 以下以确保性能。超过该阈值，写放大和其他考虑因素可能导致性能显着下降。

6.3.3.16.5 示例

```
CREATE TABLE strings (a STRING PRIMARY KEY, b STRING(4), c TEXT);
```

```
SHOW COLUMNS FROM strings;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |  indices  | is_hidden
+--
   ↪ -----------+-----------+-------------+----------------+----------------------+-----
   ↪
  a           | STRING    |   false     | NULL           |
     ↪                     | {primary} |   false
  b           | STRING(4) |   true      | NULL           |
     ↪                     | {}        |   false
  c           | STRING    |   true      | NULL           |
     ↪                     | {}        |   false
(3 rows)
```

```
INSERT INTO strings VALUES ('a1b2c3d4', 'e5f6', 'g7h8i9');
```

```
SELECT * FROM strings;
```

```
     a      |  b   |   c
+----------+------+--------+
  a1b2c3d4 | e5f6 | g7h8i9
(1 row)
```

```
CREATE TABLE aliases (a STRING PRIMARY KEY, b VARCHAR, c CHAR);
```

```
SHOW COLUMNS FROM aliases;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |  indices  | is_hidden
+--
   ↪ -----------+-----------+-------------+----------------+-----------------------+-----
   ↪
  a           | STRING    |    false    | NULL           |
     ↪                     | {primary} |   false
  b           | VARCHAR   |    true     | NULL           |
     ↪                     | {}        |   false
  c           | CHAR      |    true     | NULL           |
     ↪                     | {}        |   false
(3 rows)
```

```
INSERT INTO aliases VALUES ('abcd1234', 'ef56', 'g');
```

```
SELECT * FROM aliases;
```

```
     a      |  b   | c
+----------+------+---+
  abcd1234 | ef56 | g
(1 row)
```

6.3.3.16.6 支持的转换

可以将STRING值强制转换为以下任何数据类型：

类型	细节
BOOL	需要支持BOOL字符串类型，如：'true'..
BYTES	需要支持的BYTES字符串格式，例如 b'\141\061\142\062\143\063'
DATE	需要支持DATE字符串类型，如：'2019-01-28'.
DECIMAL	需要支持DECIMAL字符串类型，如：'1.1'.
FLOAT	需要支持FLOAT字符串类型，如：'1.1'.
INET	需要支持INET字符串类型，e.g, '192.168.0.1'.
INT	需要支持INT字符串类型，如：'10'.
INTERVAL	需要支持INTERVAL字符串类型，如：'1h2m3s'.
TIME	需要支持TIME字符串类型，如：'01:22:12' .
TIMESTAMP	需要支持TIMESTAMP字符串类型，如：'2019-01-28 10:10:10.222222'.

STRING与BYTES

尽管在许多情况下，STRING和BYTES似乎都具有相似的行为，但在将它们转换为另一种之前，应先了解它们的细微差别。

STRING将其所有数据视为字符，或更具体地说，将其视为 Unicode 代码点。BYTES将其所有数据视为字节字符串。在实现上的这种差异可能导致行为发生显着不同。例如，让我们采用一个复杂的 Unicode 字符，例如（雪人表情符号）：

```
SELECT length(' '::string);
```

```
  length
+--------+
       1
```

```
SELECT length(' '::bytes);
```

```
  length
+--------+
       3
```

在这种情况下，LENGTH（string）测量字符串中存在的 Unicode 代码点的数量，而LENGTH（bytes）测量存储该值所需的字节数。每个字符（或 Unicode 代码点）可以使用多个字节进行编码，因此两者之间输出存在差异。

将文字转换为STRING与BYTES

通过 SQL 客户端输入的文字将根据类型转换为其他值：

- BYTES在开头对\x赋予特殊含义，并通过将十六进制数字对替换为单个字节来转换其余部分。例如，\xff等效于值为 255 的单个字节。
- STRING没有给\x赋予特殊含义，因此所有字符都被视为不同的 Unicode 代码点。例如，\xff被视为长度为 4（\，x，f和f）的STRING。

6.3.3.17 TIME

TIME数据类型以 UTC 格式存储一天中的时间。

6.3.3.17.1 别名

TIME WITHOUT TIME ZONE

6.3.3.17.2 语法

TIME类型的常量可以使用解释文本表示，或用使用TIME类型注释的字符串文字或强制为TIME类型来表示。

时间的字符串格式为HH:MM:SS.SSSSSS。例如：TIME '08:30:30.000001'。

如果没有歧义，则简单的未注释字符串文字也可以自动解释为TIME类型。请注意，TIME的小数部分是可选的，并且四舍五入到微秒（即小数点后六位）以与 PostgreSQL 协议兼容。

6.3.3.17.3 大小

TIME列最多支持 8 个字节的值，但是由于库元数据的因素，总存储大小可能会更大。

6.3.3.17.4 示例

```
CREATE TABLE time (time_id INT PRIMARY KEY, time_val TIME);
```

```
SHOW COLUMNS FROM time;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |  indices  | is_hidden
+--
   ↪ -----------+-----------+-------------+----------------+----------------------+-----
   ↪
  time_id     | INT8      |    false    | NULL           |
     ↪                      | {primary} |   false
  time_val    | TIME      |    true     | NULL           |
     ↪                      | {}        |   false
(2 rows)
```

```
INSERT INTO time VALUES (1, TIME '08:40:00'), (2, TIME '08:41:40');
```

```
SELECT * FROM time;
```

```
  time_id | time_val
+---------+----------+
        1 | 08:40:00
        2 | 08:41:40
(2 rows)
```

比较TIME值:

```
SELECT (SELECT time_val FROM time WHERE time_id = 1) < (SELECT time_val FROM
   ↪ time WHERE time_id = 2);
```

```
  ?column?
+---------+
    true
(1 row)
```

6.3.3.17.5 支持的转换

可以将TIME值强制转换为以下任何数据类型:

类型	细节
INTERVAL	转换为从午夜（00:00）开始的时间范围
STRING	转换为格式“HH：MM：SS”

6.3.3.18 TIMESTAMP/TIMESTAMPTZ

TIMESTAMP和TIMESTAMPTZ数据类型以 UTC 格式存储日期和时间对。

6.3.3.18.1 变体

TIMESTAMP有两个变形：

- TIMESTAMP以 UTC 形式表示所有TIMESTAMP值。
- TIMESTAMPTZ将TIMESTAMP值从 UTC 转换为客户端的会话时区（除非为该值指定另一个时区）。但是，从概念上讲，TIMESTAMPTZ不存储任何时区数据。

> 注意：默认会话时区为 UTC，这意味着默认情况下TIMESTAMPTZ值以 UTC 显示。

这两个变形之间的区别在于，TIMESTAMPTZ使用客户端的会话时区，而另一个根本不使用。这种情况扩展到类似TIMESTAMPTZ值的函数，如now()和extract()。

6.3.3.18.2 最佳实践

我们建议始终使用TIMESTAMPTZ变体，因为TIMESTAMP有时会忽略会话偏移量时导致的意想不到的问题。但是我们建议您避免为数据库设置会话时间。

6.3.3.18.3 别名

- TIMESTAMP, TIMESTAMP WITHOUT TIME ZONE
- TIMESTAMPTZ, TIMESTAMP WITH TIME ZONE

6.3.3.18.4 语法

TIMESTAMP，TIMESTAMPTZ类型的常量值可以使用解释文本表示，或用TIMESTAMP，TIMESTAMPTZ类型注释字符串文字或强制为TIMESTAMP，TIMESTAMPTZ类型来表示。

TIMESTAMP常量可以使用以下字符串文字格式表示：

格式	示例
Date only	TIMESTAMP '2016-01-25'
Date and Time	TIMESTAMP '2016-01-25 10:10:10.555555'
ISO 8601	TIMESTAMP '2016-01-25T10:10:10.555555'

要表示TIMESTAMPTZ值（时区与 UTC 偏移），请使用以下格式：TIMESTAMPTZ '2019-02-25 10: 10: 10 ↪ .555555-05: 00'

如果明确，简单的未注释字符串文字也可以自动解释为TIMESTAMP或TIMESTAMPTZ类型。

请注意，小数部分是可选的，并且四舍五入到微秒（十进制后的 6 位数），以便与 PostgreSQL 协议兼容。

6.3.3.18.5 大小

TIMESTAMP，TIMESTAMPTZ列最多支持宽度为 12 个字节的值，但是由于元数据因素，总存储大小可能会更大。

6.3.3.18.6 示例

```
CREATE TABLE timestamps (a INT PRIMARY KEY, b TIMESTAMPTZ);
```

```
SHOW COLUMNS FROM timestamps;
```

```
  column_name |  data_type   | is_nullable | column_default |
     ↪ generation_expression |  indices   | is_hidden
+--
   ↪ -----------+-------------+-------------+----------------+-----------------------+---
   ↪
  a           | INT8        |    false    | NULL           |
     ↪                      | {primary} |   false
  b           | TIMESTAMPTZ |    true     | NULL           |
     ↪                      | {}        |   false
(2 rows)
```

```
INSERT INTO timestamps VALUES (1, TIMESTAMPTZ '2019-03-25 10:10:10'), (2,
   ↪ TIMESTAMPTZ '2019-03-25');
```

```
SELECT * FROM timestamps;
```

```
  a |           b
+---+-----------------------+
  1 | 2019-03-25 10:10:10+08
  2 | 2019-03-25 00:00:00+08
(2 rows)
```

6.3.3.18.7 支持的转换

可以将TIMESTAMP值强制转换为以下任何数据类型：

类型	细节
DECIMAL	转换为 Unix 时间至今的秒数（1970 年 1 月 1 日）。
FLOAT	转换为 Unix 时间至今的秒数（1970 年 1 月 1 日）。
TIME	转换为时间戳的时间部分（HH：MM：SS）
INT	转换为 Unix 时间至今的秒数（1970 年 1 月 1 日）。
DATE	--
STRING	--

6.3.3.19 UUID

UUID（通用唯一标识符）数据类型存储一个 128 位的值，该值在空间和时间上都是唯一的。

> 建议：要自动生成唯一的行 ID，建议将带有gen_random_uuid()函数的UUID作为默认值。

6.3.3.19.1 句法

UUID值可以使用以下格式表示：

HUBBLE

格式	描述
标准 RFC4122 指定的格式	连字号分隔的 8、4、4、4、12 个十六进制数字的组。如: `acde070d-8c4c-4f0d-9d8a-162843c10333`
带括号	带括号的标准 RFC4122 指定格式。如: `{acde070d-8c4c-4f0d-9d8a-162843c10333}`
作为 BYTES	指定为字节的UUID值。如: `b'kafef00ddeadbeed'`
UUID 用作 URN	UUID可以用作统一资源名称（URN）。在这种情况下，格式指定为“urn:uuid:”，后跟着标准 RFC4122 指定的格式。如: `urn:uuid:63616665-6630-3064-6465-616462656564`

6.3.3.19.2 大小

一个UUID值的宽度为 128 位，但是由于元数据因素，总存储大小可能会更大。

6.3.3.19.3 示例

创建具有手动输入的 `UUID` 值的表

使用标准 RFC4122 指定的格式创建具有UUID的表

```
CREATE TABLE v (token uuid);
```

```
INSERT INTO v VALUES ('63616665-6630-3064-6465-616462656562');
```

```
SELECT * FROM v;
```

```
                 token
+--------------------------------------+
  63616665-6630-3064-6465-616462656562
(1 row)
```

用大括号以标准 RFC4122 指定的格式创建具有UUID的表

```
INSERT INTO v VALUES ('{63616665-6630-3064-6465-616462656563}');
```

```
SELECT * FROM v;
```

```
                 token
+--------------------------------------+
  63616665-6630-3064-6465-616462656562
  63616665-6630-3064-6465-616462656563
(2 rows)
```

用UUID创建字节格式的表

```
INSERT INTO v VALUES (b'kafef00ddeadbeed');
```

```
SELECT * FROM v;
```

```
+--------------------------------------+
  63616665-6630-3064-6465-616462656562
  63616665-6630-3064-6465-616462656563
  6b616665-6630-3064-6465-616462656564
(3 rows)
```

创建一个表并将UUID用作 URN

```
INSERT INTO v VALUES ('urn:uuid:63616665-6630-3064-6465-616462656564');
```

```
SELECT * FROM v;
```

```
                 token
+--------------------------------------+
  63616665-6630-3064-6465-616462656562
  63616665-6630-3064-6465-616462656563
  6b616665-6630-3064-6465-616462656564
  63616665-6630-3064-6465-616462656564
(4 rows)
```

创建具有自动生成的唯一行 ID 的表

要自动生成唯一的行 ID，请使用带有gen_random_uuid()函数的UUID列作为默认值：

```
CREATE TABLE users (
        id UUID NOT NULL DEFAULT gen_random_uuid(),
        city STRING NOT NULL,
        name STRING NULL,
        address STRING NULL,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
        FAMILY "primary" (id, city, name, address)
);
```

```
INSERT INTO users (name, city) VALUES ('Zhang', 'Beijing'), ('Wang', 'Shanghai')
    ↪ , ('Li', 'Beijing');
```

```
SELECT * FROM users;
```

```
                   id                   |   city   | name  | address
+--------------------------------------+----------+-------+---------+
  5a583e85-a1ce-45bc-a103-ff3490c365e2 | Beijing  | Li    | NULL
  f61dec54-3fc6-489d-baef-de71bd6509b8 | Beijing  | Zhang | NULL
  f92c53f1-0992-47f8-9a95-b43d54895f6e | Shanghai | Wang  | NULL
(3 rows)
```

或者，您可以将BYTES列与uuid_v4()函数一起用作默认值：

```
CREATE TABLE users2 (
        id BYTES DEFAULT uuid_v4(),
```

```
        city STRING NOT NULL,
        name STRING NULL,
        address STRING NULL,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
        FAMILY "primary" (id, city, name, address)
);
```

```
INSERT INTO users2 (name, city) VALUES ('Zhao', 'Beijing'), ('Liu', 'Shanghai'),
    ↪ ('Chen', 'Beijing');
```

```
SELECT * FROM users2;
```

```
                          id                          |    city    | name |
                              ↪ address
+--
    ↪ ----------------------------------------------------+----------+------+----------+
    ↪
  ~T\021\015'\355G\315\251\270\317\020\304\002\020\345 | Beijing  | Zhao | NULL
  \354\3331\374\365\364J'\242\305-\000H\2514\011       | Beijing  | Chen | NULL
  ,`?\233t\235O\361\201\037\353\021\370\321\335G       | Shanghai | Liu  | NULL
(3 rows)
```

无论哪种情况，生成的 ID 都是 128 位，基数足够大，几乎不可能生成重复值。同样，一旦表超出单个键值范围（默认情况下超过 64MB），新的 ID 将散布在表的所有范围内，因此可能散布在不同的节点上。这意味着多个节点将分担负载。

这种方法的缺点是创建的主键在直接查询中可能没有用，这可能需要与另一个表或辅助索引联接。

如果将生成的 ID 存储在相同的键值范围内很重要，则可以显式地或通过SERIAL类型将整数类型与unique_rowid()函数用作默认值：

```
CREATE TABLE users3 (
        id INT DEFAULT unique_rowid(),
        city STRING NOT NULL,
        name STRING NULL,
        address STRING NULL,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
        FAMILY "primary" (id, city, name, address)
);
```

```
INSERT INTO users3 (name, city) VALUES ('Zhang', 'Beijing'), ('Zhao', 'Shanghai'
    ↪ ), ('Li', 'Beijing');
```

```
SELECT * FROM users3;
```

```
          id          |    city    | name   | address
+--------------------+----------+-------+---------+
  541068110770601985 | Beijing  | Zhang | NULL
  541068110770700289 | Beijing  | Li    | NULL
```

```
  541068110770667521 | Shanghai | Zhao   | NULL
(3 rows)
```

在插入或向上插入时，unique_rowid()函数根据执行插入的节点的时间戳和 ID 生成默认值。这样的时间排序的值可能是全局唯一的，除非每个节点每秒生成大量 ID（100,000+）。

6.3.3.19.4 支持的转换

可以将UUID值强制转换为以下任何数据类型：

类型	细节
BYTES	--

6.3.3.20 ENUM

该CREATE TYPE语句在数据库中创建一个新的枚举数据类型。

6.3.3.20.1 所需权限

要创建类型，用户必须具有数据库的CREATE权限。

6.3.3.20.2 参数介绍

参数	详情
type_name	类型的名称。您可以使用数据库和模式名称来限定名称db.typename，但是在创建类型之后，它只能从包含该类型的数据库中引用。
IF NOT EXISTS	仅当数据库中不存在同名类型时才创建新类型；如果确实存在，则不返回错误。
opt_enum_val_list	构成类型枚举集的值列表。

6.3.3.20.3 示例

创建类型

```
create type status as enum ('a', 'b', 'c');
```

创建与类型相关的表，并插入数据

```
create table account (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        balance DECIMAL,
        status status
);
```

```
insert into account(balance,status) values (500.50,'a'), (0.00,'b'), (1.25,'c');
```

查看数据

```
select * from account;
```

```
                    id                    | balance |  status
--------------------------------------+---------+-----------
  3848e36d-ebd4-44c6-8925-8bf24bba957e |  500.50 |   a
  60928059-ef75-47b1-81e3-25ec1fb6ff10 |    0.00 |   b
  71ae151d-99c3-4505-8e33-9cda15fce302 |    1.25 |   c
```

6.3.4 函数

6.3.4.1 特殊的语法形式

语法	相当于	描述
AT TIME ZONE	timezone()	时区
CURRENT_CATALOG	current_catalog()	现在使用的 catalog
COLLATION FOR	pg_collation_for()	排序
CURRENT_DATE	current_date()	现在的日期 (年, 月, 日)
CURRENT_ROLE	current_user()	现在的用户
CURRENT_SCHEMA	current_schema()	现在使用的 schema
CURRENT_TIMESTAMP	current_timestamp()	现在的时间戳
CURRENT_USER	current_user()	现在的用户
CURRENT_DATABASE	current_database()	现在的数据库
EXTRACT(<part> from <value>)	extract("<part>", <value>)	从.. 中抽取部分内容
OVERLAY(<text1> PLACING <text2> from <int1> FOR <int2>)	overlay(<text1>, <text2>, <int1>, <int2>)	替换
OVERLAY(<text1> PLACING <text2> from <int>)	overlay(<text1>, <text2>, <int>)	替换
SESSION_USER	current_user()	session 用户
SUBSTRING(<text> FOR <int1> from <int2>))	substring(<text>, <int2>, <int1>)	截取
SUBSTRING(<text> FOR <int>)	substring(<text>, 1, <int>))	
SUBSTRING(<text> from <int1> FOR <int2>)	substring(<text>, <int1>, <int2>)	
SUBSTRING(<text> from <int>)	substring(<text>, <int>)	
TRIM(<text1> from <text2>)	btrim(<text2>, <text1>)	祛除字符或字符串。
TRIM(<text2>, <text1>)	btrim(<text2>, <text1>)	
TRIM(from <text>)	btrim(<text>)	
TRIM(LEADING <text1> from <text2>)	ltrim(<text2>, <text1>)	
TRIM(LEADING from <text>)	ltrim(<text>)	
TRIM(TRAILING <text1> from <text2>)	rtrim(<text2>, <text1>)	
TRIM(TRAILING from<text>)	rtrim(<text>)	
USER	current_user()	查询使用用户

- 展示当前用户

```
select current_user();
```

```
 current_user
+--------------+
   hubble
```

- 展示当前数据库

```
select current_database();
```

```
  current_database
--------------------
  test
```

- 日期转化为时间戳

```
select extract(epoch from cast('2022-11-13 11:22:10'  as timestamp))::int;
```

```
   extract
+--------------+
1668338530
```

- 时间戳转化为日期

```
select 1668338530::timestamp;
```

```
       timestamp
-----------------------
  2022-11-13 11:22:10
```

- substring()用于字符串截取

从第 1 个位置开始截取，截取 4 个字符，返回结果:hubb

```
select  substring('hubble', 1, 4);
```

```
   substring
+--------------+
   hubb
```

从 8 个位置开始截取，截取到最后一个字符

```
select substring('information', 8);
```

```
   substring
+--------------+
   tion
```

正则表达式截取

HUBBLE

```
select substring('PostgreSQL' from 'gre' );
```

```
   SUBSTRING
+--------------+
   gre
```

反向截取

```
select reverse(substring(reverse('PostgreSQL'),1,2));
```

```
   reverse
+--------------+
   QL
```

- position()，子串在一字符串中的位置

```
select position('om' in 'Thomas');
```

```
  position
+--------------+
   3
```

- 获得客户端编码

```
select pg_client_encoding();
```

```
pg_client_encoding
+--------------+
   UTF8
```

- 利用正则表达式对字符串进行替换

```
select regexp_replace('Thomas', '.[mN]a.', 'M');
```

```
regexp_replace
+--------------+
   ThM
```

- 重复字符串 (指定次数)

```
select repeat('hb', 3);
```

```
    repeat
+--------------+
   hbhbhb
```

- 默认为去除空白字符，当然可以自己指定

```
select trim(both 'x' from 'xTomxx');
```

HUBBLE

```
  btrim
+--------------+
  Tom
```

```
select trim('    xTomxxn     ');
```

```
  btrim
+--------------+
  xTomxxn
```

6.3.4.2 条件运算符和类函数运算符

语法	描述
ANNOTATE_TYPE(...)	显式输入表达式
ARRAY(...)	子查询结果转换为数组
ARRAY[...]	将标量表达式转换为数组
CAST(...)	类型转换
COALESCE(...)	第一个非 NULL 短路表达式
EXISTS(...)	子查询结果的存在性检验
IF(...)	条件评估
IFNULL(...)	COALESCE限制为两个操作数的别名
NULLIF(...)	NULL有条件地返回
ROW(...)	元组构造函数

语法:

ANNOTATE_TYPE()

```
<expr>:::<type>
ANNOTATE_TYPE(<expr>, <type>)
```

计算给定的表达式，要求表达式具有给定的类型。如果表达式没有给定的类型，则返回错误。

类型注释对于指导数值的算术特别有用。

例如:

```
select (1 / 0):::FLOAT;
```

```
pq: division by zero
```

句法:

```
ARRAY( ... subquery ... )
```

计算子查询并将其结果转换为数组。任何选择查询都可以用作子查询。

语法:

```
ARRAY[ <expr>, <expr>, ... ]
```

评估为包含指定值的数组。

例如：

```
select ARRAY[1,2,3] AS a;
```

```
+---------+
|    a    |
+---------+
| {1,2,3} |
+---------+
```

语法：

```
<expr> :: <type>
CAST (<expr> AS <type>)
```

计算表达式并将结果值转换为指定的类型。如果转换无效，将报告错误。

例如：

```
CAST(now() AS DATE)
```

语法：

```
<expr> COLLATE <collation>
```

计算表达式并将其结果转换为具有指定归类的归类字符串。

例如：

```
'a' COLLATE de
```

语法：

```
EXISTS ( ... subquery ... )
NOT EXISTS ( ... subquery ... )
```

评估子查询，然后返回TRUE或FALSE取决于子查询是否返回任何行（用于EXISTS）或不返回任何行（用于NOT
↪ EXISTS）。任何选择查询都可以用作子查询。

语法：

```
NULLIF ( <expr1>, <expr2> )
```

相当于：IF (<expr1> = <expr2>, NULL, <expr1>)

语法：

```
IFNULL ( <expr1>, <expr2> )
COALESCE ( <expr1> [, <expr2> [, <expr3> ] ...] )
```

COALESCE首先计算第一个表达式。如果其值不是 NULL，则直接返回其值。否则，它将返回COALESCE对其余表达式应用的结果。如果所有表达式均为NULL，NULL则返回。

不评估第一个非空参数右边的参数。

IFNULL(a, b)等同于COALESCE(a, b)

语法:

```
(<expr>, <expr>, ...)
ROW (<expr>, <expr>, ...)
```

评估为包含提供的表达式值的元组。

例如:

```
select ('x', 123, 12.3) AS a;
```

```
+----------------+
|       a        |
+----------------+
| ('x',123,12.3) |
+----------------+
```

从值推断结果元组的数据类型。元组中的每个位置可以具有不同的数据类型。

6.3.4.3 数组函数

函数	说明	返回值类型
array_append(array: bool[], elem: bool)	将 elem 附加到数组中，返回结果。	bool[]
array_append(array: bytes[], elem: bytes)	将 elem 附加到数组中，返回结果。	bytes[]
array_append(array: date[], elem: date)	将 elem 附加到数组中，返回结果。	date[]
array_append(array: decimal[], elem: decimal)	将 elem 附加到数组中，返回结果。	decimal[]
array_append(array: float[], elem: float)	将 elem 附加到数组中，返回结果。	float[]
array_append(array: inet[], elem: inet)	将 elem 附加到数组中，返回结果。	inet[]
array_append(array: int[], elem: int)	将 elem 附加到数组中，返回结果。	int[]
array_append(array: interval[], elem: interval)	将 elem 附加到数组中，返回结果。	interval[]
array_append(array: string[], elem: string)	将 elem 附加到数组中，返回结果。	string[]
array_append(array: time[], elem: time)	将 elem 附加到数组中，返回结果。	time[]
array_append(array: timestamp[], elem: timestamp)	将 elem 附加到数组中，返回结果。	timestamp[]
array_append(array: timestamptz[], elem : timestamptz)	将 elem 附加到数组中，返回结果。	timestamptz[]
array_append(array: uuid[], elem: uuid)	将 elem 附加到数组中，返回结果。	uuid[]
array_append(array: varbit[], elem: varbit)	将 elem 附加到数组中，返回结果。	varbit[]
array_cat(left: bool[], right: bool[])	追加两个数组。	bool[]

