HUBBLE 产品手册 V3.7

Hubble 团队

2024-12-13

目录

1 部署 **17**

1.1 软硬件环境配置 . . . 17
1.1.1 Linux 操作系统版本要求 . . . 17
1.1.2 开发和测试环境-推荐配置 . . . 17
1.1.3 生产环境-推荐配置 . . . 17
1.1.4 网络端口要求 . . . 18
1.1.5 客户端 Web 浏览器要求 . . . 18
1.2 安装前服务配置 . . . 18
1.2.1 系统级别配置 . . . 18
1.2.1.1 关闭交换分区 . . . 18
1.2.1.2 关闭服务器的防火墙 . . . 19
1.2.1.3 关闭透明大页 . . . 19
1.2.1.4 禁用 selinux . . . 19
1.2.1.5 sysctl 配置 . . . 20
1.2.1.6 ulimit 配置 . . . 21
1.2.1.7 主机与 