函数	说明	返回值类型
array_cat(left: bytes[], right: bytes ↪ [])	追加两个数组。	bytes[]
array_cat(left: date[], right: date[])	追加两个数组。	date[]
array_cat(left: decimal[], right: ↪ decimal[])	追加两个数组。	decimal[]
array_cat(left: float[], right: float ↪ [])	追加两个数组。	float[]
array_cat(left: inet[], right: inet[])	追加两个数组。	inet[]
array_cat(left: int[], right: int[])	追加两个数组。	int[]
array_cat(left: interval[], right: ↪ interval[])	追加两个数组。	interval[]
array_cat(left: string[], right: string ↪ [])	追加两个数组。	string[]
array_cat(left: time[], right: time[])	追加两个数组。	time[]
array_cat(left: timestamp[], right: ↪ timestamp[])	追加两个数组。	timestamp[]
array_cat(left: timestamptz[], right: ↪ timestamptz[])	追加两个数组。	timestamptz[]
array_cat(left: uuid[], right: uuid[])	追加两个数组。	uuid[]
array_cat(left: varbit[], right: varbit ↪ [])	追加两个数组。	varbit[]
array_length(input: anyelement[], ↪ array_dimension: int)	计算input提供的上的长度array_dimension。但是，由于目前尚不支持多维数组，因此唯一支持的array_dimension是 1。	int
array_lower(input: anyelement[], ↪ array_dimension: int)	计算input提供的上的最小值array_dimension。但是，由于目前尚不支持多维数组，因此唯一支持的array_dimension是 1。	int
array_position(array: bool[], elem: ↪ bool)	返回数组中elem 第一次出现的索引。	int
array_position(array: bytes[], elem: ↪ bytes)	返回数组中elem 第一次出现的索引。	int
array_position(array: date[], elem: ↪ date)	返回数组中elem 第一次出现的索引。	int
array_position(array: decimal[], elem: ↪ decimal)	返回数组中elem 第一次出现的索引。	int
array_position(array: float[], elem: ↪ float)	返回数组中elem 第一次出现的索引。	int
array_position(array: inet[], elem: ↪ inet)	返回数组中elem 第一次出现的索引。	int
array_position(array: int[], elem: int)	返回数组中elem 第一次出现的索引。	int
array_position(array: interval[], elem: ↪ interval)	返回数组中elem 第一次出现的索引。	int

函数	说明	返回值类型
array_position(array: string[], elem: ↪ string)	返回数组中elem 第一次出现的索引。	int
array_position(array: time[], elem: ↪ time)	返回数组中elem 第一次出现的索引。	int
array_position(array: timestamp[], elem ↪ : timestamp)	返回数组中elem 第一次出现的索引。	int
array_position(array: timestamptz[], ↪ elem: timestamptz)	返回数组中elem 第一次出现的索引。	int
array_position(array: uuid[], elem: ↪ uuid)	返回数组中elem 第一次出现的索引。	int
array_position(array: oid[], elem: oid)	返回数组中elem 第一次出现的索引。	int
array_position(array: varbit[], elem: ↪ varbit)	返回数组中elem 第一次出现的索引。	int
array_positions(array: bool[], elem: ↪ bool)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: bytes[], elem: ↪ bytes)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: date[], elem: ↪ date)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: decimal[], elem: ↪ decimal)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: float[], elem: ↪ float)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: inet[], elem: ↪ inet)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: int[], elem: int ↪)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: interval[], elem ↪ : interval)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: string[], elem: ↪ string)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: time[], elem: ↪ time)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: timestamp[], ↪ elem: timestamp)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: timestamptz[], ↪ elem: timestamptz)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: uuid[], elem: ↪ uuid)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: oid[], elem: oid ↪)	返回所有elem 在数组中出现的索引的数组。	int[]
array_positions(array: varbit[], elem: ↪ varbit)	返回所有elem 在数组中出现的索引的数组。	int[]

HUBBLE

函数	说明	返回值类型
array_prepend(elem: bool, array: bool ↪ [])	将elem 加入数组，返回结果。	bool[]
array_prepend(elem: bytes, array: bytes ↪ [])	将elem 加入数组，返回结果。	bytes[]
array_prepend(elem: date, array: date ↪ [])	将elem 加入数组，返回结果。	date[]
array_prepend(elem: decimal, array: ↪ decimal[])	将elem 加入数组，返回结果。	decimal[]
array_prepend(elem: float, array: float ↪ [])	将elem 加入数组，返回结果。	float[]
array_prepend(elem: inet, array: inet ↪ [])	将elem 加入数组，返回结果。	inet[]
array_prepend(elem: int, array: int[])	将elem 加入数组，返回结果。	int[]
array_prepend(elem: interval, array: ↪ interval[])	将elem 加入数组，返回结果。	interval[]
array_prepend(elem: string, array: ↪ string[])	将elem 加入数组，返回结果。	string[]
array_prepend(elem: time, array: time ↪ [])	将elem 加入数组，返回结果。	time[]
array_prepend(elem: timestamp, array: ↪ timestamp[])	将elem 加入数组，返回结果。	timestamp[]
array_prepend(elem: timestamptz, array: ↪ timestamptz[])	将elem 加入数组，返回结果。	timestamptz[]
array_prepend(elem: uuid, array: uuid ↪ [])	将elem 加入数组，返回结果。	uuid[]
array_prepend(elem: varbit, array: ↪ varbit[])	将elem 加入数组，返回结果。	varbit[]
array_remove(array: bool[], elem: bool)	从数组中删除所有等于elem的元素。	bool[]
array_remove(array: bytes[], elem: ↪ bytes)	从数组中删除所有等于elem的元素。	bytes[]
array_remove(array: date[], elem: date)	从数组中删除所有等于elem的元素。	date[]
array_remove(array: decimal[], elem: ↪ decimal)	从数组中删除所有等于elem的元素。	decimal[]
array_remove(array: float[], elem: ↪ float)	从数组中删除所有等于elem的元素。	float[]
array_remove(array: inet[], elem: inet)	从数组中删除所有等于elem的元素。	inet[]
array_remove(array: int[], elem: int)→ ↪ int[]	从数组中删除所有等于elem的元素。	int[]
array_remove(array: interval[], elem: ↪ interval)	从数组中删除所有等于elem的元素。	interval[]
array_remove(array: string[], elem: ↪ string)	从数组中删除所有等于elem的元素。	string[]
array_remove(array: time[], elem: time)	从数组中删除所有等于elem的元素。	time[]

HUBBLE

函数	说明	返回值类型
array_remove(array: timestamp[], elem: timestamp)	从数组中删除所有等于elem的元素。	timestamp[]
array_remove(array: timestamptz[], elem: timestamptz)	从数组中删除所有等于elem的元素。	timestamptz[]
array_remove(array: uuid[], elem: uuid)	从数组中删除所有等于elem的元素。	uuid[]
array_remove(array: varbit[], elem: varbit)	从数组中删除所有等于elem的元素。	varbit[]
array_replace(array: bool[], toreplace: bool, replacewith: bool)	用 replacewith 替换数组中出现的所有 toreplace。	bool[]
array_replace(array: bytes[], toreplace: bytes, replacewith: bytes)	用 replacewith 替换数组中出现的所有 toreplace。	bytes[]
array_replace(array: date[], toreplace: date, replacewith: date)	用 replacewith 替换数组中出现的所有 toreplace。	date[]
array_replace(array: decimal[], toreplace: decimal, replacewith: decimal)	用 replacewith 替换数组中出现的所有 toreplace。	decimal[]
array_replace(array: float[], toreplace: float, replacewith: float)	用 replacewith 替换数组中出现的所有 toreplace。	float[]
array_replace(array: inet[], toreplace: inet, replacewith: inet)	用 replacewith 替换数组中出现的所有 toreplace。	inet[]
array_replace(array: int[], toreplace: int, replacewith: int)	用 replacewith 替换数组中出现的所有 toreplace。	int[]
array_replace(array: interval[], toreplace: interval, replacewith: interval)	用 replacewith 替换数组中出现的所有 toreplace。	interval[]
array_replace(array: string[], toreplace: string, replacewith: string)	用 replacewith 替换数组中出现的所有 toreplace。	string[]
array_replace(array: time[], toreplace: time, replacewith: time)	用 replacewith 替换数组中出现的所有 toreplace。	time[]
array_replace(array: timestamp[], toreplace: timestamp, replacewith: timestamp)	用 replacewith 替换数组中出现的所有 toreplace。	timestamp[]
array_replace(array: timestamptz[], toreplace: timestamptz, replacewith: timestamptz)	用 replacewith 替换数组中出现的所有 toreplace。	timestamptz[]
array_replace(array: uuid[], toreplace: uuid, replacewith: uuid)	用 replacewith 替换数组中出现的所有 toreplace。	uuid[]
array_replace(array: varbit[], toreplace: varbit, replacewith: varbit)	用 replacewith 替换数组中出现的所有 toreplace。	varbit[]
array_to_string(input: anyelement[], delim: string)	用定界符将数组连接到字符串中。	string

函数	说明	返回值类型
array_to_string(input: anyelement[], ↪ delimiter: string, null: string)	用分隔符将数组连接到字符串中，用空字符串替换 NULL。	string
array_upper(input: anyelement[], ↪ array_dimension: int)	计算input提供的上的最大值array_dimension。但是，由于目前尚不支持多维数组，因此唯一支持的array_dimension是 1。	int
string_to_array(str: string, delimiter: ↪ string)	将字符串拆分为分隔符上的组件。	string[]
string_to_array(str: string, delimiter: ↪ string, null: string)	使用指定的字符串将字符串拆分为定界符上的组件，以将其视为 NULL。	string[]

- 利用正则表达式将字符串分割成数组

```
select regexp_split_to_array('hello world', E'\\s+');
```

```
  regexp_split_to_array
+----------------------------+
      {hello,world}
```

数据示例

```
create table test_array (a STRING[]);
insert into  test_array values (ARRAY['Monday', 'Tuesday', 'Wednesday']);
select * from test_array;
```

```
              a
+----------------------------+
  {Monday,Tuesday,Wednesday}
```

- regexp_split_to_table

```
select  regexp_split_to_table('11,22,33',',');
```

```
  regexp_split_to_table
-------------------------
  11
  22
  33
```

- append(array: x[], elem: y)→ bool[]

数组拼接单个元素

```
select array_append(a,'Thursday') from test_array;
```

```
             array_append
+-------------------------------------+
```

HUBBLE

```
  {Monday,Tuesday,Wednesday,Thursday}
```

- array_cat(array: x[], array y[])

两个数组拼接

```
select array_cat(a,ARRAY['Thursday', 'Friday', 'Saturday']) from test_array;
```

```
                        array_cat
+-----------------------------------------------------+
  {Monday,Tuesday,Wednesday,Thursday,Friday,Saturday}
```

- array_length(array:x[],1)

数组大小，第二个参数暂时只支持 1 个间隔计算。

```
select array_length(a,1) from test_array;
```

```
  array_length
+--------------+
          3
```

- array_lower(array:x[],1)

返回指定的数组维数的下界，第二个参数暂时只支持 1 个间隔计算。

```
select array_lower(a,1) from test_array;
```

```
  array_lower
+-------------+
          1
```

- array_upper(array:x[],1)

返回指定数组维数的上界，第二个参数暂时只支持 1 个间隔计算。

```
select array_upper(a,1) from test_array;
```

```
  array_upper
+-------------+
          3
```

- array_position(array:x[],y)

返回元素的位置

```
select array_position(a,'Tuesday') from test_array;
```

```
  array_position
+----------------+
          2
```

HUBBLE

- array_prepend(y,array:x[])

数组之前插入元素

```
select array_prepend('Thursday',a) from test_array;
```

```
              array_prepend
+-------------------------------------+
  {Thursday,Monday,Tuesday,Wednesday}
```

- array_remove(array:x[],y)

数组中删除元素

```
select array_remove(a,'Tuesday') from test_array;
```

```
     array_remove
+--------------------+
  {Monday,Wednesday}
```

- array_replace(array:x[],y,z)

替换数组中的元素y为z

```
select array_replace(a,'Tuesday','Thursday') from test_array;
```

```
         array_replace
+-----------------------------+
  {Monday,Thursday,Wednesday}
```

- array_to_string(array:x[],delim)

数组转字符串，按delim分隔

```
select array_to_string(a,',') from test_array;
```

```
      array_to_string
+--------------------------+
  Monday,Tuesday,Wednesday
```

- array_to_string(array:x[],delim,y)

数组转字符串，按delim分隔，y代替null，即保留null在字符串中的位置。

```
select array_to_string(ARRAY['Thursday', 'Friday', 'Saturday',null,'Friday'],','
    ↪ );
```

```
          array_to_string
+---------------------------------+
  Thursday,Friday,Saturday,Friday
```

```
select array_to_string(ARRAY['Thursday', 'Friday', 'Saturday',null,'Friday'],','
    ↪ ,'y');
```

```
          array_to_string
+----------------------------------+
  Thursday,Friday,Saturday,y,Friday
```

- string_to_array(str:string,delim)

字符串转数组，按分隔符拆分

```
select string_to_array('Thursday,Friday,Saturday,Friday',',');
```

```
           string_to_array
+-----------------------------------+
  {Thursday,Friday,Saturday,Friday}
```

- string_to_array(str:string,delim,y)

字符串转数组，按分隔符拆分，将y视为null

```
select string_to_array('Thursday,Friday,Saturday,y,Friday',',','y');
```

```
             string_to_array
+----------------------------------------+
  {Thursday,Friday,Saturday,NULL,Friday}
```

6.3.4.4 比较函数

函数	说明	返回值类型
greatest(anyelement...)	返回具有最大值的元素。	anyelement
least(anyelement...)	返回具有最小值的元素。	anyelement

- greatest(anyelement...)

取元素中的最大值

```
select greatest(1,10,100,15);
```

```
  greatest
+----------+
     100
```

- least(anyelement...)

取元素中的最小值

```
select least(1,2,10,100,122);
```

```
  least
+-------+
      1
```

HUBBLE

6.3.4.5 时间函数

函数	说明	返回值类型
age(end: timestamptz, begin: timestamptz ↪)	计算begin和end之间的时间间隔。	interval
age(val: timestamptz)	计算val与当前时间之间的间隔。	interval
clock_timestamp()	返回一个集群节点上的当前系统时间。	timestamp
current_date()	返回当前事务的日期。该值基于事务开始时选择的时间戳记，并且在整个事务中保持不变。此时间戳与并发事务的提交顺序没有关系。	date
current_timestamp()	返回当前事务的时间。该值基于事务开始时选择的时间戳记，并且在整个事务中保持不变。此时间戳与并发事务的提交顺序没有关系。	date
current_timestamp()	返回当前事务的时间。该值基于事务开始时选择的时间戳记，并且在整个事务中保持不变。此时间戳与并发事务的提交顺序没有关系。	timestamp
date_trunc(element: string, input: time) ↪ → interval	input精确到截断element。将所有不重要的字段设置element为零。兼容元素：时，分，秒，毫秒，微秒。	interval
date_trunc(element: string, input: ↪ timestamp)→ timestamp	input精确到截断element。将所有不重要的字段设置element为零（对于日和月，则为一）兼容的元素：年，季，月，周，小时，分钟，秒，毫秒，微秒。	timestamp
date_trunc(element: string, input: ↪ timestamptz)	input精确到截断element。将所有不重要的字段设置element为零（对于日和月，则为一）兼容的元素：年，季，月，周，小时，分钟，秒，毫秒，微秒。	timestamptz
experimental_follower_read_timestamp()	返回一个时间戳，很可能可以安全地针对跟随者副本执行。此功能旨在与 AS OF SYSTEM TIME 子句一起使用，以对最近的时间执行历史读取，但时间要足够长，以便针对给定范围内的当前租户（相对于最近的租户）执行读取操作。请注意，此功能需要 CCL 发行版上的企业许可证才能无错误地返回。	timestamptz
experimental_strftime(input: date, ↪ extract_format: string)	从中input提取和格式化extract_format使用标准strftime符号标识的时间（尽管并非所有格式都受支持）。	string
experimental_strftime(input: timestamp, ↪ extract_format: string)	从中input提取和格式化extract_format使用标准strftime符号标识的时间（尽管并非所有格式都受支持）。	string
experimental_strftime(input: timestamptz ↪ , extract_format: string)	从中input提取和格式化extract_format使用标准strftime符号标识的时间（尽管并非所有格式都受支持）。	string
extract(element: string, input: date)	element从中提取input。兼容的元素：年，季度，月，周，星期几，年日，小时，分钟，秒，毫秒，微秒	int

函数	说明	返回值类型
extract(element: string, input: time)	element从中提取input。兼容的元素：小时，分钟，秒，毫秒，微秒	int
extract(element: string, input: timestamp)	element从中提取input。兼容的元素：年，季度，月，周，星期几，年日，小时，分钟，秒，毫秒，微秒	int
extract(element: string, input: timestamptz)	element从中提取input。兼容的元素：年，季度，月，周，星期几，年日，小时，分钟，秒，毫秒，微秒	int
now()	返回当前交易的时间。该值基于事务开始时选择的时间戳记，并且在整个事务中保持不变。此时间戳与并发事务的提交顺序没有关系。	date
now()	返回当前交易的时间。该值基于事务开始时选择的时间戳记，并且在整个事务中保持不变。此时间戳与并发事务的提交顺序没有关系。	timestamp
statement_timestamp()	返回当前语句的开始时间。	timestamp
timezone(timestamp: timestamp, timezone: string)	将不带时区的给定时间戳视为位于指定时区中	timestamp
transaction_timestamp()	返回当前交易的时间。该值基于事务开始时选择的时间戳记，并且在整个事务中保持不变。此时间戳与并发事务的提交顺序没有关系。	date
transaction_timestamp()	返回当前交易的时间。该值基于事务开始时选择的时间戳记，并且在整个事务中保持不变。此时间戳与并发事务的提交顺序没有关系。	timestamp

案例:

- age(end: timestamptz, begin: timestamptz)

计算end和begin之间的时间间隔，结果体现为前值减去后值

```
select age(timestamptz '2022-03-26 14:22:10+08:00',timestamptz '2020-02-26
    ↪ 10:10:10-05:00');
```

```
            age
+---------------------------+
  2 years 28 days 15:12:00
```

- age(val: timestamptz)

与当前时间的间隔

```
select age(timestamptz '2021-11-26 14:22:10+08:00');
```

```
                age
+--------------------------------+
  10 mons 14 days 23:57:43.199154
```

- current_date()

展示当前时间 (年月日)

```
select current_date();
```

```
         current_date
+-----------------------------+
          2022-10-11
```

- localtime()

展示当前时间 (时分秒)

```
select date_trunc('second',localtime());
```

```
    localtime
+---------------+
    17:26:42
```

- localtimestamp 显示时间 (年月日时分秒)

```
select date_trunc('second',localtimestamp());
```

```
    localtimestamp
--------------------------
  2023-04-19 15:33:33
```

- current_timestamp()或者now()

展示当前完整时间

```
select current_timestamp();
```

```
       current_timestamp()
+-----------------------------+
 2022-10-11 14:28:42.693 +0800
```

或者

```
select now();
```

```
            now()
+-----------------------------+
 2022-10-11 14:32:38.070 +0800
```

- date_trunc(x,input:date)

截断时间，后边清零，x(year, quarter, month, week, hour, minute, second, millisecond, microsecond)

```
select date_trunc('day',timestamptz '2022-11-12 14:22:10+08:00')  as  date_day;
```

```
         date_day
+---------------------------+
 2022-11-12 00:00:00+08
(1 row)
```

```
select date_trunc('month',timestamptz '2022-11-12 14:22:10+08:00') as date_month
    ↪ ;
```

```
         date_month
+---------------------------+
 2022-11-01 00:00:00+08
(1 row)
```

- extract(x,input:date)

input中　提　取x(year, quarter, month, week, dayofweek, hour, minute, second, millisecond,
↪ microsecond)

提取日期中的月份

```
select extract('month',timestamptz '2022-10-11 16:22:10+08:00');
```

```
  extract
+---------+
    10
```

根据日期提取今天是星期几，若返回 0，则代表星期日

```
select extract('dayofweek',timestamptz '2022-10-16 16:22:10+08:00') as dayinweek
    ↪ ;
```

```
 dayinweek
+---------+
    0
```

```
select extract('dayofweek',timestamptz '2022-10-11 16:22:10+08:00') as dayinweek
    ↪ ;
```

```
 dayinweek
+---------+
    2
```

- 获取当前时间是在年中的第多少周

```
select extract('week',timestamptz '2022-10-16 16:22:10+08:00') as weekofyear;
```

```
   weekofyear
+------------+
      41
```

- 日期转字符串

```
select experimental_strftime(now(),'%Y-%m-%d %H:%M:%S');
```

```
  experimental_strftime
-------------------------
  2023-03-08 16:14:31
(1 row)
```

或者

```
select to_char(now()::timestamp);
```

```
           to_char
------------------------------
  2023-10-25 16:30:30.672578
```

- 字符串转日期

```
select cast('2023-03-08 16:14:31' as timestamp);
```

```
  timestamp
--------------------------
  2023-03-08 16:14:31
```

或者

```
select experimental_strptime('2023-10-26 13:44:01','%Y-%m-%d %H:%M:%S');
```

```
  experimental_strptime
--------------------------
  2023-10-26 21:44:01+08
(1 row)
```

- 时间的增减

增加一秒

```
select  now(), now() + interval '1s' as newnow;
```

```
           now()                        |           newnow
----------------------------------------+-------------------------------
2022-10-11 15:47:23.380 +0800           | 2022-10-11 15:47:24.380 +0800
```

增加一天

```
select now(), now() + interval '1 day' as newnow;
```

```
                now                |             newnow
-----------------------------------+-----------------------------------
  2022-10-18 11:06:25.522553+08 | 2022-10-19 11:06:25.522553+08
```

- 获取本月的最后一天的日期

```
select (date_trunc('month',now()) + interval '1 month' + interval '-1 day')::
    ↪ DATE;
```

```
           date
--------------------------
       2022-10-31
```

- 或取当月最后一天的时间戳

```
select (date_trunc('month',now()) + interval '1 month' + interval '-1 second')::
    ↪ timestamp;
```

```
          date
--------------------------
  2022-10-31 23:59:59
```

6.3.4.6 ID 函数

函数	说明	返回值类型
gen_random_uuid()	生成随机UUID并将其作为UUID类型的值返回。	uuid
unique_rowid()	如果未为表定义主键，则返回用于生成唯一行 ID 的唯一 ID。该值是插入时间戳记与执行该语句的节点的 ID 的组合，从而确保此组合在全局上是唯一的。但是，可能存在差距，不能完全保证顺序。	int

- gen_random_uuid()

生成随机uuid

```
select gen_random_uuid();
```

```
            gen_random_uuid
+--------------------------------------+
  9c5b836c-486b-46ed-acb3-701c093f59fa
```

- unique_rowid()

```
select unique_rowid();
```

```
     unique_rowid
+--------------------+
  514553266254184454
```

生成唯一键，保证全局唯一，可用作主键。但由于集群性质，不能完全保证排序。

6.3.4.7 数学类函数

函数	返回类型	参数类型	说明
abs	float	float	计算绝对值
abs	decimal	decimal	计算绝对值
abs	int	int	计算绝对值
acos	float	float	计算反余弦
asin	float	float	计算反正弦
atan	float	float	计算反正切
atan2	float	float，float	计算入参的相除的反正切
cbrt	float	float	计算立方根
ceil	decimal	decimal	计算最小整数
ceil	float	float	计算最小整数
cos	float	float	计算余弦值
cot	float	float	计算余切
crc32c	int	bytes	compute CRC(Castagnoli polynomial.)
crc32c	int	string	compute CRC(Castagnoli polynomial.)
crc32ieee	int	bytes	compute CRC(IEEE polynomial)
crc32ieee	int	string	compute CRC(IEEE polynomial)
degrees	float	float	将弧度值转换为度值
div	decimal	decimal，decimal	计算 x/y 的整数商
div	float	float，float	计算 x/y 的整数商
div	int	int，int	计算 x/y 的整数商
exp	float	float	e 的幂次方
floor	decimal	decimal	计算不大于入参的最大整数
floor	float	float	计算不大于入参的最大整数
fnv32	int	bytes	计算 32 位的 FNV-1 hash 值
fnv32	int	string	计算 32 位的 FNV-1 hash 值
fnv32a	int	bytes	计算 32 位的 FNV-1a hash 值
fnv32a	int	string	计算 32 位的 FNV-1a hash 值
fnv64	int	bytes	计算 64 位的 FNV-1 hash 值
fnv64	int	string	计算 64 位的 FNV-1 hash 值
fnv64a	int	bytes	计算 64 位的 FNV-1a hash 值
fnv64a	int	string	计算 64 位的 FNV-1a hash 值
isnan	bool	decimal	判断是否为 NaN
isnan	bool	float	判断是否为 NaN
ln	decimal	decimal	计算自然对数
ln	float	float	计算自然对数
log	decimal	decimal	计算以 10 为底的对数
log	float	float	计算以 10 为底的对数
mod	decimal	decimal，decimal	计算余数
mod	float	float，float	计算余数
mod	int	int，int	计算余数
pi	float		3.141592653589793
pow	decimal	decimal，decimal	x^y
pow	float	float，float	x^y
pow	int	int,int	x^y
radians	float	float	度值转换为弧度值

函数	返回类型	参数类型	说明
random()	float	float	0 与 1 之间的随机数
round	decimal	decimal,int	四舍五入到给定的小数位数
round	float	float,int	四舍五入到给定的小数位数
round	decimal	decimal	四舍五入取整数
round	float	float	四舍五入取整数
sign	decimal	decimal	符号函数
sign	float	float	符号函数
sign	int	int	符号函数
sin	float	float	计算正弦
sqrt	decimal	decimal	计算平方根
sqrt	float	float	计算平方根
tan	float	float	计算正切
trunc	decimal	decimal	截断小数
trunc	float	float	截断小数

- abs

绝对值

```
select abs(-11.19);
```

```
   abs
---------
  11.19
```

- cbrt

```
select cbrt(27.0);
```

```
          cbrt
--------------------------
  3.0000000000000000000
```

- ceil

向上取整

```
select ceil(23.7);
```

```
  ceil
--------
   24
```

```
select ceiling(-95.3);
```

```
  ceiling
-----------
      -95
```

- degrees

把弧度转为角度

```
select degrees(0.5);
```

```
       degrees
---------------------
  28.64788975654116
```

- div(y/x)

y/x的整数商

```
select div(9,4);
```

```
  div
-------
    2
```

- exp

指数 (e 的幂次方)

```
select exp(1.0);
```

```
           exp
--------------------------
  2.7182818284590452354
```

- floor

不大于参数的最近的整数

```
select floor(-42.8);
```

```
  floor
---------
    -43
```

- ln

计算自然对数

```
select ln(2.0);
```

```
            ln
---------------------------
  0.69314718055994530942
```

- log

以 10 为底的对数

HUBBLE

```
select log(100.0);
```

```
           log
-------------------------
  2.0000000000000000000
```

以 2 为底的对数

```
select log(2.0, 64.0);
```

```
           log
-------------------------
  6.0000000000000000000
```

- mod

计算余数

```
 select mod(8,3);
```

```
  mod
-------
    2
```

- pi

```
select pi();
```

```
         pi
---------------------
  3.141592653589793
```

- pow

a的b次方

```
select pow(2,3);
```

```
 pow
-------
    8
```

- random

0-1的随机数

```
select random();
```

```
        random
----------------------
  0.7155664026177302
```

HUBBLE

- round

四舍五入

```
select round(10.7);
```

```
  round
---------
     11
```

```
select round(10.1);
```

```
  round
---------
     10
```

四舍五入到给定的小数位数

```
select round(10.3455,2);
```

```
  round
---------
  10.35
```

- trunc

截断小数，取整

```
select trunc(-45.71);
```

```
  trunc
---------
    -45
```

- sind

```
select sind(45.0);
```

```
         sind
----------------------
  0.7071067811865475
```

- sinh

```
select sinh(90.0);
```

```
          sinh
-------------------------
  6.102016471589204e+38
```

- STDDEV_POP

HUBBLE

数据准备

```
create table Player
(
PlayerName VARCHAR(100) NOT NULL,
RunScored INT NOT NULL,
WicketsTaken INT NOT NULL
);


insert into Player
(PlayerName, RunScored,  WicketsTaken )
values
('KL Rahul', 52, 0 ),
('Hardik Pandya', 30, 1 ),
('Ravindra Jadeja', 18, 2 ),
('Washington Sundar', 10, 1),
('D Chahar', 11, 2 ),
('Mitchell Starc', 0, 3);
```

将找到WicketsTaken列的总体标准差

```
select  STDDEV_POP(WicketsTaken)
as Pop_Std_Dev_Wickets
from Player ;
```

```
  pop_standard_deviation
---------------------------
   16.876183086099639597
```

- uuid_generate_v4

```
select uuid_generate_v4();
```

```
            uuid_generate_v4
----------------------------------------
  165f8c68-d670-4708-aa50-ae6fc1bb72b0
(1 row)
```

- VAR_POP

```
create table Player
(
PlayerName VARCHAR(100) NOT NULL,
RunScored INT NOT NULL,
WicketsTaken INT NOT NULL
);
```

HUBBLE

```
insert into Player
(PlayerName, RunScored,  WicketsTaken )
values
('KL Rahul', 52, 0 ),
('Hardik Pandya', 30, 1 ),
('Ravindra Jadeja', 18, 2 ),
('Washington Sundar', 10, 1),
('D Chahar', 11, 2 ),
('Mitchell Starc', 0, 3);
```

计算RunScored列的总体标准方差

```
select  VAR_POP(RunScored ) as Run_POPVariance
                          from Player ;
```

```
     run_popvariance
-------------------------
  284.80555555555555556
```

- VAR_SAMP

数据准备同上一个示例

RunScored列的样本方差。

```
select  VAR_SAMP(RunScored ) as Run_Variance
from Player ;
```

```
      run_variance
-------------------------
  341.76666666666666667
```

- TIMEOFDAY

返回当前日期和时间

```
select timeofday();
```

```
               timeofday
----------------------------------------
  Fri Nov 4 11:33:43.240953 2022 +0800
```

- asind

```
select asind(1);
```

```
  asind
---------
     90
(1 row)
```

HUBBLE

说明 1 的反正弦正好是 90°

- acosd

```
select acosd(0);
```

```
  acosd
---------
     90
(1 row)
```

确认 0 的反余弦正好是 90°。

- EVERY

```
create table book (id int,sal int ,bname string);

insert into book values (1, 1, '1984');
insert into book values (2, 1, 'Animal Farm');
insert into book values (3, 2, 'O Alquimista');
insert into book values (4, 2, 'Brida');
```

判断每本书的价格是否都小于 10 元

```
select every(sal < 10) from book;
```

```
  every
---------
  true
(1 row)
```

6.3.4.8 字符串函数

函数	返回类型	参数	说明
ascii	int	string	返回第一个字符的 asc 编码
bit_length	int	bytes	计算 bits 数
bit_length	int	string	计算 bits 数
btrim	string	string,string	删除匹配的字符
btrim	string	string	去除空格
char_length	int	bytes	计算字节长度
char_length	int	string	计算字符长度
chr	string	int	返回 ascii 值
concat	string	string...	拼接字符串
concat_ws	string	string...	使用第一个参数拼接字符串
convert_from	string	bytes，string	将字节转化为执行编码的字符串（支持 UTF8 与 LATIN1）
convert_to	bytes	string，string	将字符串转化为指定编码的字节（支持 UTF8 与 LATIN1）
decode	bytes	string，string	换化为字节码，支持 hex，escape，base64
encode	string	bytes，string	换化为字符串，支持 hex，escape，base64
from_ip	string	bytes	将 IP 的字节字符串表示形式转换为其字符串表示形式

函数	返回类型	参数	说明
from_uuid	string	bytes	将 UUID 的字节字符串表示形式转换为其字符串表示形式。
initcap	string	string	第一个字母大写。
left	bytes	bytes，int	获取头几位字节
left	string	string，int	获取头几位字符串
length	int	bytes	计算字节码的长度
length	int	string	计算字符串的长度
lower	string	string	将大写转化为小写
lpad	string	string，int	通过在字符串的左侧添加' '来增加长度。如果字符串比长度长，就被截断。
lpad	string	string，int ，string	通过在字符串的左侧添加传入参数来增加长度。如果字符串比长度长，就被截断。
ltrim	string	string，string	递归从输入的开始 (左侧) 删除包含的所有字符
ltrim	string	string	去除左边的空格
md5	string	bytes	计算 MD5 值
md5	string	string	计算 MD5 值
octet_length	int	bytes	计算字节数
octet_length	int	string	计算字节数
overlay	string	string，string，int	从指定下标开始替换字符
overlay	string	string，string，int，int	从指定下标开始替换字符
quote_ident	string	string	返回 val 作为 SQL 语句中的标识符。
quote_literal	string	string	返回 val 作为 SQL 语句中的字符串文字适当引用。
quote_nullable	string	string	将参数强制转换为字符串，然后将其作为文字引用。
regexp_extract	string	string，string	返回输入中正则表达式的第一个匹配项。
regexp_replace	string	string，string，string	将输入中的正则表达式`regex`的匹配项替换为正则表达式替换。
repeat	string	string，int	返回输入指定重复次数的参数
replace	string	string，string，string	用`replace`替换输入的第三个参数替换匹配的第二个参数
reverse	string	string	反转字符串字符的顺序。
right	bytes	bytes，int	获取从右开始头几位字节
right	string	string，int	获取从右开始头几位字符
rpad	string	string，int	通过在字符串的右侧添加' '来增加长度。如果字符串比长度长，就被截断。
rpad	string	string，int，string	通过在字符串的右侧添加传入参数来增加长度。如果字符串比长度长，就被截断。
rtrim	string	string，string	递归从输入的开始 (右侧) 删除包含的所有字符
rtrim	string	string	去除右边的空格
sha1	string	bytes	计算 SHA1 值
sha1	string	string	计算 SHA1 值
sha256	string	bytes	计算 SHA256 值
sha256	string	string	计算 SHA256 值
sha512	string	bytes	计算 SHA512 值
sha512	string	string	计算 SHA512 值
split_part	string	string，string，int	分割分隔符上的输入并返回第几位

函数	返回类型	参数	说明
strpos	int	string，string	计算输入中字符串开始的位置。
substr	string	string，string	返回与正则表达式regex匹配的输入子字符串。
substr	string	string，string，string	返回与正则表达式regex匹配的输入子字符串，使用第三个参数作为转义字符
substr	string	string，int	从指定下标开始截取字符串
substr	string	string，int，int	从指定下标开始截取字符串，并指定截取字符串长度
to_english	string	int	此函数使用英语基数来声明其参数的值。
to_hex	string	bytes	将 val 转换为其十六进制表示形式。
to_hex	bytes	string	将 val 转换为其十六进制表示形式。
to_uuid	bytes	string	将 UUID 的字符串表示形式转换为其字节字符串表示形式。
translate	string	string，string，string	指定替代字符串内容
upper	string	string	将字符内容转为大写

- ascii

```
select  ascii('a');
```

```
  ascii
+--------------+
   97
```

- 将字符串所有的单词进行格式化，首字母大写，其它为小写

```
select initcap('hi THOMAS');
```

```
  initcap
+--------------+
 Hi Thomas
```

- decode，对字符串按指定的类型进行解码

```
select decode('MTIzAAE=', 'base64');
```

```
  decode
+--------------+
   123
```

- char_length()

```
select char_length('stfwwwwwfsf');
```

```
  char_length
+--------------+
   11
```

- chr()

```
select chr(76);
```

```
  chr
+--------------+
   L
```

- CONCAT字符串的连接

```
create table full_test(
first_name varchar(10),
last_name  varchar(10)
);

insert into full_test values ('zhang','sanfeng'),('liu','dehua'),('王','阳明');

select
    CONCAT(first_name, ' ', last_name) AS "Full name"
from
    full_test;
```

```
  Full name
+--------------+
  zhang sanfeng
  liu dehua
    王 阳明
```

- concat_ws()

使用等号=拼接字段username，address

```
select concat_ws('=', username, address) as info from user;
```

```
  Full name
+--------------+
    张三=上海
    李四=上海
    王五=上海
```

- lower

将大写字母转化为小写

```
select  lower('ADDRESS');
```

```
    lower
+--------------+
   address
```

- overlay()用于函数的替换

第一个起始位置参数，不是从 0 开始的，从 1 开始，第一个参数是起始位置，第二个是要被代替的字符长度

```
select overlay('Txxxxas' placing 'hom' from 2 for 4) ;
```

```
   overlay
+--------------+
   Thomas
```

去掉for的情况，则会走默认代替的长度，故建议在使用时候选择语句中带有for的情况

```
select overlay('Txxxxas' placing 'hom' from 2 ) ;
```

```
   overlay
+--------------+
   Thomxas
```

- convert_from

```
select  convert_from('text_in_utf8', 'UTF8');
```

```
  convert_from
----------------
  text_in_utf8
```

- convert_to

```
select convert_to('some text', 'UTF8');
```

```
   convert_to
+--------------+
   some text
```

- encode

```
select encode(E'123\000\001', 'base64');
```

```
   encode
+--------------+
   MTIzAAE=
```

- LEFT()

获取前 5 位字符串

```
select  left('1234567890', 5);
```

```
  left
---------
  12345
```

- upper()

小写字母转大写

```
select  upper('abcdef');
```

```
   upper
+--------------+
   ABCDEF
```

- md5()

计算MD5值

```
select  md5('hubble');
```

```
                md5
+--------------------------------+
 90dc10ef0211b0088ac9430df0f6c158
```

- quote_nullable

可以将数字类型转化成字符串

```
select quote_nullable(123);
```

```
quote_nullable
+--------------+
     '123'
```

- replace

字符串的替换

```
select replace (
        'http://www.baidu.cn',
        'http',
        'https'
    );
```

```
    replace
+--------------------+
 https://www.baidu.cn
```

- sha256()加密

```
select sha256('hubble');
```

```
    sha256
+----------------------------------------------------------------+
 18d9486f99df32c58605e574ffae3773efe42294e95a1577c52a6f5d987ad44f
```

- split_part()

```
select split_part('add.ress', '.', 1);
```

```
   split_part
+--------------------+
   add
```

- strpos

指定字符串在目标字符串的位置

```
select strpos('high', 'ig');
```

```
   strpos
+--------------+
   2
```

- substr()

三个参数情况，以下代表从第 2 位截取，截取 3 个字符

```
select substr('address', 2,3);
```

```
   substr
+--------------+
   ddr
```

二个参数情况，代表从 2 位截取，默认截取到最后

```
select substr('address', 2);
```

```
   substr
+--------------+
   ddress
```

- to_english

```
select to_english(100);
```

```
 to_english
+--------------+
 one-zero-zero
```

- 将字符串中某些匹配的字符替换成指定字符串，目标字符与源字符都可以同时指定多个

```
select translate('abcde', 'ad', '14');
```

```
   translate
+--------------+
   1bc4e
```

HUBBLE

- DIFFERENCE，值返回 0-4，值越大相似性越大

用法:

```
DIFFERENCE(string, string)
```

```
select SOUNDEX('poor') soundex_poor, SOUNDEX('pour') soundex_pour,
DIFFERENCE('poor', 'pour') similarity;
```

```
  soundex_poor | soundex_pour | similarity
---------------+--------------+-------------
  P600         | P600         | 4
```

6.3.4.9 聚合函数

函数	说明	返回值类型
cume_dist()	计算当前行的相对排名：（当前行之前或与之对等的行数）/（总行数）。	float
dense_rank()	计算当前行的排名，不留空格；此功能计算对等组。	int
first_value(val: bool)	返回val在窗口框第一行的那一行求值。	bool
first_value(val: bytes)	返回val在窗口框第一行的那一行求值	bytes
first_value(val: date)	返回val在窗口框第一行的那一行求值。	date
first_value(val:decimal)	返回val在窗口框第一行的那一行求值。	decimal
first_value(val: float)	返回val在窗口框第一行的那一行求值。	float
first_value(val: int)	返回val在窗口框第一行的那一行求值。	int
first_value(val: string)	返回val在窗口框第一行的那一行求值。	string
first_value(val: time)	返回val在窗口框第一行的那一行求值。	time
first_value(val: timestamp)	返回val在窗口框第一行的那一行求值。	timestamp
first_value(val: uuid)	返回val在窗口框第一行的那一行求值。	uuid
first_value(val: jsonb)	返回val在窗口框第一行的那一行求值。	jsonb
lag(val: bool)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	bool
lag(val: bool, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	bool
lag(val: bool, n: int, default: bool)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	bool
lag(val: date, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	date
lag(val: date, n: int, default: date)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	date
lag(val: decimal)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	decimal

函数	说明	返回值类型
lag(val: decimal, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	decimal
lag(val: decimal, n: int, default: ↪ decimal)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	decimal
lag(val: float)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	float
lag(val: float, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	float
lag(val: float, n: int, default: float)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	float
lag(val: inet)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	inet
lag(val: inet, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	inet
lag(val: inet, n: int, default: inet)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	inet
lag(val: int)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	int
lag(val: int, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	int
lag(val: int, n: int, default: int)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	int
lag(val: interval)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	interval
lag(val: interval, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	interval
lag(val: interval, n: int, default: ↪ interval)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	interval
lag(val: string)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	string
lag(val: string, n: int)→ string	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	string

函数	说明	返回值类型
lag(val: string, n: int, default: string ↪)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	string
lag(val: time)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	time
lag(val: time, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	time
lag(val: time, n: int, default: time)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	time
lag(val: timestamp)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	timestamp
lag(val: timestamp, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	timestamp
lag(val: timestamp, n: int, default: ↪ timestamp)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	timestamp
lag(val: timestamptz)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	timestamptz
lag(val: timestamptz, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	timestamptz
lag(val: timestamptz, n: int, default: ↪ timestamptz)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	timestamptz
lag(val: uuid)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	uuid
lag(val: uuid, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	uuid
lag(val: uuid, n: int, default: uuid)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	uuid
lag(val: jsonb)	返回val在当前行分区内的前一行求值的结果；如果没有这样的行，则返回 null。	jsonb
lag(val: jsonb, n: int)	返回在其分区中当前行之前的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	jsonb
lag(val: jsonb, n: int, default: jsonb)	返回在其分区中当前行之前的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	jsonb
last_value(val: bool)	返回val在窗框的最后一行的那一行求值。	bool

函数	说明	返回值类型
last_value(val: bytes)	返回val在窗框的最后一行的那一行求值。	bytes
last_value(val: date)	返回val在窗框的最后一行的那一行求值。	date
last_value(val: decimal)	返回val在窗框的最后一行的那一行求值。	decimal
last_value(val: float)	返回val在窗框的最后一行的那一行求值。	float
last_value(val: inet)	返回val在窗框的最后一行的那一行求值。	inet
last_value(val: int)	返回val在窗框的最后一行的那一行求值。	int
last_value(val: interval)	返回val在窗框的最后一行的那一行求值。	interval
last_value(val: string)	返回val在窗框的最后一行的那一行求值。	string
last_value(val: time)	返回val在窗框的最后一行的那一行求值。	time
last_value(val: timestamp)	返回val在窗框的最后一行的那一行求值。	timestamp
last_value(val: timestamptz)	返回val在窗框的最后一行的那一行求值。	timestamptz
last_value(val: uuid)	返回val在窗框的最后一行的那一行求值。	uuid
last_value(val: jsonb)	返回val在窗框的最后一行的那一行求值。	jsonb
lead(val: bool)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	bool
lead(val: bool, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	bool
lead(val: bool, n: int, default: bool)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	bool
lead(val: bytes)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	bytes
lead(val: bytes, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	bytes
lead(val: bytes, n: int, default: bytes)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	bytes
lead(val: date)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	date
lead(val: date, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	date
lead(val: date, n: int, default: date)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	date
lead(val: decimal)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	decimal
lead(val: decimal, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	decimal

函数	说明	返回值类型
lead(val: decimal, n: int, default: ↪ decimal)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	decimal
lead(val: float)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	float
lead(val: float, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	float
lead(val: float, n: int, default: float)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	float
lead(val: inet)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	inet
lead(val: inet, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	int
lead(val: inet, n: int, default: inet)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	inet
lead(val: int)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	int
lead(val: int, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	int
lead(val: int, n: int, default: int)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	int
lead(val: interval)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	interval
lead(val: interval, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	interval
lead(val: interval, n: int, default: ↪ interval)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	interval
lead(val: string)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	string
lead(val: string, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	string
lead(val: string, n: int, default: ↪ string)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	string

函数	说明	返回值类型
lead(val: time)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	time
lead(val: time, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	time
lead(val: time, n: int, default: time)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	time
lead(val: timestamp)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	timestamp
lead(val: timestamp, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	timestamp
lead(val: timestamp, n: int, default: ↪ timestamp)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	timestamp
lead(val: timestamptz)→ timestamptz	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	timestamptz
lead(val: timestamptz, n: int)→ ↪ timestamptz	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	timestamptz
lead(val: timestamptz, n: int, default: ↪ timestamptz)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	timestamptz
lead(val: uuid)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	uuid
lead(val: uuid, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	uuid
lead(val: uuid, n: int, default: uuid)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	uuid
lead(val: jsonb)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	jsonb
lead(val: jsonb, n: int)	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	jsonb
lead(val: jsonb, n: int, default: jsonb)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	jsonb
lead(val: varbit)	返回val当前行分区中下一行的求值；如果没有这样的行，则返回 null。	varbit

函数	说明	返回值类型
lead(val: varbit, n: int)→ varbit	返回在其分区中当前行之后的行val处求值n；如果没有这样的行，则返回 null。n针对当前行进行评估。	varbit
lead(val: varbit, n: int, default: ↪ varbit)	返回在其分区中当前行之后的行val处求值n；如果没有，则返回行default（必须与类型相同val）。双方n并default相对于当前行评估。	varbit
nth_value(val: bool, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	bool
nth_value(val: bytes, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	bytes
nth_value(val: date, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	date
nth_value(val: decimal, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	decimal
nth_value(val: float, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	float
nth_value(val: inet, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	inet
nth_value(val: int, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	int
nth_value(val: interval, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	interval
nth_value(val: string, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	string
nth_value(val: time, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	time
nth_value(val: timestamp, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	timestamp
nth_value(val: timestamptz, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	timestamptz
nth_value(val: uuid, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	uuid
nth_value(val: jsonb, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	jsonb
nth_value(val: varbit, n: int)	返回val在n窗框第 th 行（从 1 开始）处求值的值；如果没有这样的行，则返回 null。	varbit
ntile(n: int)	计算范围为 1 到的整数n，将分区尽可能平均地划分。	int
percent_rank()	计算当前行的相对等级：（等级-1）/（总行-1）。	float
rank()	计算带有间隙的当前行的排名；与第一个对等方的 row_number 相同。	int
row_number()	计算分区中当前行的数量，从 1 开始计算。	int

- cume_dist

计算当前行的相对排名，统计小于等于当前工资的人数占总人数的比例

```
select ename,
       deptno,
       sal,
       cume_dist() OVER (order by sal) as cume_dist
from (table emp order by sal desc limit 10);
```

```
   ename   | deptno |    sal     | cume_dist
----------+--------+----------+------------
    TH     |     30 |  4801.79 |        0.1
  FORD     |     20 |  4831.53 |        0.3
  SCOTT    |     20 |  4831.53 |        0.3
  MILLER   |     10 |  4936.76 |        0.4
  KING     |     10 |  5000.00 |        0.5
  PAIABRS  |     20 |  5381.50 |        0.8
  PAIRS    |     10 |  5381.50 |        0.8
  LEAERS   |     40 |  5381.50 |        0.8
  LAIHUI   |     10 | 18000.00 |          1
   LUCK    |     10 | 18000.00 |          1
```

- PERCENT_RANK()

语法

```
PERCENT_RANK() OVER (
    [partition by partition_expression, ... ]
    order by sort_expression [ASC | desc], ...
)
```

分析以上语法：

- 这个分区依据子句将行分为多个分区，PERCENT_RANK()函数已应用。
- 这个排序子句指定应用该函数的每个分区中的行顺序。
- 这个PERCENT_RANK()函数返回大于等于0且小于等于1的结果。

```
select
    ename,
    sal,
    PERCENT_RANK() OVER (
        order by sal
    )
from
    (table emp  order by sal desc limit 10);
```

```
   ename   |    sal     |     percent_rank
----------+----------+---------------------
  LUCK     |  4801.79 |                   0
  FORD     |  4831.53 | 0.1111111111111111
  SCOTT    |  4831.53 | 0.1111111111111111
```

```
  MILLER   |  4936.76 | 0.3333333333333333
  KING     |  5000.00 | 0.4444444444444444
  PAIABRS  |  5381.50 | 0.5555555555555556
  PAIRS    |  5381.50 | 0.5555555555555556
  LEAERS   |  5381.50 | 0.5555555555555556
  LAIHUI   | 18000.00 | 0.8888888888888888
  WANG     | 18000.00 | 0.8888888888888888
```

- 有关dense_rank()，rank()，row_number()的示例可以参考窗口函数
- NTILE()

在 hubble 中，的NTILE()函数用于将分区中的有序行划分为指定数量的已排序存储，NTILE()函数允许用户将分区中的有序行划分为指定数量的分级组，并使其大小尽可能相等。这些排名的组称为存储桶。NTILE()函数为每个组分配一个从 1 开始的存储桶号。对于组中的每一行，NTILE()函数分配一个存储桶号，该存储桶号代表该行所属的组。

语法

NTILE()函数的语法如下：

```
NTILE(buckets) OVER (
    [partition by partition_expression, ... ]
    [order by sort_expression [ASC | desc], ...]
)
```

语法分析语法：

- 这个水桶代表排名组的数量。它可以是数字或表达式，每个分区的计算结果为正整数值 (大于 0)。的水桶不能为空。
- partition by子句将行分布到应用了该函数的分区中。partition by子句是可选的。
- 这个排序子句对应用该函数的每个分区中的行进行排序。

示例数据

```
create table sales_stats(
    name   VARCHAR(100) NOT NULL,
    year   INT          NOT NULL CHECK (year > 0),
    amount DECIMAL(10, 2) CHECK (amount >= 0)

);

insert into
    sales_stats(name, year, amount)
values
    ('Raju kumar', 2018, 120000),
    ('Alibaba', 2018, 110000),
    ('Gabbar Singh', 2018, 150000),
    ('Kadar Khan', 2018, 30000),
    ('Amrish Puri', 2018, 200000),
    ('Raju kumar', 2019, 150000),
    ('Alibaba', 2019, 130000),
```

```
    ('Gabbar Singh', 2019, 180000),
    ('Kadar Khan', 2019, 25000),
    ('Amrish Puri', 2019, 270000);
```

NTILE()用于将数据分配到 3 个存储桶中的函数:

```
select
    name,
    amount,
    NTILE(3) OVER(
        order by amount
    )
from
    sales_stats
WHERE
    year = 2018;
```

```
      name      |  amount    | ntile
---------------+-----------+--------
  Raju kumar    | 120000.00 |      2
  Alibaba       | 110000.00 |      1
  Gabbar Singh  | 150000.00 |      2
  Kadar Khan    |  30000.00 |      1
  Amrish Puri   | 200000.00 |      3
```

NTILE()划分行的函数将表分为两个分区，每个分区有 3 个存储桶:

```
select
    name,
    amount,
    NTILE(3) OVER(
        partition by year
        order by amount
    )
from
    sales_stats;
```

```
      name      |  amount    | ntile
---------------+-----------+--------
  Raju kumar    | 120000.00 |      2
  Alibaba       | 110000.00 |      1
  Gabbar Singh  | 150000.00 |      2
  Kadar Khan    |  30000.00 |      1
  Amrish Puri   | 200000.00 |      3
  Raju kumar    | 150000.00 |      2
  Alibaba       | 130000.00 |      1
  Gabbar Singh  | 180000.00 |      2
  Kadar Khan    |  25000.00 |      1
```

HUBBLE

```
  Amrish Puri   | 270000.00 |      3
```

- FIRST_VALUE()

在 hubble，FIRST_VALUE()函数用于在结果集的排序分区中返回第一个值。

语法

```
FIRST_VALUE ( expression )
OVER (
    [partition by partition_expression, ... ]
    order by sort_expression [ASC | desc], ...
)
```

语法分析

- 这个表达可以是根据结果集排序分区的第一行的值求值的表达式，列或子查询。的表达必须返回单个值。
- partition by子句将结果集中的行划分为应用了FIRST_VALUE()函数的分区。当用户使用partition by
  ↪ 子句时，FIRST_VALUE()函数将整个结果集视为单个分区。
- 这个排序子句指定每个分区中行的排序顺序FIRST_VALUE()函数被应用。
- 这个rows_range_clause通过定义分区的开始和结束来进一步限制分区中的行。

数据准备

```
create table product_groups (
    group_id int PRIMARY KEY,
    group_name VARCHAR(255) NOT NULL
);

create table products (
    product_name VARCHAR (255) NOT NULL,
    price DECIMAL (11, 2),
    group_id INT NOT NULL
);

insert into product_groups (group_name)
values
    (1,'Smartphone'),
    (2,'Laptop'),
    (3,'Tablet');

insert into products (product_name, group_id, price)
values
    ('Microsoft Lumia', 1, 200),
    ('HTC One', 1, 400),
    ('Nexus', 1, 500),
    ('iPhone', 1, 900),
    ('HP Elite', 2, 1200),
    ('Lenovo Thinkpad', 2, 700),
    ('Sony VAIO', 2, 700),
    ('Dell Vostro', 2, 800),
```

```
    ('iPad', 3, 700),
    ('Kindle Fire', 3, 150),
    ('Samsung Galaxy Tab', 3, 200);
```

使用FIRST_VALUE()函数返回所有产品以及价格最低的产品:

```
select
    product_name,
    group_id,
    price,
    FIRST_VALUE(product_name)
    OVER(
        order by price
    ) lowest_price
from
    products;
```

```
     product_name     | group_id |  price   | lowest_price
----------------------+----------+---------+---------------
  Microsoft Lumia     |        1 |  200.00 | Kindle Fire
  HTC One             |        1 |  400.00 | Kindle Fire
  Nexus               |        1 |  500.00 | Kindle Fire
  iPhone              |        1 |  900.00 | Kindle Fire
  HP Elite            |        2 | 1200.00 | Kindle Fire
  Lenovo Thinkpad     |        2 |  700.00 | Kindle Fire
  Sony VAIO           |        2 |  700.00 | Kindle Fire
  Dell Vostro         |        2 |  800.00 | Kindle Fire
  iPad                |        3 |  700.00 | Kindle Fire
  Kindle Fire         |        3 |  150.00 | Kindle Fire
  Samsung Galaxy Tab  |        3 |  200.00 | Kindle Fire
```

使用FIRST_VALUE()函数返回按产品组分组的所有产品。对于每个产品组，它以最低的价格返回产品:

```
select
    product_name,
    group_id,
    price,
    FIRST_VALUE(product_name)
    OVER(
    partition by group_id
        order by price
        RANGE between
            UNBOUNDED PRECEDING and
            UNBOUNDED FOLLOWING
    ) lowest_price
from
    products;
```

```
     product_name     | group_id |  price   |  lowest_price
----------------------+----------+----------+------------------
  Microsoft Lumia     |        1 |  200.00 | Microsoft Lumia
  HTC One             |        1 |  400.00 | Microsoft Lumia
  Nexus               |        1 |  500.00 | Microsoft Lumia
  iPhone              |        1 |  900.00 | Microsoft Lumia
  HP Elite            |        2 | 1200.00 | Lenovo Thinkpad
  Lenovo Thinkpad     |        2 |  700.00 | Lenovo Thinkpad
  Sony VAIO           |        2 |  700.00 | Lenovo Thinkpad
  Dell Vostro         |        2 |  800.00 | Lenovo Thinkpad
  iPad                |        3 |  700.00 | Kindle Fire
  Kindle Fire         |        3 |  150.00 | Kindle Fire
  Samsung Galaxy Tab  |        3 |  200.00 | Kindle Fire
```

- LAST_VALUE()

LAST_VALUE()函数返回结果集的有序分区中的最后一个值。

语法

```
LAST_VALUE ( expression )
OVER (
    [partition by partition_expression, ... ]
    order by sort_expression [ASC | desc], ...
)
```

语法分析

- 表达式可以是根据结果集的有序分区中最后一行的值求值的表达式，列或子查询。表达式必须返回单个值。
- 这个分区依据子句将结果集的行划分为分区，LAST_VALUE()函数已应用。
- 这个排序子句指定每个分区中行的排序顺序LAST_VALUE()函数已应用。
- 这个frame_clause定义当前分区中行的子集，LAST_VALUE()函数已应用。

使用LAST_VALUE()函数返回所有产品以及价格最高的产品：

```
select
    product_name,
    price,
    LAST_VALUE(product_name)
    OVER(
        order by price
        RANGE between
            UNBOUNDED PRECEDING and
            UNBOUNDED FOLLOWING
    ) highest_price
from
    products;
```

```
     product_name     |  price   | highest_price
----------------------+----------+----------------
```

```
  Microsoft Lumia     |   200.00 | HP Elite
  HTC One             |   400.00 | HP Elite
  Nexus               |   500.00 | HP Elite
  iPhone              |   900.00 | HP Elite
  HP Elite            |  1200.00 | HP Elite
  Lenovo Thinkpad     |   700.00 | HP Elite
  Sony VAIO           |   700.00 | HP Elite
  Dell Vostro         |   800.00 | HP Elite
  iPad                |   700.00 | HP Elite
  Kindle Fire         |   150.00 | HP Elite
  Samsung Galaxy Tab  |   200.00 | HP Elite
```

使用LAST_VALUE()用于将所有产品连同每个产品组中最昂贵的产品一起退回的函数:

```
select
    product_name,
    group_id,
    price,
    LAST_VALUE(product_name)
    OVER(
    partition by group_id
        order by price
        RANGE between
            UNBOUNDED PRECEDING and
            UNBOUNDED FOLLOWING
    ) highest_price
from
    products;
```

```
     product_name     | group_id |   price   | highest_price
----------------------+----------+---------+----------------
  Microsoft Lumia     |        1 |   200.00 | iPhone
  HTC One             |        1 |   400.00 | iPhone
  Nexus               |        1 |   500.00 | iPhone
  iPhone              |        1 |   900.00 | iPhone
  HP Elite            |        2 |  1200.00 | HP Elite
  Lenovo Thinkpad     |        2 |   700.00 | HP Elite
  Sony VAIO           |        2 |   700.00 | HP Elite
  Dell Vostro         |        2 |   800.00 | HP Elite
  iPad                |        3 |   700.00 | iPad
  Kindle Fire         |        3 |   150.00 | iPad
  Samsung Galaxy Tab  |        3 |   200.00 | iPad
```

- ARRAY_AGG()

ARRAY_AGG()函数是一个聚合函数，它接受一组值并返回一个数组，其中将输入集中的每个值分配给该数组的元素。

用法

```
ARRAY_AGG(expression [order by [sort_expression {ASC | desc}], [...])
```

order by子句是自愿性子句。它指定集合中要处理的行的顺序，从而确定结果数组中元素的顺序。它通常与group by子句一起使用。

```
create table film (id int,pname  string,fname string);
insert into film values (1,'战狼二','wujing');
insert into film values (1,'断背山','lian');
insert into film values (3,'西游记','wuchengen');;
insert into film values (3,'水浒传','shinaian');
insert into film values (2,'亮剑','liyunlong');
insert into film values (2,'五号特工组','wanglikun');
```

```
select
    id,
    ARRAY_AGG (pname || ' ' || fname) actors
from
    film
group by id;
```

```
 id  |                actors
-----+----------------------------------------
   1 | {战狼二 wujing,断背山 lian}
   3 | {西游记 wuchengen,水浒传 shinaian}
   2 | {亮剑 liyunlong,五号特工组 wanglikun}
```

- LEAD()

在 hubble 中，LEAD()函数用于以特定的物理偏移量访问当前行之后的行。LEAD()函数提供对以指定物理偏移量访问当前行之后的行的访问。这意味着从当前行开始，LEAD()函数可以访问下一行，下一行之后的行等等的数据。LEAD()函数对于将当前行的值与当前行之后的行的值进行比较非常有用。

语法：

```
LEAD(expression [, offset [, default_value]])
OVER (
    [partition by partition_expression, ... ]
    order by sort_expression [ASC | desc], ...
)
```

语法分析：

- 这个表达根据与当前行的指定偏移量针对下一行进行评估。的表达可以是必须计算为单个值的列，表达式，子查询。
- 偏移量是一个正整数，它指定从当前行转发访问的行数。偏移量可以是表达式，子查询或列。偏移量默认为 1。
- partition by子句将行划分为应用了LEAD()函数的分区。默认情况下，如果省略partition by子句，则整个结果集是单个分区。
- 这个排序子句指定每个分区中行的排序顺序LEAD()函数已应用。

示例数据准备

```
create table sales(
    year INT CHECK(year > 0),
    group_id INT NOT NULL,
    amount DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY(year, group_id)
);

insert into
    sales(year, group_id, amount)
values
    (2018, 1, 1474),
    (2018, 2, 1787),
    (2018, 3, 1760),
    (2019, 1, 1915),
    (2019, 2, 1911),
    (2019, 3, 1118),
    (2020, 1, 1646),
    (2020, 2, 1975),
    (2020, 3, 1516);
```

以下查询使用LEAD()函数返回当年和下一年的销售额:

```
WITH cte AS (
    select
        year,
        SUM(amount) amount
    from sales
    group by year
    order by year
)
select
    year,
    amount,
    LEAD(amount, 1) OVER (
        order by year
    ) next_year_sales
from
    cte;
```

```
  year | amount   | next_year_sales
-------+----------+------------------
  2018 | 5021.00 |          4944.00
  2019 | 4944.00 |          5137.00
  2020 | 5137.00 |          NULL
```

以下语句使用LEAD()函数将每个产品组的当年销售额与下一年的销售额进行比较

```
select
    year,
    amount,
    group_id,
    LEAD(amount, 1) OVER (
        partition by group_id
        order by year
    ) next_year_sales
from
    sales;
```

```
  year | amount   | group_id | next_year_sales
-------+---------+----------+------------------
  2018 | 1474.00 |        1 |          1915.00
  2019 | 1915.00 |        1 |          1646.00
  2020 | 1646.00 |        1 |          NULL
  2018 | 1787.00 |        2 |          1911.00
  2019 | 1911.00 |        2 |          1975.00
  2020 | 1975.00 |        2 |          NULL
  2018 | 1760.00 |        3 |          1118.00
  2019 | 1118.00 |        3 |          1516.00
  2020 | 1516.00 |        3 |          NULL
```

- LAG()

在 hubble 中，LAG()函数用于访问一行那在当前行之前以特定的物理偏移量出现。LAG()函数提供对在指定物理偏移量的当前行之前的行的访问。换句话说，从当前行开始，LAG()函数可以访问前一行或前一行之前的行的数据，依此类推。LAG()函数对于比较当前行和上一行的值非常有用。

语法

```
LAG(expression [, offset [, default_value]])
OVER (
    [partition by partition_expression, ... ]
    order by sort_expression [ASC | desc], ...
)
```

语法分析

- 这个表达将根据当前行之前指定偏移量的行进行评估。它可以是列，表达式或子查询。
- 这个抵消是一个正整数，指定行数那来在要从中访问数据的当前行之前。的抵消可以是表达式，子查询或列。默认为 1。
- 这个LAG()函数将返回default_value抵消超出了分区的范围。
- partition by子句将行划分为应用了LAG()函数的分区。默认情况下，如果用户忽略partition by子句，该函数会将整个结果集视为一个分区。
- 这个排序子句指定每个分区中行的顺序LAG()函数已应用。

示例数据准备

```
create table sales(
```

```
    year INT CHECK(year > 0),
    group_id INT NOT NULL,
    amount DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY(year, group_id)
);

insert into
    sales(year, group_id, amount)
values
    (2018, 1, 1474),
    (2018, 2, 1787),
    (2018, 3, 1760),
    (2019, 1, 1915),
    (2019, 2, 1911),
    (2019, 3, 1118),
    (2020, 1, 1646),
    (2020, 2, 1975),
    (2020, 3, 1516);
```

LAG()函数返回当年和上一年的销售额:

```
WITH cte AS (
    select
        year,
        SUM(amount) amount
    from sales
    group by year
    order by year
)
select
    year,
    amount,
    LAG(amount, 1) OVER (
        order by year
    ) previous_year_sales
from
    cte;
```

```
  year | amount   | previous_year_sales
-------+----------+----------------------
  2018 | 5021.00 |              NULL
  2019 | 4944.00 |              5021.00
  2020 | 5137.00 |              4944.00
```

LAG()函数比较每个产品组的当年销售额与上一年的销售额:

```
select
    year,
```

```
    amount,
    group_id,
    LAG(amount, 1) OVER (
        partition by group_id
        order by year
    ) previous_year_sales
from
    sales;
```

```
  year | amount   | group_id | previous_year_sales
-------+---------+----------+----------------------
  2018 | 1474.00 |        1 |              NULL
  2019 | 1915.00 |        1 |              1474.00
  2020 | 1646.00 |        1 |              1915.00
  2018 | 1787.00 |        2 |              NULL
  2019 | 1911.00 |        2 |              1787.00
  2020 | 1975.00 |        2 |              1911.00
  2018 | 1760.00 |        3 |              NULL
  2019 | 1118.00 |        3 |              1760.00
  2020 | 1516.00 |        3 |              1118.00
```

- NTH_VALUE()

NTH_VALUE()函数用于从获取值在结果集中的行。NTH_VALUE()函数从结果集的有序分区中返回一个值的第N行。

语法

```
NTH_VALUE(expression, offset)
OVER (
    [partition by partition_expression]
    [ order by sort_expression [ASC | desc]
    frame_clause ]
)
```

分析语法:

- 这个表达是目标列或表达式NTH_VALUE()函数运行。
- 这个分区依据子句将结果集的行分配到分区中，NTH_VALUE()函数适用。
- 这个排序子句对应用该函数的每个分区中的行进行排序。
- 这个frame_clause定义当前分区的子集或框架。

数据准备

```
create table products (
    product_name VARCHAR (255) NOT NULL,
    price DECIMAL (11, 2),
    group_id INT NOT NULL
);

insert into products (product_name, group_id, price)
values
```

```
    ('Microsoft Lumia', 1, 200),
    ('HTC One', 1, 400),
    ('Nexus', 1, 500),
    ('iPhone', 1, 900),
    ('HP Elite', 2, 1200),
    ('Lenovo Thinkpad', 2, 700),
    ('Sony VAIO', 2, 700),
    ('Dell Vostro', 2, 800),
    ('iPad', 3, 700),
    ('Kindle Fire', 3, 150),
    ('Samsung Galaxy Tab', 3, 200);
```

NTH_VALUE()函数返回每个产品组中价格第 3 贵的产品

```
select
    product_name,
    price,
    NTH_VALUE(product_name, 3)
    OVER(
        order by price desc
        RANGE between
            UNBOUNDED PRECEDING and
            UNBOUNDED FOLLOWING
    )
from
    products;
```

```
     product_name     |  price  |  nth_value
----------------------+---------+--------------
  Microsoft Lumia     |  200.00 | Dell Vostro
  HTC One             |  400.00 | Dell Vostro
  Nexus               |  500.00 | Dell Vostro
  iPhone              |  900.00 | Dell Vostro
  HP Elite            | 1200.00 | Dell Vostro
  Lenovo Thinkpad     |  700.00 | Dell Vostro
  Sony VAIO           |  700.00 | Dell Vostro
  Dell Vostro         |  800.00 | Dell Vostro
  iPad                |  700.00 | Dell Vostro
  Kindle Fire         |  150.00 | Dell Vostro
  Samsung Galaxy Tab  |  200.00 | Dell Vostro
```

NTH_VALUE()函数返回每个产品组中价格最贵的产品:

```
select
    product_name,
    price,
    group_id,
    NTH_VALUE(product_name, 1)
```

```
    OVER(
        partition by group_id
        order by price desc
        RANGE between
            UNBOUNDED PRECEDING and
            UNBOUNDED FOLLOWING
    )
from
    products;
```

```
     product_name     |  price   | group_id | nth_value
----------------------+----------+----------+------------
  Microsoft Lumia     |  200.00  |        1 | iPhone
  HTC One             |  400.00  |        1 | iPhone
  Nexus               |  500.00  |        1 | iPhone
  iPhone              |  900.00  |        1 | iPhone
  HP Elite            | 1200.00  |        2 | HP Elite
  Lenovo Thinkpad     |  700.00  |        2 | HP Elite
  Sony VAIO           |  700.00  |        2 | HP Elite
  Dell Vostro         |  800.00  |        2 | HP Elite
  iPad                |  700.00  |        3 | iPad
  Kindle Fire         |  150.00  |        3 | iPad
  Samsung Galaxy Tab  |  200.00  |        3 | iPad
```

- AVG()

hubble 提供了AVG()函数来计算集合的平均值。AVG()函数是 hubble 中最常用的聚合函数之一。AVG()函数使用户能够计算数字列的平均值。

语法

```
AVG(column)
```

它可以与select和HAVING子句一起使用。

查询员工平均支出

```
select avg(sal) from emp;
```

```
           avg
-------------------------
  4655.867777777777778
```

按照员工部门计算平均支出

```
select deptno,avg(sal) from emp group by deptno;
```

```
  deptno |          avg
---------+------------------------
      10 | 7373.4033333333333333
```

```
      20 |               3490.455
      40 |                5381.50
      30 | 3010.2916666666666667
      50 |                1000.00
```

按照员工部门计算平均支出，且平均支出大于 5000

```
select deptno,avg(sal) from emp group by deptno having(avg(sal))>5000;
```

```
  deptno |          avg
---------+------------------------
      10 | 7373.4033333333333333
      40 |                5381.50
```

- COUNT()

COUNT()函数是一个聚合函数，使用户能够获取与查询的特定要求匹配的行数。根据用户要求，COUNT()函数可以具有以下语法：

用法 1：COUNT(*)

```
All rows including null and Duplicates
```

用法 2：COUNT(column)

```
All rows except null.
```

用法 3：COUNT(DISTINCT column)

```
All rows without null and Duplicates
```

COUNT()函数与select语句一起使用

查询员工的数量

```
select count(*) from emp;
```

```
  count
---------
     27
```

查询员工部门的数量

```
select count(DISTINCT deptno) from emp;
```

```
  count
---------
     5
```

- SUM()

提供了一个SUM()函数，该函数用于获取数字列的值的加法

HUBBLE

```
SUM(column)
```

使用上述函数时，请牢记以下几点：

- 它忽略所有NULL值。
- 如果与DISTINCT运算符一起用作SUM(DISTINCT)，它将跳过重复的值。
- 将SUM()函数与select子句一起使用将返回NULL而不是零。

使用SUM()函数和group by子句计算每个部门支付的量

```
select deptno,sum(sal) from emp group by deptno ;
```

```
  deptno |    sum
---------+-----------
      10 | 66360.63
      20 | 34904.55
      40 |  5381.50
      30 | 18061.75
      50 |  1000.00
```

计算公司对员工总支出量

```
select sum(sal) from emp;
```

```
     sum
-------------
  125708.43
```

- MAX()

MAX()函数是一个聚合函数，它返回一组值中的最大值。

语法

```
MAX(expression);
```

MAX()函数可以与select，WHERE和HAVING子句一起使用。

查询支出最多的员工工资

```
select max(sal) from emp;
```

```
    max
------------
  18000.00
```

查询每个部门支出最多的员工工资

```
select deptno,max(sal) from emp group by deptno;
```

HUBBLE

```
  deptno |    max
---------+-----------
      10 | 18000.00
      20 |  5381.50
      40 |  5381.50
      30 |  4801.79
      50 |  1000.00
```

- MIN()

MIN()函数是一个聚合函数，它返回一组值中的最小值。

```
MIN(expression);
```

MIN()函数可以与select，WHERE和HAVING子句一起使用。

查询支出最少的员工工资

```
select min(sal) from emp;
```

```
    min
------------
   1000.00
```

查询每个部门支出最少的员工工资

```
select deptno,min(sal) from emp group by deptno;
```

```
 deptno  |    min
---------+----------
      10 | 2134.00
      20 | 1000.00
      40 | 5381.50
      30 | 2165.00
      50 | 1000.00
```

6.3.4.10 JSON 函数

函数	说明	返回值类型
array_to_json(array: anyelement[])	将数组返回为 JSON 或 JSONB。	json
json_array_length(json: jsonb)→ int	返回最外面的 JSON 或 JSONB 数组中的元素数。	int
json_build_array(anyelement...)	从可变参数列表中构建一个可能是异构类型的 JSON 或 JSONB 数组。	json
json_build_object(anyelement...)	从可变参数列表中构建 JSON 对象。	json
json_extract_path(jsonb, string...)	返回可变参数所指向的 JSON 值。	json
json_object(keys: string[], values: ↪ string[])	这种形式的 json_object 从两个单独的数组中成对地获取键和值。在所有其他方面，它与单参数形式相同。	json

函数	说明	返回值类型
`json_object(texts: string[])`	从文本数组构建 JSON 或 JSONB 对象。数组必须具有一维且成员数为偶数，在这种情况下，它们被视为交替的键/值对。	json
`json_remove_path(val: jsonb, path: string[])`	从 JSON 对象中删除指定的路径。	json
`json_set(val: jsonb, path: string[], to: jsonb)`	返回可变参数所指向的 JSON 值。	json
`json_set(val: jsonb, path: string[], to: jsonb, create_missing: bool)`	返回可变参数所指向的 JSON 值。如果`create_missing`为 false，则不会将新键插入对象，也不会在值之前或之后添加值。	json
`json_strip_nulls(from_json: jsonb)`	返回 from_json，其中所有具有空值的对象字段都被省略。其他空值保持不变。	json
`json_typeof(val: jsonb)`	以文字字串传回最外层 JSON 值的类型。	string
`jsonb_array_length(json: jsonb)`	返回最外面的 JSON 或 JSONB 数组中的元素数。	int
`jsonb_build_array(anyelement...)`	从可变参数列表中构建一个可能是异构类型的 JSON 或 JSONB 数组。	int
`jsonb_build_object(anyelement...)`	从可变参数列表中构建 JSON 对象。	json
`jsonb_extract_path(jsonb, string...)`	返回可变参数所指向的 JSON 值。	json
`jsonb_insert(target: jsonb, path: string[], new_val: jsonb)`	返回可变参数所指向的 JSON 值。`new_val`将在路径目标之前插入。	jsonb
`jsonb_insert(target: jsonb, path: string[], new_val: jsonb, insert_after: bool)`	返回可变参数所指向的 JSON 值。如果`insert_after`为 true（默认为 false），`new_val`则将在路径目标之后插入。	jsonb
`jsonb_object(keys: string[], values: string[])`	这种形式的 json_object 从两个单独的数组中成对地获取键和值。在所有其他方面，它与单参数形式相同。	jsonb
`jsonb_object(texts: string[])`	从文本数组构建 JSON 或 JSONB 对象。数组必须具有一维且成员数为偶数，在这种情况下，它们被视为交替的键/值对。	jsonb
`jsonb_pretty(val: jsonb)`	以缩进和换行符的形式返回给定的 JSON 值。	string
`jsonb_set(val: jsonb, path: string[], to: jsonb)`	返回可变参数所指向的 JSON 值。	jsonb
`jsonb_set(val: jsonb, path: string[], to: jsonb, create_missing: bool)`	返回可变参数所指向的 JSON 值。如果`create_missing`为 false，则不会将新键插入对象，也不会在值之前或之后添加值。	jsonb
`jsonb_strip_nulls(from_json: jsonb)`	返回 from_json，其中所有具有空值的对象字段都被省略。其他空值保持不变。	jsonb
`jsonb_typeof(val: jsonb)`	以文字字串传回最外层 JSON 值的类型。	string
`to_json(val: anyelement)`	以 JSON 或 JSONB 的形式返回值。	json
`to_jsonb(val: anyelement)`	以 JSON 或 JSONB 的形式返回值。	jsonb

- `to_json`

```
select to_json('json type'::text);
```

```
    to_json
---------------
  "json type"
```

- to_jsonb

```
select to_jsonb('json type'::text);
```

```
    to_jsonb
---------------
  "json type"
```

- json_array_length

```
select json_array_length('[1,2,3,4]');
```

```
  json_array_length
---------------------
                   4
```

- json_build_array

```
 select json_build_array(1,2,'3',4,5);
```

```
  json_build_array
---------------------
  [1, 2, "3", 4, 5]
```

- json_build_object

```
select json_build_object('1','2','3','4');
```

```
   json_build_object
------------------------
  {"1": "2", "3": "4"}
```

- jsonb_pretty

数据准备

```
create table reports (rep_id int primary key, data json);


insert into reports (rep_id, data)
values
(1, '{"objects":[{"album": 1, "src":"fooA.png", "pos": "top"}, {"album": 2, "src
    ↪ ":"barB.png", "pos": "top"}], "background":"background.png"}')
, (2, '{"objects":[{"album": 1, "src":"fooA.png", "pos": "top"}, {"album": 2, "
    ↪ src":"barC.png", "pos": "top"}], "background":"bacakground.png"}')
```

HUBBLE

```
, (3, '{"objects":[{"album": 1, "src":"fooA.png", "pos": "middle"},{"album": 2,
    ↪ "src":"barB.png", "pos": "middle"}],"background":"background.png"}')
, (4, '{"objects":[{"album": 1, "src":"fooA.png", "pos": "top"}, {"album": 3, "
    ↪ src":"barB.png", "pos": "top"}], "background":"backgroundA.png"}')
;
```

```
select jsonb_pretty( '{"name": "Alice", "agent": {"bot": true} }'::jsonb );
```

```
     jsonb_pretty
------------------------
  {
      "agent": {
          "bot": true
      },
      "name": "Alice"
  }
```

- json_object_keys

返回最外层的json对象中的键的集合

```
select * from json_object_keys('{"b":"1","a":"2"}');
```

```
  json_object_keys
--------------------
  a
  b
```

- jsonb_set

json值的更新，jsonb_set函数，格式：jsonb_set(target jsonb,path text[],new_value jsonb[,
↪ create_missing boolean])，target指源jsonb数据，path指路径，new_value指更新后的键值，create_missing值为true表示键不存在则添加，为false表示如果键不存在则不添加。

```
select jsonb_set('{"name":"bob","age":"27"}'::jsonb,'{age}','"28"'::jsonb,false)
    ↪ ;
```

```
           jsonb_set
--------------------------------
  {"age": "28", "name": "bob"}
```

- json_each

扩展最外层的json对象成为一组键值结果集

```
select * from json_each('{"b":"1","a":"2"}');
```

```
  key | value
------+--------
  a   | "2"
  b   | "1"
```

- json_each_text以文本返回结果

```
select * from json_each_text('{"b":"1","a":"2"}');
```

```
  key | value
------+--------
  a   | 2
  b   | 1
```

- json_object_keys

返回最外层的json对象中的键的集合

```
select * from json_object_keys('{"b":"1","a":"2"}');
```

```
  json_object_keys
--------------------
  a
  b
```

- json_typeof

以文字字串传回最外层JSON值的类型

```
select * from json_typeof('{"b":"1","a":"2"}');
```

```
  json_typeof
---------------
  object
```

- json键值的删除用-

```
select '{"b":"1","a":"2"}'::json - 'a';
```

```
   column
--------------
  {"b": "1"}
```

- 删除嵌套json数据

```
select '["a","b","c"]'::jsonb - 0;
```

```
  column
------------
 ["b", "c"]
```

```
select '["a","b","c"]'::jsonb - 1;
```

```
  column
------------
 ["a", "c"]
```

HUBBLE

- `json_strip_nulls`

```
select json_strip_nulls('[{"f1":1,"f2":null},2,null,3]');
```

```
     json_strip_nulls
---------------------------
  [{"f1": 1}, 2, null, 3]
```

6.3.5 数据库信息

Hubble 提供了 information_schema 的虚 schema，其中包含了数据库的表、列、索引和视图的信息。

6.3.5.1 包含的数据信息

对象	Infomation Schema 中的表	使用 show 语句
columns	columns	SHOW COLUMNS
Constraints	check_constraints, key_column_usage, referential_constraints, table_constraints	SHOW CONSTRAINTS
Databases	schemata	SHOW DATABASE
Indexes	statistics	SHOW INDEX
Privileges	schema_privileges,table_privileges	SHOW GRANTS
Roles	role_table_grants	SHOW ROLES
Sequences	sequences	SHOW CREATE SEQUENCE
Tables	tables	SHOW TABLES
Views	tables,views	SHOW CREATE

6.3.5.2 administrable_role_authorizations

当前用户是否具有管理员权限的信息，如使用普通用户，查此表则为空

列名	描述
grantee	当前用户
role_name	所属角色
is_grantable	是否可授权

6.3.5.3 applicable_roles

标识当前用户可以使用其授权的所有角色。这意味着，存在一个从当前用户到相关角色的角色授予链。当前用户本身也是一个适用的角色，但是没有列出。

列名	描述
grantee	当前用户
role_name	所属角色
is_grantable	是否可授权

6.3.5.4 check_constraints

包含检查约束应用在具体数据库列的的信息

列名	描述
constraint_catalog	包含约束的数据库的名称
constraint_schema	包含约束的 schema 的名称
constraint_name	约束名称
check_clause	检查约束的定义。

6.3.5.5 columns

包含关于每个表中列的信息

列名	描述
table_catalog	包含表的数据库的名称
table_schema	包含表的 schema 的名称
table_schema	表名称
table_schema	列名称
table_schema	表中列的序号位置 (从 1 开始)。
column_default	列的默认值。
is_nullable	如果该列接受空值，则为 YES;
data_type	列的数据类型
character_maximum_length	如果 data_type 是 sting 类型，则值的最大字符长度; 否则无效。
character_octet_length	如果 data_type 是字符串，则值的最大长度为 8 字节; 否则无效。
numeric_precision	如果 data_type 是数值型的，则声明的或隐式的精度，否则无效。
numeric_precision_radix	如果 data_type 数值型的，则表示 numeric_precision 和 numeric_scale 列中的值的基数 (2 或 10)。对于所有其他数据类型，column 为 NULL
numeric_scale	如果 data_type 是精确的数字类型，则表示精度
datetime_precision	保留字段
character_set_catalog	保留字段
character_set_schema	保留字段
character_set_name	保留字段
domain_catalog	保留字段
domain_schema	保留字段
domain_name	保留字段
generation_expression	用于计算列类型的计算表达式
is_hidden	是否为隐藏列
hubbledb_sql_type	列的数据类型

6.3.5.6 column_privileges

对于列的授予信息

列名	描述
grantor	授予权限的角色的名称
grantee	被授予权限的角色的名称
table_catalog	包含包含列的表的数据库的名称
table_schema	包含包含列的表的架构的名称

列名	描述
table_name	表明
column_name	列名
privilege_type	授权类型
is_grantable	保留字段

6.3.5.7 constraint_column_usage

某个约束使用的数据库中的所有列。

列名	描述
table_catalog	包含约束使用的列的表的数据库的名称
table_schema	包含表的 schema 的名称，该表包含某些约束使用的列
table_name	包含某些约束使用的列的表的名称
column_name	约束使用的列的名称
constraint_catalog	包含约束的数据库的名称
constraint_schema	包含约束的 schema 名称
constraint_name	约束名称

6.3.5.8 enabled_roles

当前用户的角色。这包括直接作用和间接作用。

列名	描述
role_name	角色名称

6.3.5.9 key_column_usage

使用主键、惟一键或外键/引用约束的列信息

列名	描述
constraint_catalog	包含约束的数据库的名称
constraint_schema	包含约束的 schema 的名称
constraint_name	约束名称
table_catalog	包含约束表的数据库的名称
table_schema	包含约束表的 schema 的名称
table_name	约束表的名称
column_name	约束列的名称
ordinal_position	约束中列的序号位置 (从 1 开始)
position_in_unique_constraint	对于外键约束，引用列在其唯一性约束内的序号位置 (从 1 开始)

6.3.5.10 referential_constraints

列名	描述
constraint_catalog	包含约束的数据库的名称

列名	描述
constraint_schema	包含约束的架构的名称
constraint_name	约束名称
unique_constraint_catalog	包含外键约束引用的唯一或主键约束的数据库的名称 (始终是当前数据库)。
unique_constraint_schema	包含外键约束引用的唯一或主键约束的 schema 的名称
unique_constraint_name	唯一或主键约束的名称
match_option	外键约束的匹配选项:FULL、PARTIAL 或 NONE
update_rule	外键约束的更新规则:CASCADE,SET NULL,SET DEFAULT,RESTRICT, orNO ACTION
delete_rule	外键约束的删除规则:CASCADE,SET NULL,SET DEFAULT,RESTRICT, orNO ACTION
table_name	包含约束的表的名称
referenced_table_name	包含外键约束引用的唯一或主键约束的表的名称

6.3.5.11 role_table_grants

授予者或被授予者是当前启用角色的表或视图上授予了哪些特权。这个表与 table_privileges 相同。

列名	描述
grantor	授予权限的角色的名称
grantee	被授予特权的角色的名称
table_catalog	包含表的数据库的名称
table_schema	包含表的 schema 的名称
table_name	表名
privilege_type	授权名称
is_grantable	保留字段
with_hierarchy	保留字段

6.3.5.12 schema_privileges

数据库级别上为每个用户授予信息。

列名	描述
grantee	授权的用户名
table_catalog	包含约束表的数据库的名称。
table_schema	包含约束表的 schema 名称。
table_schema	特权的名称
table_schema	保留字段

6.3.5.13 schemata

数据库的 schema 信息

列名	描述
table_catalog	数据库名称
table_schema	schema 名称

列名	描述
default_character_set_name	保留字段
sql_path	保留字段

6.3.5.14 sequences

序列标识数据库中定义的序列

列名	描述
sequence_catalog	包含序列的数据库的名称
sequence_schema	包含序列的架构的名称
sequence_name	序列名称
data_type	序列的数据类型
numeric_precision	序列的精度
numeric_precision_radix	表示列 numeric_\precision 和 numeric_scale 的值的基数（进制）。值为 2 或 10
numeric_scale	序列的精度
start_value	序列的第一个值
minimum_value	序列的最小值
maximum_value	序列的最大值
increment	序列递增的值。负数产生一个降序数列。一个正数产生一个升序
cycle_option	目前，所有的序列都被设置为无循环，序列不会换行

6.3.5.15 statistics

表的索引信息

列名	描述
table_catalog	包含约束表的数据库的名称
table_schema	包含约束表的 schema 的名称
table_name	表名
non_unique	如果索引是使用 UNIQUE 创建的，则为 NO; 如果没有使用 UNIQUE 创建索引，则为 YES。
index_schema	包含索引的数据库的名称
index_name	索引名称
seq_in_index	索引中列的序号位置 (从 1 开始)
column_name	被索引的列的名称
collation	保留字段
cardinality	保留字段
direction	升序，降序
storing	如果存储了列，则为 YES; 如果它是索引的或隐式的则为 NO
implicit	如果列是隐式的 (即，未在索引中指定，也未存储) 为 YES; 如果它被索引或存储为 NO

6.3.5.16 table_constraints

应用于表的约束信息

列名	描述
constraint_catalog	包含约束的数据库的名称
constraint_schema	包含约束的 schema 的名称
constraint_name	约束的名称
table_catalog	包含约束表的数据库的名称
table_schema	包含约束表的 schema 的名称
table_name	约束表的名称
constraint_type	约束类型:CHECK、foreign key、PRIMARY KEY 或 UNIQUE
is_deferrable	如果约束可以被延迟，则为 no
initially_deferred	如果约束是可延迟的且最初被延迟，则为 YES；否则为 NO

6.3.5.17 table_privileges

表级别的每个用户的权限信息

列名	描述
grantor	保留字段
grantee	授权用户的用户名
table_catalog	授权应用于的数据库的名称
table_schema	授权应用于的 schema 的名称
table_name	授权应用于的表的名称
privilege_type	授权类型
is_grantable	保留字段
with_hierarchy	保留字段

6.3.5.18 tables

表与视图在数据库中的信息

列名	描述
table_catalog	包含表的数据库的名称
table_schema	包含表的 schema 的名称
table_name	表名
table_type	表的类型: 普通表的基表、视图的视图或系统创建的视图的系统视图
version	表的版本号; 版本从 1 开始，并且每次在表上发出 ALTER TABLE 语句时递增

6.3.5.19 user_privileges

包含了全局权限，且此视图仅包含 root 用户的全局特权

列名	描述
grantee	授权的用户名
table_catalog	应用于的数据库权限的名称
privilege_type	权限类型
is_grantable	保留字段

6.3.5.20 views

数据库中的视图信息

列名	描述
table_catalog	包含视图的数据库的名称
table_schema	包含视图的 schema 的名称
table_name	视图的名称。
view_definition	用于创建视图的子句
check_option	保留字段
is_updatable	保留字段
is_insertable_into	保留字段
is_trigger_updatable	保留字段
is_trigger_deletable	保留字段
is_trigger_insertable_into	保留字段

6.3.6 事务

Hubble 支持将多个 SQL 语句在单个事务中提交。每个事务都能保证在集群中跨多表的 ACID 特性。当一个事务成功，则所有都确保其成功。如果事务的任何部分失败，整个事务将中止，数据库将保持不变。Hubble 保证当一个事务处于挂起状态时，它通过`serializable isolation`事务级别，将其与其他他并发事务中隔离出来

6.3.6.1 事务隔离级别

Hubble 使用最高等级的事务隔离级别，即`SERIALIZABLE`。其他 ANSI 事务隔离级别，如`SNAPSHOT`，`READ UNCOMMITTED`，`READ COMMITED`，和`REPEATABLE READ`将自动升级为`SERIALIZABLE`。

`SERIALIZABLE`事务隔离级别，能确保执行期间，不受其他任何任务的影响。这意味着 Hubble 提供最强大的隔离级别。

6.3.6.2 事务开启、提交与回滚

一个事务可以用`BEGIN`和`COMMIT`语句包裹。用户可以在一个事务中，对若干操作进行事务控制。其使用方式如下：

```
BEGIN;
<transction statements>
COMMIT;
```

只要在未提交前，使用 ROLLBACK 语句即可中止事务。

6.3.6.3 事务操作

Hubble 事务支持的操作如下：

- 写事务操作：`INSERT`、`UPDATE`、`DELETE`。
- 读事务操作：`QUERY`。

Hubble 的其它操作 (如：创建表、创建索引等其它非 CRUD 操作) 不在事务功能的考虑范围。

6.3.6.4 事务优先级

在 Hubble 中，每个事务都有默认优先级，默认为 NORMAL，但是对于在高竞争情况下应该给予优先级的事务，客户端可以在`BEGIN`语句中设置优先级：

```
begin priority <low | normal | high>;
```

客户端可以查看当前事务的优先级

```
show transaction priority;
```

6.3.7 约束

通过对列中的数据强制条件，约束提供了额外的数据完整性。每当操作值（插入、删除或更新）时，将检查约束，并拒绝违反约束的修改。例如，UNIQUE 约束要求列中的所有值彼此唯一（空值除外）。如果尝试写入重复的值，则约束将拒绝整个语句。

6.3.7.1 支持的约束

约束类型	描述信息
CHECK	对于布尔表达式，值必须返回 TRUE 或 NULL。
DEFAULT value	如果在 INSERT 语句中未为受约束列定义值，则将默认值写入该列。
FOREIGN KEY	值必须与它引用的列中的现有值完全匹配。
NOT NULL	值不能为空。
PRIMARY KEY	值必须唯一标识每一行（每个表一行）。这就好像应用了 NOTNULL 和 UNIQUE 约束，并使用约束列自动为表创建索引一样。
UNIQUE	每个非空值必须唯一。这还会使用受约束的列自动为表创建索引。

6.3.7.2 约束条件的使用

6.3.7.2.1 添加约束条件

1. 创建主键约束

```
create table student (student_no int primary key);
```

2. 创建表并在多列上添加主键约束

```
create table trans (cust_no int, cust_id int, primary key (cust_no,cust_id));
```

备注：默认和非空约束不能应用于多个列。

3. 在已创建的表中添加约束条件

```
alter table trans add  constraint id_unique unique (cust_id);
```

备注：PRIMARY KEY和 NOT NULL 约束只能在CREATE TABLE创建时添加。

4. 创建检查约束

```
create table commodities (commodity_id  int primary key,commodity_num int not
    ↪ null check (commodity_num > 0) );
```

5. 创建默认值

```
create table produces (
    product_id  int,
    loc_id      int,
    product_num int  default 100,
    primary key (product_id, loc_id)
  );
```

6. 唯一约束

```
create table login_info (
    login_id      int primary key,
    customer_id   int,
    login_date    timestamp,
    unique (customer_id, login_date)
  );
```

7. 外键约束

- 创建客户表

```
create table customers (cid int primary key, name string );
```

- 创建菜单引用表

```
create table orders (
    oid           int primary key,
    customer_id   int not null REFERENCES customers (cid),
    orderaml      decimal(9,2),
    index (customer_id)
  );
```

6.3.7.3 创建主键时定义约束的名称

```
create table guar (guar_no int constraint another_name primary key);
```

6.3.7.4 列出约束条件

```
show constraints from <tablename>;
```

6.3.7.5 移除约束条件

约束类型	操作
CHECK	Use DROP CONSTRAINT
DEFAULT value	Use ALTER COLUMN
FOREIGN KEY	Use DROP CONSTRAINT
NOT NULL	Use ALTER COLUMN
PRIMARY KEY	不支持删除主键操作。但是，可以使用此过程将表的数据移动到新表中。
UNIQUE	不支持直接删除唯一约束操作。若要删除约束，请删除由该约束创建的索引，例如，删除索引my_unique约束。

6.3.7.6 改变约束条件

约束类型	操作
CHECK	开启事务，添加一个新的 CHECK 约束（ADD constraint），然后删除现有的 CHECK 约束（DROP constraint）。
DEFAULT value	默认值可以通过 ALTER COLUMN 更改。
FOREIGN KEY	开启事务，添加一个新的外键约束（ADD constraint），然后删除现有的外键约束（DROP constraint）。
NOT NULL	不能更改 NOT NULL 约束，只能将其删除。但是，可以使用此过程将表的数据移动到新表中。
PRIMARY KEY	无法修改主键。但是，可以使用此过程将表的数据移动到新表中。
UNIQUE	开启事务，添加一个新的唯一约束（ADD constraint），然后删除现有的唯一约束（DROP constraint）。

6.3.8 窗口函数

6.3.8.1 简述

Hubble 支持在选择查询返回的行的子集上进行函数操作。即为为窗口函数，它允许您通过一次操作多个行来计算值。窗口函数操作的行子集称为窗口框架。有关受支持的窗口函数的完整列表，请参见函数内容。所有聚合函数也可以用作窗口函数。有关更多信息，请参见下面的示例。

6.3.8.2 窗口的定义

窗口使用`over`或 `window`关键字定义，窗口函数，也叫 OLAP 函数 (Online Anallytical Processing，联机分析处理)，可以对数据库数据进行实时分析处理。

- 窗口函数的基本语法如下：

```
<窗口函数> over (
                partition by    <用于分组的列名>
                 order by       <用于排序的列名>
               )
```

因为窗口函数是对`where`或者`group by`子句处理后的结果进行操作，所以窗口函数原则上只能写在`select`子句中。

6.3.8.3 语法及其参数介绍

语法

```
windowFunction  ::=

      window_name  'AS'  '('  existing_window_name partition_clause sort_clause
         ↪  frame_clause ')'
```

参数介绍

参数	说明
window_name	新窗口的名称
existing_window_name	现有窗口框架的可选名称，在不同的窗口定义中定义。
partition_clause	可选的PARTITION BY子句。
sort_clause	可选的ORDER BY子句。
frame_clause	可选的框架子句。

6.3.8.4 说明

使用窗口函数最重要的部分是理解窗口函数将在框架中操作哪些数据。默认情况下，窗口框架包含分区的所有行。如果对分区进行排序，默认框架将包含从分区的第一行到当前行的所有行。换句话说，在创建窗口框架时添加一个ORDER BY子句 (例如，PARTITION BY x ORDER by y) 会产生以下效果:

- 它使窗口框架内的行有序。
- 它改变了函数调用的行，不再是窗口框架中的所有行，而是'第一行'和当前行之间的一个子集。

您应该了解作为窗口函数使用的任何聚合函数的行为。如果您没有看到您期望从窗口函数中看到的结果，这种行为可以解释原因。您可能需要在窗口定义中显式地指定框架边界。

6.3.8.5 作用

在日常工作中，经常会遇到需要在每组内排名，比如下面的业务需求:

- 排名问题：每个部门或者机构按业绩来排名
- topN 问题：找出每个部门排名前 N 的员工进行奖励

面对这类需求，就需要使用 sql 的高级功能窗口函数了。

6.3.8.6 窗口函数机制

窗口功能通过以下方式工作:

- 使用选择查询创建'虚拟表'。
- 将该表分割为具有窗口定义的窗口框架。您可以在OVER子句中直接在窗口函数之后定义窗口框架，也可以在window子句中作为选择查询的一部分定义窗口框架。
- 将窗口函数应用于每个窗口框架。

例如，考虑一个查询，其中为每个窗口函数调用定义了窗口:

```
select *, rank() over (partition by class_no  order by fenshu desc) as ranking
    ↪ from score;
```

其操作可以描述如下:

- 窗口函数 rank()OVER () 在包含查询输出的所有行的窗口框架上进行操作。
- 窗口函数 rank()OVER (PARTITION BY class_no order by fenshu)轮流操作多个窗框，每个框架包含不同班级分区的fenshu列

rank()over()在实际项目中有较多的应用，相比于dense_rank()over(),row_number()over()能够更准的把握分区后取值，以下有示例对比。

6.3.8.7 常见示例

比较 rank()over(),dense_rank()over(),row_number()over()

- 建表并增加测试数据

```
create table score(study_no  varchar(10),class_no  int,fenshu  int);
```

```
    insert into score values('0002',1,95);
    insert into score values('0008',1,84);
    insert into score values('0001',1,86);
    insert into score values('0004',1,86);
    insert into score values('0003',2,89);
    insert into score values('0005',2,86);
    insert into score values('0006',3,92);
    insert into score values('0007',3,86);
```

- rank()over()

```
select *, rank() over (partition by class_no  order by fenshu desc) as ranking
  ↪ from score;
```

```
  study_no | class_no | fenshu | ranking
+----------+----------+--------+---------+
  0002     |        1 |     95 |       1
  0008     |        1 |     84 |       4
  0001     |        1 |     86 |       2
  0004     |        1 |     86 |       2
  0003     |        2 |     89 |       1
  0005     |        2 |     86 |       2
  0006     |        3 |     92 |       1
  0007     |        3 |     86 |       2
```

- dense_rank()over()

```
select *,dense_rank() over (partition by class_no  order by fenshu desc) as
  ↪ dese_rank from score;
```

```
  study_no | class_no | fenshu | dese_rank
+----------+----------+--------+-----------+
  0002     |        1 |     95 |         1
  0008     |        1 |     84 |         3
  0001     |        1 |     86 |         2
  0004     |        1 |     86 |         2
  0003     |        2 |     89 |         1
  0005     |        2 |     86 |         2
  0006     |        3 |     92 |         1
  0007     |        3 |     86 |         2
```

- row_number()over()

```
select *,row_number() over (partition by class_no  order by fenshu desc) as
    ↪ row_num from score;
```

```
  study_no | class_no | fenshu | row_num
+----------+----------+--------+---------+
  0002     |        1 |     95 |       1
  0008     |        1 |     84 |       4
  0001     |        1 |     86 |       2
  0004     |        1 |     86 |       3
  0003     |        2 |     89 |       1
  0005     |        2 |     86 |       2
  0006     |        3 |     92 |       1
  0007     |        3 |     86 |       2
```

- 聚合函数作为窗口函数

```
select *,
avg(fenshu) over (order by study_no) as current_avg,
count(fenshu) over (order by study_no) as current_count,
max(fenshu) over (order by study_no) as current_max,
min(fenshu) over (order by study_no) as current_min
from score;
```

```
 study_no | class_no | fenshu |   current_avg          | current_count |
    ↪ current_max | current_min
----------+----------+--------+-----------------------+---------------+-------------+----
   ↪
  0002     |        1 |     95 |                 90.5  |             2 |
    ↪          95 |          86
  0008     |        1 |     84 |                    88 |             8 |
    ↪          95 |          84
  0001     |        1 |     86 |                    86 |             1 |
    ↪          86 |          86
  0004     |        1 |     86 |                   89  |             4 |
    ↪          95 |          86
  0003     |        2 |     89 |                    90 |             3 |
    ↪          95 |          86
  0005     |        2 |     86 |                88.4   |             5 |
    ↪          95 |          86
  0006     |        3 |     92 |                    89 |             6 |
    ↪          95 |          86
  0007     |        3 |     86 | 88.571428571428571429 |             7 |
    ↪          95 |          86
```

以下示例根据emp消费信息和dept部门信息关系查询相应的数据

- 哪些部门有更多的员工，取前 3

```
SELECT * FROM
    (SELECT distinct(t.dname) as "dname",
            COUNT(*) OVER (PARTITION BY t.dname) AS deptnum
     FROM dept t JOIN emp e  ON e.deptno = t.deptno)
  ORDER BY deptnum desc limit 3;
```

```
   dname    |    deptnum
-----------+-----------
  RESEARCH |       10
ACCOUNTING |       9
     SALES |       6
```

- 查看每个部门的支出情况

```
SELECT DISTINCT t.dname,
    SUM(e.sal) OVER (PARTITION BY t.dname) AS "total revenue"
    FROM dept t JOIN emp e  ON e.deptno = t.deptno
    ORDER BY "total revenue"
```

```
   dname    |    total revenue
-----------+-----------
  RESEARCH |       34904.55
ACCOUNTING |       66360.63
     SALES |       18061.75
OPERATIONS |       5381.50
     BOSS  |       1000.00
```

- 查看哪个部门平均消费，取前 3

```
SELECT DISTINCT t.dname,
    COUNT(*)     OVER w AS "number of all",
    AVG(e.sal) OVER w AS "average revenue"
    FROM dept t JOIN emp e  ON e.deptno = t.deptno
    WINDOW w AS (PARTITION BY t.dname)
    ORDER BY "average revenue" DESC, "number of all" ASC
   limit 3;
```

```
   dname    |   number of all   |  average revenue
-----------+-------------------+--------------
ACCOUNTING |       9           | 7373.40
OPERATIONS |       1           | 5381.50
  RESEARCH |       10          | 3490.46
```

- 查看总的消费记录数及其总额

```
SELECT
    COUNT(a.empno) AS "total people",
    SUM("total rider revenue") AS "total revenue" FROM (
      SELECT DISTINCT e.empno as empno,
             SUM(e.sal) OVER (PARTITION BY e.empno) AS "total rider revenue"
        FROM dept t JOIN emp e  ON e.deptno = t.deptno
        ORDER BY "total rider revenue" DESC) a;
```

```
   total people     |     total revenue
-------------------+-----------------
       27           |      125708.43
```

- 查看每个部分的消费笔数

```
select distinct  d.dname, COUNT(*) over  (partition  by  d.dname) as   dc
from  emp e left join  dept d  on e.deptno=d.deptno order by dc desc;
```

```
   dname    |     dc
-----------+-----------
  RESEARCH  |       10
ACCOUNTING  |       9
     SALES  |       6
OPERATIONS  |       1
     BOSS   |       1
```

6.3.9 分区表

表分区对数据的存储方式和位置进行行级控制。

6.3.9.1 分区的分类

Hubble 中分区表分为两种：

- **值分区**：枚举每个分区的所有可能值。当可能值的数量很少时，列值分区是一个不错的选择。
- **范围分区**：通过指定下限和上限，为每个分区指定一个连续的值范围。当可能值的数量太大而无法明确列出时，范围分区是一个不错的选择。

按列值分区

要按列对表进行分区，请`PARTITION BY LIST`在创建表时使用语法。在定义列分区时，如果没有任何行符合已定义分区的要求，您还可以设置`DEFAULT`作为包罗万象的分区。

按范围分区

要按范围定义表分区，请`PARTITION BY RANGE`在创建表时使用语法。在定义范围分区时，您可以使用 Hubble 定义的`MINVALUE`和`MAXVALUE`参数分别定义范围的下限和上限。

> Hubble 使用分区表的时候，分区字段必须是主键或复合主键的一部分

HUBBLE

6.3.9.2 分区的查询

首先查看表的CREATE TABLE语句

```
create table  cust_info (
       cust_id INT NOT NULL,
       city VARCHAR(10) NOT NULL,
       cust_name VARCHAR(10) NULL,
       address VARCHAR(10) NULL,
       cust_card VARCHAR(10) NULL,
       CONSTRAINT "primary" PRIMARY KEY (city ASC, cust_id ASC),
       FAMILY "primary" (cust_id, city, cust_name, address, cust_card)
        ) PARTITION BY LIST (city) (
          PARTITION p1 VALUES IN (('shanghai'), ('nanjing')),
          PARTITION p2 VALUES IN (('beijing'), ('tangshan') ),
          PARTITION p3 VALUES IN (('xian'), ('xining')),
          PARTITION DEFAULT VALUES IN (default)
        );
```

- 过滤非索引列

假设您要查询表以获取有关特定用户的信息，但您只知道该用户的名称

```
explain select * from cust_info where cust_name='zhangsan';
```

```
                                   info
-------------------------------------------------------------------------------------
  ↪
  distribution: full
  vectorized: true

  • filter
    estimated row count: 1
    filter: cust_name = 'zhangsan'

      • scan
        estimated row count: 1 (100% of the table; stats collected 41 minutes
          ↪ ago)
        table: cust_info@primary
        spans: FULL SCAN
```

该查询返回相同的结果，但由于cust_name不是索引列，该查询执行跨所有分区值的全表扫描。

- 在分区列上过滤

如果知道哪个分区包含您正在查询的数据，则在用于该分区的列上使用过滤器（例如，WHERE子句）可以通过将扫描限制到包含数据的特定分区来进一步提高性能你正在查询。

现在假设您知道用户的姓名和位置。您可以使用对用户名和城市的过滤器来查询表

```
explain select * from cust_info where cust_name='zhangsan' and city='beijing';
```

```
                                        info
--------------------------------------------------------------------------------
  ↪
  distribution: local
  vectorized: true

  • filter
    estimated row count: 1
    filter: cust_name = 'zhangsan'

      • scan
        estimated row count: 1 (100% of the table; stats collected 3 minutes ago
          ↪ )
        table: cust_info@primary
        spans: [/'beijing' - /'beijing']
```

该表返回与以前相同的结果，但成本要低得多，因为查询扫描现在只跨越 beijing 分区值。

6.3.9.3 分区表举例

- 按列值分区，用到PARTITION BY LIST语句

```
CREATE TABLE t1 (
    id INT ,
    name STRING,
    az STRING,
    create_ DATE,
    PRIMARY KEY (az,id))
    PARTITION BY LIST (az) (
      PARTITION p1 VALUES IN ('A'),
      PARTITION p2 VALUES IN ('B'),
      PARTITION p3 VALUES IN ('C','D'),
      PARTITION DEFAULT VALUES IN (default)
    );
```

分区结构查询

```
show partitions from table t1;
```

- 按范围分区，用到PARTITION BY RANGE语句

```
CREATE TABLE t2 (
   cust_id INT ,
   cust_name STRING,
   email STRING,
   create_date DATE,
   PRIMARY KEY (create_date, cust_id)
   )
   PARTITION BY RANGE(create_date)
```

```
    (PARTITION p1 VALUES FROM (MINVALUE) TO ('2021-06-30'),
    PARTITION p2 VALUES FROM ('2021-06-30') TO (MAXVALUE));
```

分区结构查询

```
show partitions from table t2;
```

重设分区表

通过使用该PARTITION BY命令的子命令来实现这一点ALTER TABLE

```
ALTER TABLE t2 PARTITION BY RANGE (create_date) (
    PARTITION p1 VALUES FROM (MINVALUE) TO ('2021-03-31'),
    PARTITION p2 VALUES FROM ('2021-03-31') TO ('2021-06-30'),
    PARTITION p3 VALUES FROM ('2021-06-30') TO ('2021-09-30'),
    PARTITION p4 VALUES FROM ('2021-09-30') TO (MAXVALUE));
```

取消分区

可以使用以下PARTITION BY NOTHING语法删除表上的分区

```
alter  table t2 partition by nothing;
```

不能删除某个特定的分区

Hubble 不支持分区表数据的批量加载与删除，如果表需要频繁批量的加载与删除，不建议做表的分区。

6.3.10 视图

Hubble 的视图不是物化的：它们不存储基础查询的结果。相反，每次使用该视图时都会重新执行基础查询。

所需权限

用户必须具有CREATE权限、SELECT以及视图引用的任何表的权限

语法图

```
CreateViewStmt ::=
    'CREATE' OrReplace  'VIEW' IfNotExists ViewName  'AS' ViewSelectStmt

IfNotExists ::=

    'IF' 'NOT' 'EXISTS'

OrReplace ::=
    ( 'OR' 'REPLACE' )?
```

语法图

参数	简介
IF NOT EXISTS	仅当不存在同名视图时才创建新视图。请注意，IF NOT EXISTS仅检查视图名称。它不检查现有视图是否与新视图具有相同的列。
OR REPLACE	如果不存在同名视图，则创建一个新视图。如果已存在同名视图，请替换该视图
view_name	要创建的视图的名称，在其数据库中必须是唯一的，并遵循这些标识符规则。

6.3.10.1 创建视图

- 要创建视图，请使用以下CREATE VIEW语句：

```
create view hubble_db.user_info
  as
select  cust_no, cust_name,cust_card_no from  cust_info;
```

6.3.10.2 列表视图

- 创建后，视图将在数据库中的常规表旁边列出：

```
show  tables from  hubble_db;
```

```
  schema_name | table_name | type  | owner | estimated_row_count | locality
--------------+------------+-------+-------+---------------------+-----------
  public      | cust_info  | table | root  |                   0 | NULL
  public      | user_info  | view  | root  |                   0 | NULL
```

- 要仅列出视图，可以views在信息模式中查询表：

```
select * from hubble_db.information_schema.views;
```

```
  table_catalog | table_schema | table_name |
     ↪ view_definition                          |
----------------+--------------+------------+-------------------------------------------
   ↪
  hubble_db     | public       | user_info  | SELECT cust_no, cust_name,
     ↪ cust_card_no FROM hubble_db.public.cust_info | NULL
```

6.3.10.3 查询视图

- 要查询视图，请使用表表达式（例如，使用SELECT子句）作为目标，就像使用存储表一样：

```
select * from  hubble_db.user_info;
```

- 要检查SELECT视图执行的语句，请使用以下SHOW CREATE语句：

```
show  create hubble_db.user_info;
```

HUBBLE

6.3.10.4 重命名视图

- 要重命名视图，请使用以下ALTER VIEW语句：

```
alter view hubble_db.user_info rename to hubble_db.user_info_bak;
```

6.3.10.5 删除视图

- 要删除视图，请使用以下DROP VIEW语句

```
drop view hubble_db.user_info_bak;
```

6.3.11 会话

6.3.11.1 会话变量

有关于会话变量在实际应用中可能会用到，而会话变量的类型也比较多，常见的值主要有以下几种：

- enable_experimental_alter_column_type_general: 默认值为false，如设置为true，则在一般情况下用作列类型更改。
- experimental_enable_hash_sharded_indexes: 默认值为off，如果设置为on，用于散列分片索引。
- experimental_enable_temp_tables: 默认值为off，如果设置为on，用于临时对象，包括临时表、临时视图。

6.3.11.1.1 列类型更改

对更改列数据类型的支持是的,有一定的限制。要启用列类型更改,请将enable_experimental_alter_column_type_general
↪ 会话变量设置为 true. 以下在 Hubble 中是等价的：

- ALTER TABLE ... ALTER ... TYPE
- ALTER TABLE ... ALTER COLUMN TYPE
- ALTER TABLE ... ALTER COLUMN SET DATA TYPE

> 更改数据类型的限制，在以下情况下，不能更改列的数据类型：

- 该列是索引的一部分。
- 该列有CHECK约束。
- 该列拥有一个序列。
- 该ALTER COLUMN TYPE声明是合并ALTER TABLE声明的一部分。

> string 改成 varchar(n)，必须保证 n 大于现有数据的最大长度，否则会导致数据缺失。字段只能进行扩长，不能进行缩减。要更改数据的类型，首先要将会话变量设置为 true，可以理解为先开会话，后改数据类型。

数据类型转换

- 表结构

```
create table cust(id varchar(20),
age int,
sal DECIMAL(12,2),
pers DECIMAL(12,2));
```

- 将会话变量设置为true

```
SET enable_experimental_alter_column_type_general = true;
```

- int类型改为string类型

```
alter table cust alter age type string;
```

- DECIMAL改为string

```
alter table cust alter sal type string;
```

- 查看信息

```
show  columns from cust;
```

```
 column_name |   data_type    | is_nullable | column_default |
    ↪ generation_expression |  indices   | is_hidden
--
   ↪ ------------+---------------+-------------+----------------+-----------------------+
   ↪
  id           | VARCHAR(20)   |    true     | NULL           |
     ↪                        | {}        |   false
  age          | STRING        |    true     | NULL           |
     ↪                        | {}        |   false
  sal          | STRING        |    true     | NULL           |
     ↪                        | {}        |   false
  pers         | DECIMAL(12,2) |    true     | NULL           |
     ↪                        | {}        |   false
  rowid        | INT8          |    false    | unique_rowid() |
     ↪                        | {primary} |   true
```

更改数据类型精度

- DECIMAL类型精度改变只能变大，不能变小

```
ALTER TABLE cust ALTER pers TYPE DECIMAL(14,2);
```

- 查看信息

```
show  columns from cust;
```

```
 column_name |   data_type    | is_nullable | column_default |
    ↪ generation_expression |  indices   | is_hidden
--
   ↪ ------------+---------------+-------------+----------------+-----------------------+
   ↪
  id           | VARCHAR(20)   |    true     | NULL           |
     ↪                        | {}        |   false
  age          | STRING        |    true     | NULL           |
     ↪                        | {}        |   false
```

HUBBLE

```
  sal          | STRING         |    true      | NULL            |
    ↪                           | {}         |   false
  pers         | DECIMAL(14,2)  |    true      | NULL            |
    ↪                           | {}         |   false
  rowid        | INT8           |    false     | unique_rowid()  |
    ↪                           | {primary}  |   true
```

6.3.11.1.2 临时表

细节说明

- 临时表会在会话结束时自动删除。
- 只能从创建临时表的会话中访问临时表。
- 临时表在同一会话中的事务中持续存在。
- 临时表可以引用持久表，但持久表不能引用临时表。
- 临时表不能转换为持久表。

创建临时表

- 前置条件：要使用临时表，需要设置experimental_enable_temp_tables为 on

```
SET experimental_enable_temp_tables=on;
```

- 创建样例

```
CREATE TEMP TABLE users (
        id UUID,
        city STRING,
        name STRING,
        address STRING,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC)
);
```

- SHOW CREATE 查看临时表

```
SHOW CREATE TABLE users;
```

```
 table_name |                    create_statement
-------------+-------------------------------------------------------------
  users      | CREATE TEMP TABLE pg_temp_1648014738602055526_1.users (
             |     id UUID NOT NULL,
             |     city STRING NOT NULL,
             |     name STRING NULL,
             |     address STRING NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
             |     FAMILY "primary" (id, city, name, address)
             | )
```

- 创建领一张临时表引用 users

```
CREATE TEMP TABLE vehicles (
        id UUID NOT NULL,
        city STRING NOT NULL,
        type STRING,
        owner_id UUID,
        creation_time TIMESTAMP,
        CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
        CONSTRAINT fk_city_ref_users FOREIGN KEY (city, owner_id) REFERENCES
           ↪ users(city, id)
);
```

```
SHOW CREATE TABLE vehicles;
```

```
  table_name |                                        create_statement
--
   ↪ -----------+-------------------------------------------------------------------------------
   ↪
  vehicles   | CREATE TEMP TABLE vehicles (
             |     id UUID NOT NULL,
             |     city STRING NOT NULL,
             |     type STRING NULL,
             |     owner_id UUID NULL,
             |     creation_time TIMESTAMP NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
             |     CONSTRAINT fk_city_ref_users FOREIGN KEY (city, owner_id)
               ↪ REFERENCES users(city, id),
             |     INDEX vehicles_auto_index_fk_city_ref_users (city ASC,
               ↪ owner_id ASC),
             |     FAMILY "primary" (id, city, type, owner_id, creation_time)
             | )
(1 row)
```

- 显示会话中所有临时表

```
SHOW TABLES FROM pg_temp;
```

```
         schema_name          | table_name | type  | owner | estimated_row_count
            ↪  | locality
--
   ↪ ----------------------------+------------+-------+-------+---------------------+--
   ↪
  pg_temp_1648014738602055526_1 | users      | table | root  |
     ↪                    0 | NULL
  pg_temp_1648014738602055526_1 | vehicles   | table | root  |
     ↪                    0 | NULL
```

- 取消会话

```
SHOW session_id;
```

```
             session_id
-------------------------------------
  15fd69f9831c1ed00000000000000001
(1 row)
```

```
CANCEL SESSION '15fd69f9831c1ed00000000000000001';
```

6.3.11.1.3 临时视图

细节说明

- 会话结束时会自动删除临时视图。
- 临时视图只能从创建它的会话中访问。
- 临时视图在同一会话中的事务中持续存在。
- 临时视图不能转换为持久视图。

创建临时视图

- 前置条件：要使用临时视图，需要设置experimental_enable_temp_tables为 on

```
SET experimental_enable_temp_tables=on;
```

- 视图样例

```
 CREATE TEMP VIEW temp_user_view (sname ,sage )
as select
name,
age
from custs;
```

```
SELECT * FROM temp_user_view;
```

```
 sname   | sage
---------+-------
  张三丰 |   20
  李四娘 |   20
```

6.3.11.2 sql 语句

6.3.11.2.1 sql 审计日志保留

- 保留 SQL 审计日志：出于安全目的，将对表的所有查询记录到文件中

```
ALTER TABLE t EXPERIMENTAL_AUDIT SET READ WRITE;
```

6.3.12 名称解析

要在查询中引用一个对象 (例如，一个表)，可以指定一个数据库、一个模式或者两者都指定，或者都不指定。为了解析查询引用的对象，Hubble 会按照一组规则扫描适当的命名空间。

6.3.12.1 命名层次结构

为了与 PostgreSQL 兼容，Hubble 支持 names 的三级结构。这称为命名层次结构。

在命名层次结构中，存储对象的路径包含三个部分：

- 数据库名称
- 模式名称
- 对象名称

Hubble 集群可以存储多个数据库。每个数据库可以存储多个模式，每个模式可以存储多个表、视图、序列和用户定义的类型。

首次启动集群时，会包含许多预加载的数据库和模式，包括`defaultdb`数据库和`public`模式。默认情况下，对象 (例如，表) 存储在当前数据库 (默认情况下) 中的预加载`public`模式中。

要创建新数据库，请使用`CREATE DATABASE`语句；要创建新模式，请使用`CREATE SCHEMA`语句。所有数据库的列表可以通过`SHOW DATABASES`查看，可以使用`SHOW SCHEMAS`获取给定数据库的所有模式的列表，可以使用其他`SHOW` ↪ 语句获得给定模式的所有对象的列表。

6.3.12.2 从以前版本的数据库迁移命名空间

建议使用模式命名空间，而不是数据库命名空间，来创建命名结构。如果您要升级，请在升级完成后执行以下操作的任意组合：

- 在集群上的数据库中创建新模式。创建模式后，根据需要使用`ALTER TABLE ... RENAME`、`ALTER SEQUENCE` ↪ `... RENAME`、`ALTER TYPE ... RENAME`或`ALTER VIEW ... RENAME`语句在数据库之间移动对象。要在模式之间移动对象，请使用`ALTER TABLE ... SET SCHEMA`、`ALTER SEQUENCE ... SET SCHEMA`或`ALTER` ↪ `VIEW ... SET SCHEMA`。
- 如果您的集群包含跨数据库引用 (例如，跨数据库外键引用或跨数据库视图引用)，请使用相关`ALTER TABLE` ↪ 、`ALTER SEQUENCE`、`ALTER TYPE`或`ALTER VIEW`语句将任何交叉引用对象移动到同一数据库，但不同模式。早期版本允许跨数据库对象引用，以使数据库对象命名层次结构对用户更加灵活。

6.3.12.3 名称解析的工作原理

名称解析分别发生以查找现有对象并确定新对象的全名。

查找现有对象的规则如下：

- 如果名称已经完全指定了数据库和模式，请使用该信息。
- 如果名称具有单组件前缀（例如，模式名称），请尝试在当前数据库和当前模式中查找具有前缀名称的模式。如果失败，请尝试在 public 具有前缀名称的数据库模式中查找对象。
- 如果名称没有前缀，则使用当前数据库的搜索路径。

同样，决定新对象全名的规则如下：

- 如果名称已经完全指定了数据库和模式，请使用它。
- 如果名称具有单组件前缀（例如，模式名称），请尝试查找具有该名称的模式。如果不存在这样的架构，请使用 public 数据库中带有前缀名称的架构。
- 如果名称没有前缀，则使用当前数据库中的当前模式。

6.3.12.4 名称解析参数

6.3.12.4.1 当前数据库

当前数据库在名称不合格或只有一个组件前缀时使用。它是`database`会话变量的当前值。

- 您可以使用`SHOW database`查看会话变量的当前值，并使用`SET database`更改它。
- 对于使用表单 URL 连接到数据库的客户端应用程序，可以使 URL 的路径组件设置会话变量 postgres://... 的初始值。例如，postgres://node/mydb 设置`database`为`mydb`建立连接。

6.3.12.4.2 搜索路径

当名称不合格（即没有前缀）时使用搜索路径。它列出了查找对象的模式。它的第一个元素也是创建新对象的当前模式。

- 您可以使用`SET search_path`设置当前搜索路径并使用`SHOW search_path`进行检查。
- 您可以用`SHOW SCHEMAS`列出的有效模式列表。
- 默认情况下，搜索路径包含`$user`、`public`、`pg_catalog`和`pg_extension`。为了与 PostgreSQL 兼容，`pg_catalog`必须始终存在`search_path`，即使未使用`SET search_path`。

6.3.12.4.3 当前架构

如果名称没有前缀，则在创建新对象时将当前模式用作目标模式。

- `search_path`为了与 PostgreSQL 兼容，当前模式始终是默认第一个值。
- 可以使用特殊的内置函数检查当前模式`current_schema()`。

6.3.12.4.4 索引名称解析

hubble 支持以下方法来为需要一个索引名称的语句指定索引名称（例如`DROP INDEX`、`ALTER INDEX ... RENAME` ↪ ）:

- 索引名称使用`@`字符（例如）相对于表名称进行解析`DROP INDEX tbl@idx`，这是默认和最常见的语法。
- 索引名称是通过搜索当前模式中的所有表来解析的，以查找具有名为`idx`的索引的表，例如`DROP INDEX` ↪ `idx`或（具有可选模式前缀）`DROP INDEX public.idx`; 这种语法对于 PostgreSQL 兼容性是必要的，因为 PostgreSQL 索引名称存在于模式命名空间中。

6.4 工具

6.4.1 dbeaver 连接数据库

6.4.1.1 下载

下载地址为 https://dbeaver.io/download/

6.4.1.2 安装步骤

安装步骤比较简单，全部下一步或者接受即可

6.4.1.3 工具连接案例

6.4.1.3.1 步骤一: 新建连接

6.4.1.3.2 步骤二: 选择 SQL 下的 postgresql

6.4.1.3.3 步骤三: 测试

6.4.1.3.4 步骤四: 新建模式

HUBBLE

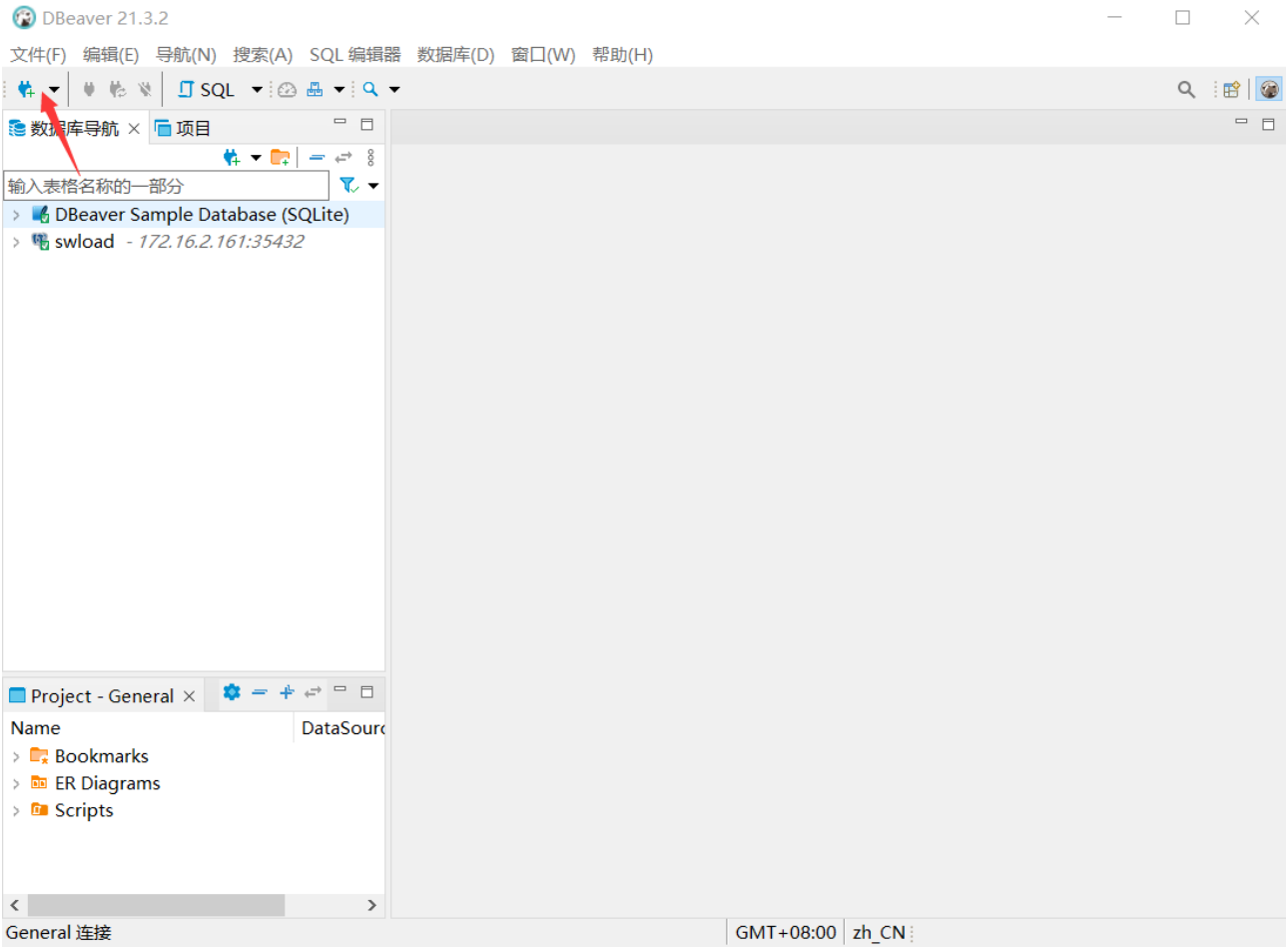

DBeaver 21.3.2
文件(F) 编辑(E) 导航(N) 搜索(A) SQL 编辑器 数据库(D) 窗口(W) 帮助(H)
SQL
数据库导航
项目
输入表格名称的一部分
DBeaver Sample Database (SQLite)
swload - 172.16.2.161:35432
Project - General
Name
DataSourc
Bookmarks
ER Diagrams
Scripts
General 连接
GMT+08:00
zh_CN

图 2: 图 1

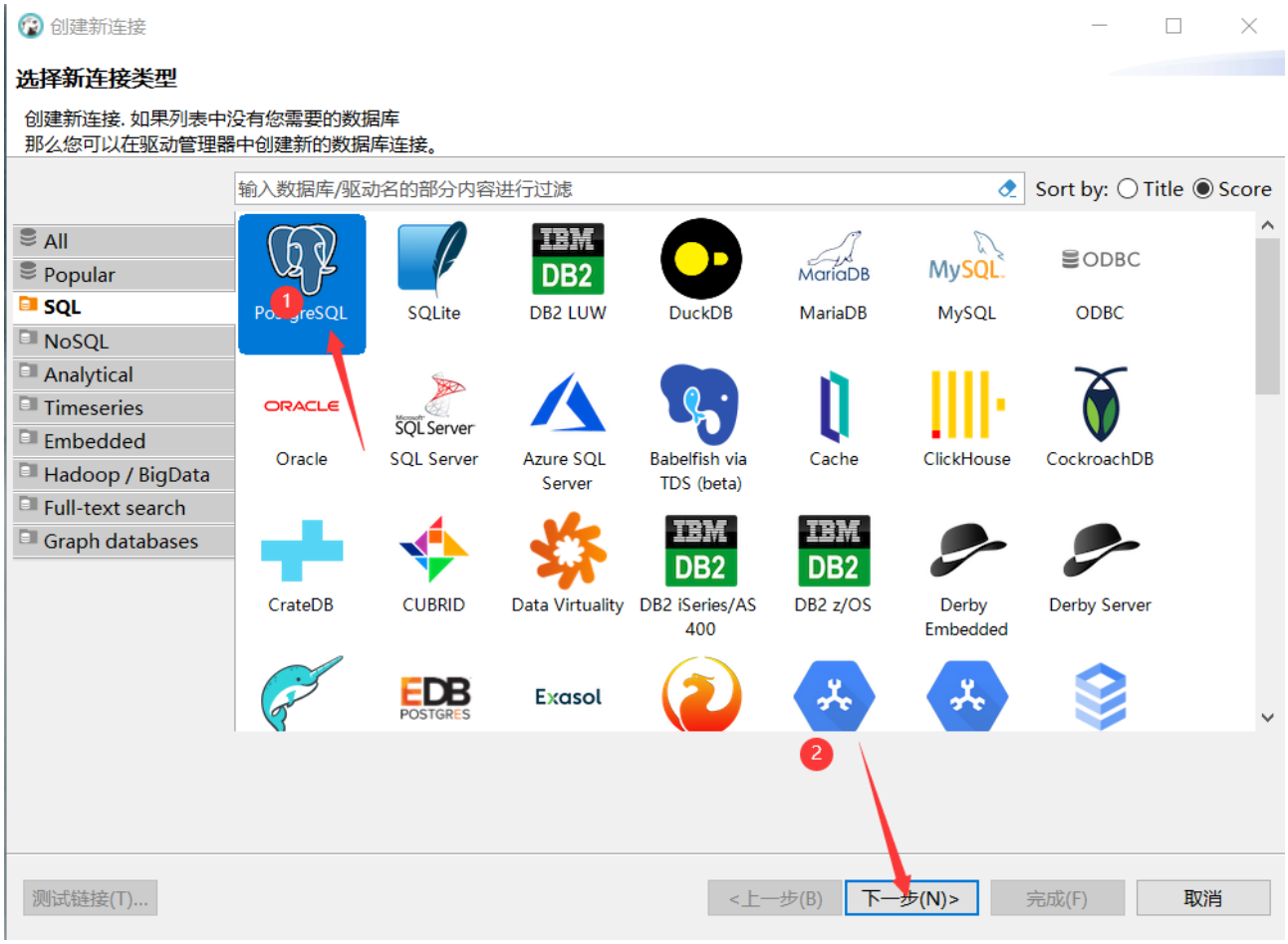

创建新连接
选择新连接类型
创建新连接. 如果列表中没有您需要的数据库
那么您可以在驱动管理器中创建新的数据库连接。
输入数据库/驱动名的部分内容进行过滤
Sort by: Title Score
All
Popular
SQL
NoSQL
Analytical
Timeseries
Embedded
Hadoop / BigData
Full-text search
Graph databases
PostgreSQL
SQLite
DB2 LUW
DuckDB
MariaDB
MySQL
ODBC
Oracle
SQL Server
Azure SQL Server
Babelfish via TDS (beta)
Cache
ClickHouse
CockroachDB
CrateDB
CUBRID
Data Virtuality
DB2 iSeries/AS 400
DB2 z/OS
Derby Embedded
Derby Server
Exasol
1
2
测试链接(T)...
<上一步(B)
下一步(N)>
完成(F)
取消

图 3: 图 2

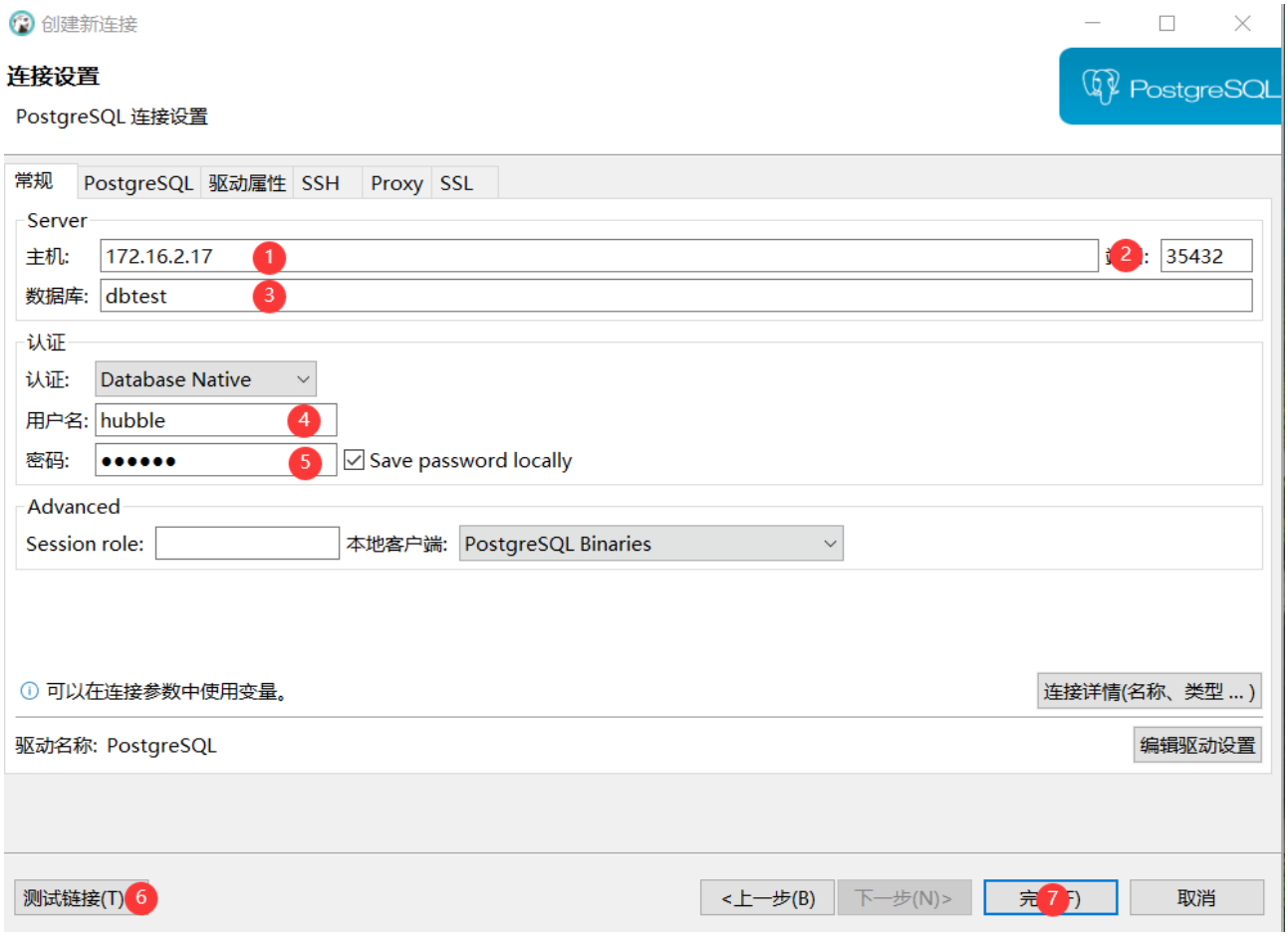

创建新连接
连接设置
PostgreSQL 连接设置
PostgreSQL
常规
PostgreSQL
驱动属性
SSH
Proxy
SSL
Server
主机:
172.16.2.17
1
2
35432
数据库:
dbtest
3
认证
认证:
Database Native
用户名:
hubble
4
密码:
5
Save password locally
Advanced
Session role:
本地客户端:
PostgreSQL Binaries
可以在连接参数中使用变量。
连接详情(名称、类型 ...)
驱动名称: PostgreSQL
编辑驱动设置
测试链接(T)
6
<上一步(B)
下一步(N)>
7
取消

图 4: 图 3

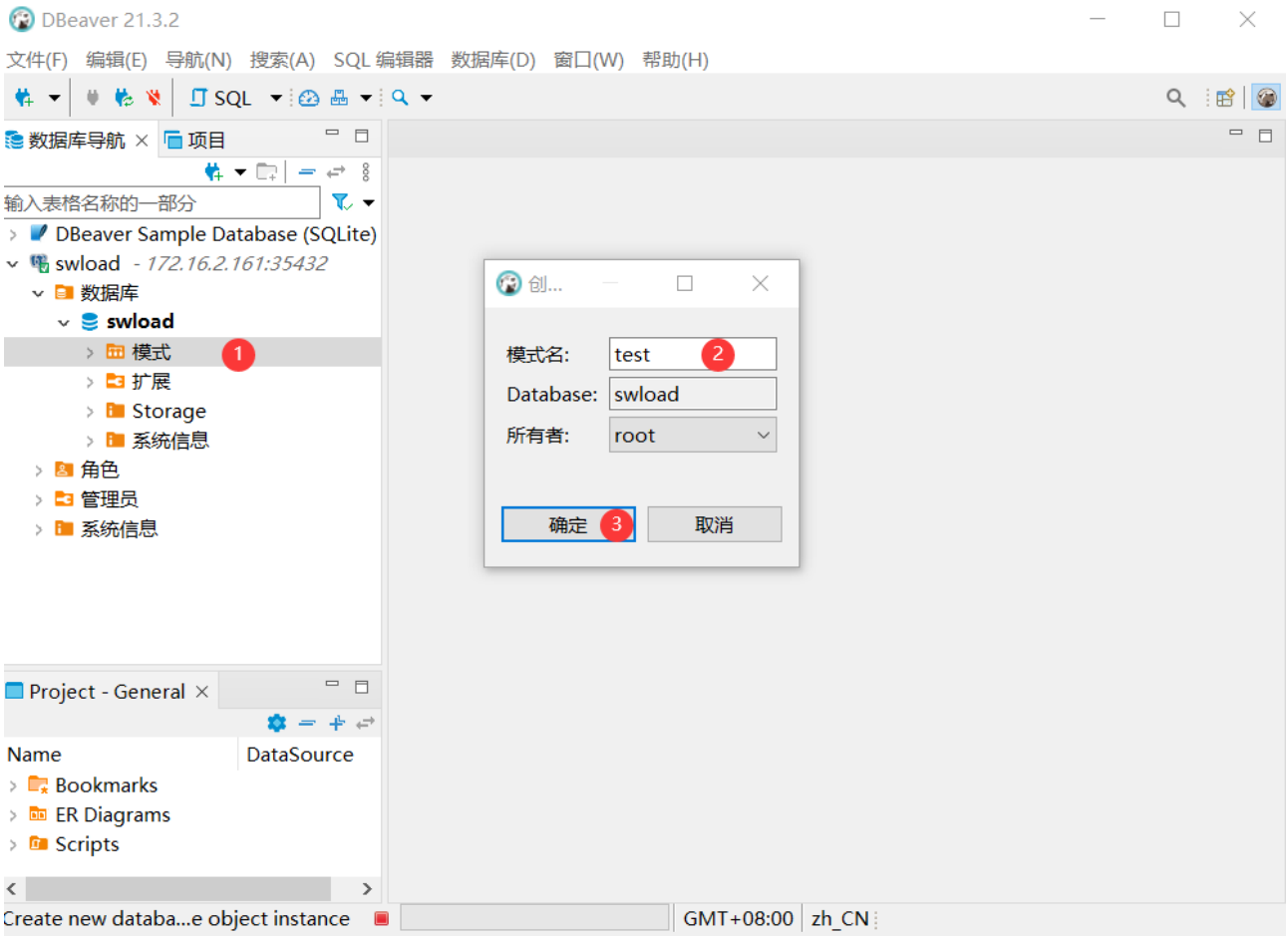

DBeaver 21.3.2
文件(F) 编辑(E) 导航(N) 搜索(A) SQL 编辑器 数据库(D) 窗口(W) 帮助(H)
SQL
数据库导航
项目
输入表格名称的一部分
DBeaver Sample Database (SQLite)
swload - 172.16.2.161:35432
数据库
swload
模式
1
扩展
Storage
系统信息
角色
管理员
系统信息
创...
模式名:
test
2
Database:
swload
所有者:
root
确定
3
取消
Project - General
Name
DataSource
Bookmarks
ER Diagrams
Scripts
Create new databa...e object instance
GMT+08:00
zh_CN

图 5: 图 4

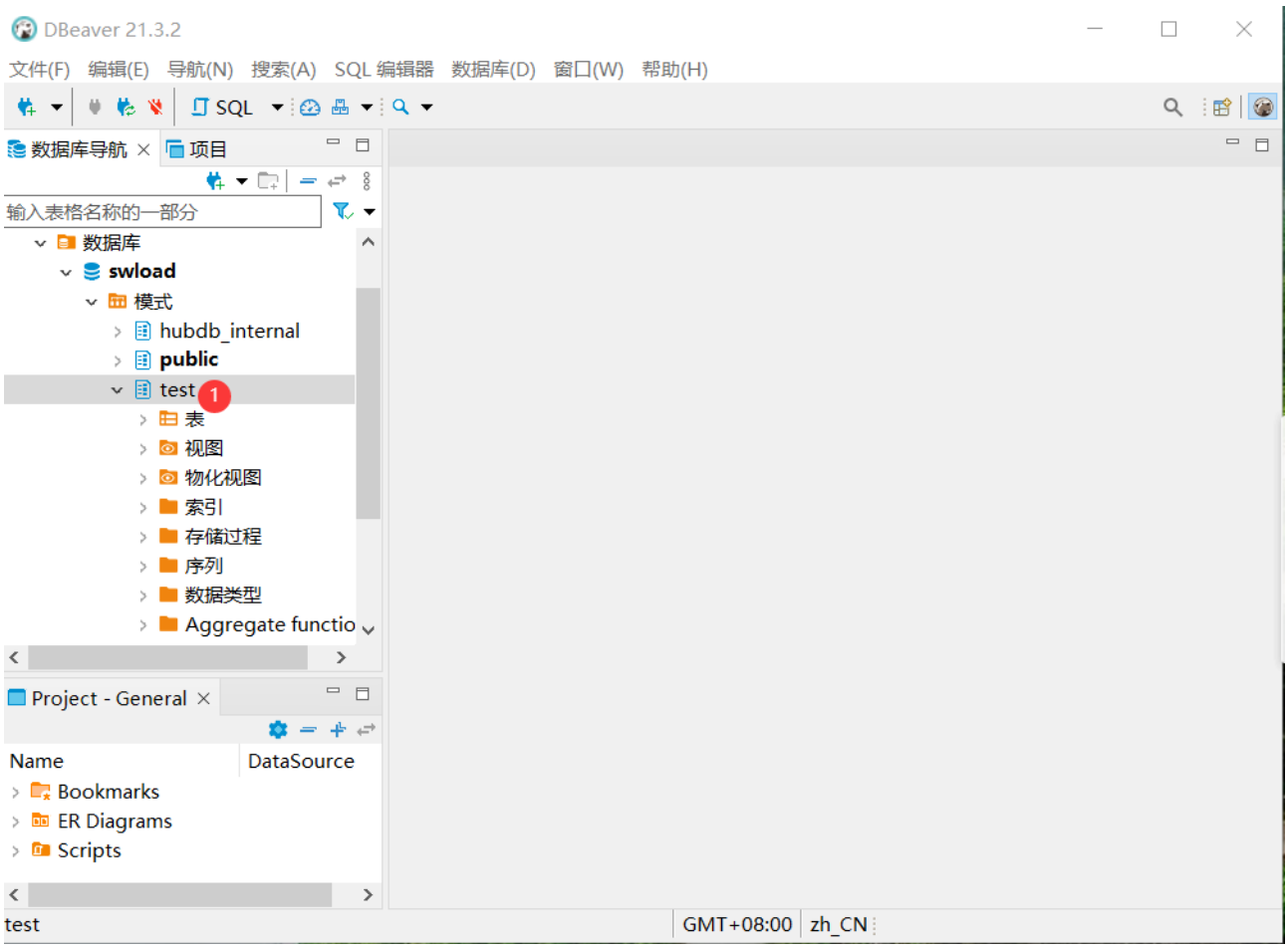

DBeaver 21.3.2
文件(F) 编辑(E) 导航(N) 搜索(A) SQL 编辑器 数据库(D) 窗口(W) 帮助(H)
SQL
数据库导航
项目
输入表格名称的一部分
数据库
swload
模式
hubdb_internal
public
test
1
表
视图
物化视图
索引
存储过程
序列
数据类型
Aggregate functio
Project - General
Name
DataSource
Bookmarks
ER Diagrams
Scripts
test
GMT+08:00
zh_CN

图 6: 图 5

6.4.1.3.5 步骤五: 新建 SQL 编辑器

sql 样例 (根据图片的标记，首先确认库名和模式)

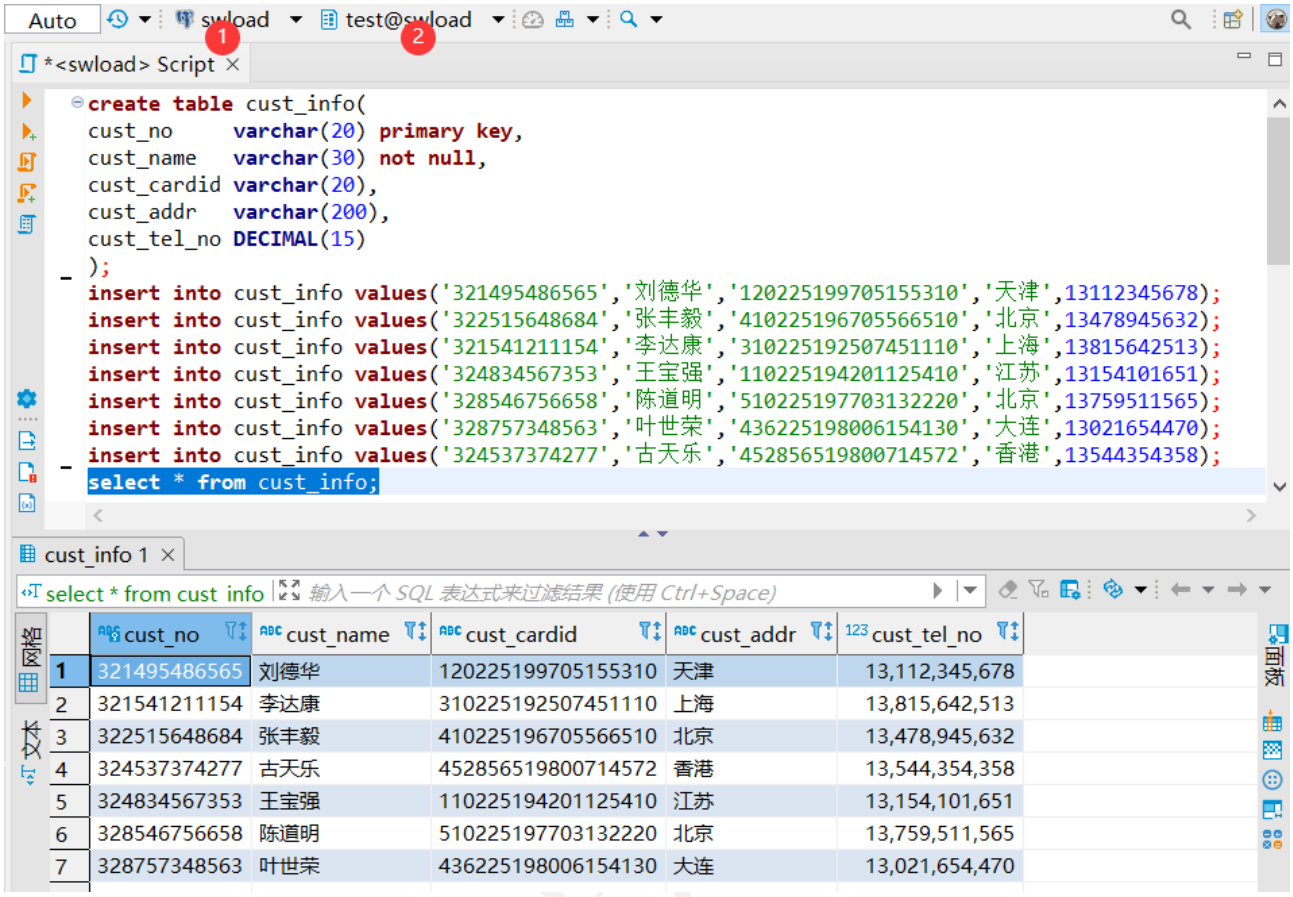


图 7: 图 6

7 故障诊断

7.1 诊断报告

默认情况下，数据库控制台和 Hubble 集群的每个节点与 Hubble 共享匿名使用详细信息。这些细节完全清除了可识别信息，极大地帮助我们了解和改进系统在现实场景中的行为方式。

此页面总结了共享的详细信息、如何自己查看详细信息以及如何选择退出共享。

要深入了解集群的性能和健康状况，请使用内置监控或第三方监控工具 Prometheus。

7.1.1 分享的内容

当诊断报告开启时，Hubble 集群的每个节点每小时共享匿名详细信息，包括（但不限于）:

- 部署和配置特征，例如硬件大小、已从默认值更改的集群设置、配置的复制区域数量等。
- 使用情况和集群运行状况详细信息，例如崩溃、意外错误、尝试使用不受支持的功能、运行的查询类型及其执行特征以及使用的模式类型等。

7.1.2 选择退出诊断报告

7.1.2.1 在集群初始化时

为确保绝对不共享任何诊断详细信息，您可以在启动集群的第一个节点之前设置环境变量。请注意，这仅在启动集群的第一个节点之前设置时才有效。集群运行后，您需要使用SET CLUSTER SETTING下面描述的方法。

7.1.2.2 集群初始化后

要在集群运行后停止向发送诊断详细信息，请使用内置 SQL 客户端执行以下SET CLUSTER SETTING语句，将diagnostics.reporting.enabled集群设置切换为false：

```
SET CLUSTER SETTING diagnostics.reporting.enabled = false;
```

此更改不会是即时的，因为它必须传播到集群中的其他节点。

7.1.3 检查诊断报告的状态

要检查诊断报告的状态，请使用内置 SQL 客户端执行以下SHOW CLUSTER SETTING语句：

```
SHOW CLUSTER SETTING diagnostics.reporting.enabled;
```

```
  diagnostics.reporting.enabled
+-------------------------------+
              false
(1 row)
```

7.2 慢 sql 设置分析

Hubble 数据库如果 SQL 语句返回意外结果或处理时间超过预期，则可以通过设置来输出查看慢 sql。

7.2.1 识别慢 SQL

- 方式 1：使用慢 sql 日志查询
- 方式 2：Hubble 页面来查看慢 sql

> 采用方式 1 慢 sql 日志查询的形式为超过预期的 sql，将会持久记录到 sql-slow.log 中，因需要过滤筛选慢 sql 性能下降 10% 左右。默认未开启。
> 采用方式 2 hubble 页面来查看慢 sql，最大设置可以查看 24h 之内 sql。

7.2.2 设置与查看慢 sql 方法

7.2.2.1 方式 1

7.2.2.1.1 慢 sql 日志设置

- 设置 sql.log.slow_query.latency_threshold 为您所选择的阈值。例如，60 秒表示用户感觉到的系统在瞬间做出反应的极限。

```
SET CLUSTER SETTING sql.log.slow_query.latency_threshold = '60s';
```

- 要将慢速内部查询写入一个单独的日志，请将 sql.log.slow_query.internal_queries.enabled 集群设置为 true

```
SET CLUSTER SETTING sql.log.slow_query.internal_queries.enabled = true;
```

- 每个节点的慢 SQL 日志默认写入 Hubble 数据库日志目录下

7.2.2.1.2 慢 sql 日志查询

进入 Hubble 数据库日志目录 /data/hubbledir310/logs/ 查看sql-slow.log文件

```
[root@poc-hubble04 logs]# cd /data/hubbledir310/logs/
[root@poc-hubble04 logs]# tail -n 4 hubble310-sql-slow.log
```

```
I220613 08:57:03.414866 1147934625 util/log/file_sync_buffer.go:238   [config]
    ↪   log format (utf8= ): hubdb-v2
I220613 08:57:03.414873 1147934625 util/log/file_sync_buffer.go:238   [config]
    ↪   line format: [IWEF]yymmdd hh:mm:ss.uuuuuu goid [chan@]file:
    ↪ lineredactionmark \[tags\] [counter] msg
I220613 08:57:03.412822 1147934625 10@util/log/event_log.go:32   [n3,client
    ↪ =<192.168.1.11:50628>,hostssl,user=root] 1 ={"Timestamp
    ↪ ":1655110514810544287,"EventType":"slow_query","Statement":"<SELECT * FROM
    ↪  \"\".\"\".bptfhist_2y2 AS t WHERE main_ac LIKE '%10268427281' LIMIT 2>","
    ↪ Tag":"SELECT","User":"root","ApplicationName":"$ hubble sql","ExecMode":"
    ↪ exec","Age":108602.01,"FullTableScan":true,"TxnCounter":6}
```

查看 sql 语句，进行慢 sql 分析。

7.2.2.2 方式 2

进入 Hubble 页面控制台，单击左侧的 '语句列表' ，查看 '语句列表' 页面。

8 常见问题 FAQ

8.1 集群常见问题

8.1.1 节点启动常见问题

8.1.1.1 节点无法启动报 `500ms`

```
clock synchronization error: this node is more than 500ms away from at least
    ↪ half of the known nodes
```

解决方案: 这个错误表明一个节点服务关闭了，因为检测到它的时钟与集群中至少一半的其他节点不同步，误差是允许的最大偏移量（默认为 500ms）。为了保持数据一致性，Hubble 数据库需要适当的时钟同步级别，因此通过这种方式关闭节点可以避免出现一致性异常的风险。检查服务器时间同步工具，保证服务器之间时钟同步一致。

8.1.1.2 节点宕机报 `open file descriptor limit`

```
open file descriptor limit of <number> is under the minimum required <number>
```

解决方案: Hubble 数据库打开大量的文件描述符，通常比默认情况下可用的还要多。信息表明 Hubble 数据库使用的文件描述符超过了服务器中给定的范围。需要修改/etc/security/limits.d/hubble.conf中的文件。如果修好后还出现当前错误，请检查/etc/systemd/system/hubble.service 中的 LimitNOFILE=1000000的值

8.1.1.3 节点启动报 `bind: address already in use`

```
ports already in use ERROR: hubble server exited with error: consider changing the port via
↪ --listen-addr: listen tcp 127.0.0.1:15432: bind: address already in use
```

解决方案：节点服务器端口被占用，关闭被占用的端口进程或者重新指定`--listen-addr`的端口

8.1.1.4 节点加入到现有的 Hubble 集群 `Store directory already exists`

```
no resolvers found; use --join to specify a connected node

node belongs to cluster {"cluster hash"} but is attempting to connect to a
    ↪ gossip network for cluster {"another cluster hash"}
```

解决方案：启动节点时，选择存储数据的目录还包含标识数据来自哪个集群的元数据。当您已经启动了服务器上的一个节点，退出了 Hubble 进程，然后试图加入另一个集群时，这会导致冲突。因为现有目录的集群 ID 与新的集群 ID 不匹配，节点无法加入它。

解决办法: 删除现有 Hubble 数据库目录，重新启动。

8.2 sql 常见问题

本文档介绍 Hubble 中常见的 SQL 问题。

8.2.1 Hubble 是否支持 SELECT FOR UPDATE

用于并发控制的行级锁定`SELECT FOR UPDATE`

该`SELECT FOR UPDATE`语句用于通过控制对表的一行或多行的并发访问来对事务进行排序。

它通过锁定选择查询返回的行来工作，这样试图访问这些行的其他事务被迫等待锁定这些行的事务完成。这些其他事务根据尝试读取锁定行的值的时间有效地放入队列。

因为这种排队发生在读取操作期间，所以如果多个并发执行的事务尝试访问相同的数据，会阻止该选择的结果，Hubble 还可以防止可能发生的事务重试。

8.2.2 Hubble 支持的字符集类型

Hubble 字符集默认是 UTF8，目前只支持 UTF8。

8.2.3 Hubble 如何定位历史数据

使用`AS OF TIMESTAMP`语法读取历史数据。

8.2.4 sql 优化

请参考 sql 性能最佳实践。

8.2.5 多个查询同时进行，防止死锁

解决方案：需要将`CLUSTER SETTING sql.distsql.acquire_vec_fds.max_retries`设置为 0

```
set   CLUSTER SETTING sql.distsql.acquire_vec_fds.max_retries=0;
```

```
show  CLUSTER SETTING sql.distsql.acquire_vec_fds.max_retries;
```

```
  sql.distsql.acquire_vec_fds.max_retries
-----------------------------------------
                                        0
```

8.2.6 文件多次导入失败

导入大型文件可能遇到context canceled错误，或者重复重启导入很多次也无法完成，导致这个的原因是比较高的磁盘争用。解决方法，设置一个低于最大磁盘写入速度的值来缓解

```
SET CLUSTER SETTING kv.bulk_io_write.max_rate = '10MB';
```

8.2.7 Hubble 数据库支持 JSON 或 Protobuf 数据类型吗？

Hubble 数据库支持 JSONB 数据类型

创建表

```
create table users (
    profile_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    last_updated TIMESTAMP DEFAULT now(),
    user_profile JSONB
  );
```

```
insert info users (user_profile) values
    ('{"first_name": "Zhang", "last_name": "San", "location": "Beijing", "online
       ↪ " : true, "friends" : 520}'),
    ('{"first_name": "Wang", "status": "Looking for treats", "location" : "
       ↪ ShangHai"}');
```

具体请查看数据类型的 jsonb。

8.2.8 可以使用 Hubble 作为键值存储吗？

Hubble 是一个分布式 SQL 数据库，构建在事务性和强一致性的键值存储上。虽然无法直接访问键值存储，但可以使用由两列组成的简单表镜像直接访问，其中一列作为主键：

```
create table kv (k int PRIMARY KEY, v BYTES);
```

```
insert info kv values (1, b'hello');
```

这种 SQL 表方法还为您提供了定义良好的查询语言、已知的事务模型，以及在需要时向表中添加更多列的灵活性。

8.2.9 如何将数据批量插入 Hubble？

- 要将数据批量插入现有表中，在一个表中批处理多行插入语句。通过监视不同批大小 (100 行、1000 行、10000 行) 的性能，从测试上确定应用程序的最佳批大小。
- 要将数据批量插入到新表中import语句的性能优于insert。
- 也可以导出 csv 数据将数据导入到数据库中。

8.2.10 如何将最后一个 ID/SERIAL 值插入到表中？

Hubble 中没有返回最后插入值的函数。

使用RETURNING返回自动生成的值。

```
create table custs (id int default unique_rowid(), cust_name string,cust_age int
    ↪ );
```

```
insert into custs (cust_name,cust_age) values ('liudehua',63) returning id;
```

```
          id
----------------------
  911379594567778305
```

8.2.11 什么是事务争用？

当同时从多个客户端发出事务时，会发生事务争用对相同的数据进行操作。这可能导致事务相互等待，就像很多人在商店等同一个收银员结账一样。

8.3 数据库常见问题

8.3.1 选择 Hubble

8.3.1.1 什么是 Hubble 数据库？

Hubble 是一种分布式 SQL 建立在事务性和强一致性的键值存储基础上的数据库。它的磁盘、机器、机架甚至数据中心故障发生概率低，延迟中断情况的可能性小；支持强烈一致 ACID 交易；并提供熟悉的 SQL 语言用于构造、操作和查询数据的 API。

8.3.1.2 什么时候 Hubble 数据库是一个好的选择？

Hubble 数据库非常适合高可用和以毫秒响应时间的应用程序；无论其规模如何，它重在以最小的配置和开销实现数据自动复制、重新平衡和恢复。具体用例包括：

- 分布式部署
- 多数据中心部署
- 多区域部署

Hubble 数据库在 2ms 或更短时间内返回单行读取，在 4ms 或更少时间内返回单行写入。并支持 SQL 优化查询性能。

8.3.2 关于 Hubble 数据库

8.3.2.1 Hubble 数据库是如何扩展的？

Hubble 数据库水平扩展，开销最小。

在键值级别，Hubble 数据库从一个单一的空范围开始；当您输入数据时，这个单一范围最终会达到最大阈值。发生这种情况时，数据分为两个 ranges，每个覆盖整个键值空间的一个连续段。这一过程将无限期地继续下去；随着新数据的流入，现有 range 将继续拆分，以保持相对较小且一致的大小。

当集群跨越多个节点 (物理机、虚拟机或容器) 时，新分割的 range 会自动重新平衡为具有更多容量的节点。

HUBBLE

8.3.2.2 Hubble 数据库如何在发生故障后继续提供服务？

Hubble 数据库的设计从服务器重启到数据中心停机的软件和硬件故障后，能继续提供服务。使用强一致性复制以及故障后自动修复技术，可以在不混淆其他系统的情况下实现这一点。

复制

基于 Raft 一致性算法，Hubble 复制您的数据以确保可用性。根据您希望防止的故障的实际情况来定义副本的位置，可以采用多种方式。

在跨多个地理区域的数据库集群中，区域之间的往返延迟将直接影响数据库的性能。在这种情况下，重要的是要考虑每个表的延迟要求，然后使用适当的数据拓扑定位数据以实现最佳性能和恢复能力。

自动修复

对于短期故障，例如服务器重启，只要大多数副本仍然可用，Hubble 就能保证服务正常运行。Raft 确保在前负责人失败时为每个副本组选出一个新的负责人，以便事务可以继续进行，受影响的副本可以在重新联机后重新加入其组。对于较长期的故障，例如服务器长时间停机或数据中心停机，Hubble 会使用未受影响的副本作为源，自动从丢失的节点重新平衡副本。使用所有可用节点以及群集的聚合磁盘和网络带宽以分布式方式重新复制丢失的副本。

8.3.2.3 Hubble 数据库的一致性如何？

Hubble 确保可序列化 SQL 事务标准定义了最高隔离级别。它通过将写操作的 Raft 一致性算法和读操作的自定义基于时间的同步算法相结合来实现。

- 存储的数据使用多并发进行版本控制，因此读取只是将其范围限制为读取事务启动时可见的数据。
- 一致性算法可确保所有大多数副本在是否成功提交更新方面始终保持一致。更新或者插入必须到达大多数副本确认，才能被视为已提交。

为了确保写入事务不会干扰其后开始的读取事务，Hubble 还使用时间戳缓存技术，它会记住正在进行的事务读取数据的时间。

8.3.2.4 Hubble 数据库如何既具有高可用性又具有强一致性？

这个 CAP 定理声明分布式系统不可能同时提供以下三种保证中的两种以上：

- 一致性
- 可利用性
- 分区容差

Hubble 是一个 CP(一致和分区容忍) 系统。这意味着这样，在有分区的情况下，系统将变得不可用。例如，写操作需要大多数副本的确认，而读操作需要租约，只有在可以写操作时才能将租约传输到其他节点。

Hubble 数据库也是高度可用的，尽管这里的可用与 CAP 定理中的使用方式不同。在 CAP 定理中，可用性是一个二进制属性，但对于高可用性，我们将可用性称为频谱 (对于 99.999% 可用的系统，使用'五个九'之类的术语)。

既是 CP 又是 HA，意味着只要大多数副本可以相互通信，它们就应该能够取得进展。例如，如果您将 Hubble 部署到三个数据中心，而其中一个数据中心的网络链接出现故障，那么其他两个数据中心应该能够正常运行，且只有短暂的中断时间。

8.3.2.5 Hubble 支持分布式事务吗？

支持。Hubble 数据库在集群中分发事务，无论是单个位置的几台服务器，还是多个数据中心的多台服务器。与分片设置不同，您不需要知道数据的精确位置；您只需与集群中的任何节点进行对话，Hubble 就会将您的事务

无缝地发送到正确的位置。在进行重新平衡时，分布式事务可以在没有停机或额外延迟的情况下继续进行。

8.3.2.6 为什么要用 Hubble 数据库 sql？

在最底层，Hubble 数据库是一个分布式、强一致性、事务性键值存储，外部 API 是带有扩展的标准 SQL。这为开发人员提供了熟悉的关系概念，如模式、表、列和索引，以及使用成熟，有过长时间验证，操作性简单、查询能力突出。此外，由于 Hubble 支持 PostgreSQL 连线协议，所以让应用程序与数据库对话很简单。

8.3.2.7 Hubble 中的事务是否保证 ACID 语义？

支持，每次事务确保确保 ACID 语义跨越任意表和行。

Hubble 支持将多个 SQL 语句在单个事务中提交。每个事务都能保证在集群中跨多表的 ACID 特性。当一个事务成功，则所有都确保其成功。如果事务的任何部分失败，整个事务将中止，数据库将保持不变。Hubble 保证当一个事务处于挂起状态时，它通过`serializable isolation`事务级别，将其与其他他并发事务中隔离出来。

- 原子性：Hubble 中的事务是全有或全无的。如果事务的任何部分失败，整个事务将中止，数据库保持不变。如果一个事务成功，所有操作要么全部成功。
- 一致性：SQL 操作永远不会看到任何中间状态，操作总是看到以前完成的语句对重叠数据的结果，总是使数据库从一个一致性状态变换到另一个一致性状态。
- 隔离性：Hubble 中的事务实现了最强的 ANSI 隔离级别：可序列化 (可串行)，这意味着交易永远不会导致异常。
- 持久性：通过 Raft 一致性算法。电源或磁盘故障只影响少数副本，但不会阻止群集运行，也不会丢失任何数据。

8.3.2.8 Hubble 数据库需要原子钟来同步时间吗？

不需要。Hubble 数据库的设计没有原子钟。Hubble 是一个在任意节点集合上运行的数据库，然而 Hubble 数据库确实需要适度的时钟同步才能保证正确性。如果时钟漂移超过最大阈值，节点将脱机。

8.3.2.9 我可以使用哪些语言来连接数据库？

Hubble 支持 PostgreSQL 连线协议，因此您可以使用任何可用的 PostgreSQL 客户端驱动程序。汇总语言如下：

- `JavaScript/TypeScript`
- `Python`
- `Go`
- `Java`
- `Ruby`
- `C`
- `C#(.NET)`
- `Rust`

8.3.2.10 为什么 Hubble 使用 PostgreSQL 协议而不是 MySQL 协议？

Hubble 数据库使用 PostgreSQL 有线协议，因为它比 MySQL 协议有更好的文档记录，并且因为 PostgreSQL 有一个自由的开源许可证，类似于 BSD 或 MIT 许可证，而 MySQL 有更严格的 GNU 许可证。

8.3.2.11 Hubble 数据库的安全模型是什么？

如果安全的时候，可以对客户端节点和节点间通信进行加密，并且 SSL 证书对客户端和节点的身份进行身份验证。当不安全时，没有加密或身份验证。

此外，Hubble 数据库支持数据库和表上的通用 SQL 权限。这个根用户拥有所有数据库的权限，而唯一用户可以被授予数据库和表级别特定语句的权限。

8.3.3 数据库的比较

8.3.3.1 Hubble 与 MySQL 或 PostgreSQL 相比如何？

虽然所有这些数据库都支持 SQL 语法，但 Hubble 是唯一可以轻松扩展 (无需手动复杂的切分)、自动重新平衡和修复以及跨集群分发事务的数据库。

8.3.3.2 Hubble 数据库与 HBase、MongoDB 相比如何？

虽然所有这些都是分布式数据库，但只有 Hubble 支持分布式事务并提供了强大的一致性。此外，这些其他数据库还提供自定义 API，而 Hubble 提供带扩展的标准 SQL。

9 版本发布

9.1 数据库版本

9.1.1 v3.17

9.1.1.1 BUG 修复

- 修复了在向表中添加新列 (或更改列类型) 时会长时间持有锁的错误。
- 修复了对`changefeed.memory.per_changefeed_limit`参数调整，现在通过集群设置，会减少对前台延迟的影响。
- 修复了`LEFT JOIN`在对`not null`约束的虚拟计算列的表进行操作时导致错误的结果。
- 在查询溢出的情况下，Hubble 现在可以自动取消查询这项任务。
- 修复了`EXPLAIN`在某些复杂场景下运行语句时索引推荐不佳的情况。
- 修复了`RESTORE`在恢复的数据时作业会挂起的情况。

9.1.1.2 性能

- 优化了数据库中有大量表以及集群中有大量 range 时页面加载性能。
- 改进了优化器对涉及许多受约束列的谓词的基数估计，优化索引命中。
- 提高每个节点稳定性。
- 提升了群集恢复的稳定性。
- 减少了有大量列或索引的表进行查询的计划时间。
- 提高了大数据量复杂查询时内存利用率。
- 提高了`IMPORT`任务的导入速度。
- 提升了`ORDER BY` 和 `LIMIT` 查询的语句性能。
- 提升了`DECIMAL`数据类型的大型数据集的算术和聚合速度。
- 相同的数据量备份相比以前现在使用更少的内存。

9.1.1.3 函数

- host 将主机地址类型抽出为文本

```
select host('192.168.1.5/24');
```

```
     host
---------------
  192.168.1.5
```

- exp 自然指数

```
select  exp(1.0);
```

```
        exp
---------------------
2.7182818284590452354
```

- trim 从字串 string 的开头/结尾/两边/ 删除包含 characters 的字串

```
select trim(both 'x' from 'xTomxx')
```

```
 trim
--------
  Tom
```

- sign(x) x 为负，零，正时返回结果依次为：-1,0,1

```
select sign(20);
```

```
sign
------
 1
```

- broadcast 网络广播地址

```
select broadcast('192.168.1.5/24');
```

```
     broadcast
--------------------
  192.168.1.255/24
```

- masklen 抽取网络掩码长度

```
select masklen('192.168.1.5/24');
```

```
  masklen
-----------
    24
```

- netmask 为网络构造网络掩码

```
select netmask('192.168.1.5/24');
```

```
     netmask
-----------------
  255.255.255.0
```

- hostmask 为网络构造主机掩码

```
select hostmask('192.168.23.20/30');
```

HUBBLE

```
  hostmask
------------
  0.0.0.3
```

9.1.1.4 sql 变化

- 添加对在架构中显示默认权限的支持，支持SHOW DEFAULT PRIVILEGES子句

```
SHOW DEFAULT PRIVILEGES IN SCHEMA S;
```

```
role       for_all_roles  object_type  grantee    privilege_type
testuser   false          tables       testuser2  DROP
testuser   false          tables       testuser2  SELECT
testuser   false          tables       testuser2  UPDATE
```

- 添加对SHOW SUPER REGIONS FROM DATABASE语句

```
SHOW SUPER REGIONS FROM DATABASE ABC;
```

```
ABC  ca-central-sr  {ca-central-1}
ABC  test           {ap-southeast-2,us-east-1}
```

- 增加用户应该依赖默认权限

```
ALTER DEFAULT PRIVILEGES GRANT ALL ON TABLES TO foo;
```

- 创建数据库能指定所有者

```
CREATE DATABASE dbname WITH OWNER S;
```

- SHOW GRANTS ON table/type/schema...语句多了is_grantable列

```
SHOW GRANTS ON DATABASE dbname;
```

```
database_name   grantee privilege_type     is_grantable
dbname            admin            ALL             true
dbname            public       CONNECT            false
dbname            root             ALL             true
dbname             s               ALL             true
```

- 添加了RESET语句，它会将会话变量的值重置为其默认值。

```
RESET session_name;
```

- 添加了会话变量default_transaction_quality_of_service，该变量代表会话中提交的 SQL 请求控制增加后续 SQL 请求的准入控制优先级

```
SET default_transaction_quality_of_service=critical;
```

要降低后续 SQL 请求的准入控制优先级：

```
SET default_transaction_quality_of_service=background;
```

要将准入控制优先级重置为默认会话设置（介于背景和关键之间）

```
SET default_transaction_quality_of_service=regular;
```

- 现在可以在同一事务中交换名称（表等）。例如：

```
CREATE TABLE foo();
BEGIN;
ALTER TABLE foo RENAME TO bar;
CREATE TABLE foo();
COMMIT;
```

- 添加了一个内置函数，返回一个无序的整数

```
select unordered_unique_rowid();
```

```
1554671853219610625
```

- 默认情况下不允许对序列进行跨数据库引用。这可以通过集群设置sql.cross_db_sequence_references
  ↪ .enabled启用。

```
set sql.cross_db_sequence_references.enabled=dbname;
```

- 在展示约束的时候增加了with comment子句;

```
SHOW CONSTRAINTS from t with comment;
```

- 当表中只有主键约束的时候，show create table不在展示famlily项

```
show create table t;
```

```
CREATE TABLE public.t (
    a INT8 NOT NULL,
    CONSTRAINT t_pkey PRIMARY KEY (a ASC)
)
```

©2022 天云融创数据科技 (北京) 有限公司保留所有权利

地址：北京市朝阳区金田公园内 8 号-22 栋

网址：http://www.beagledata.com

邮箱：websales@beagledata.com

电话：18610182713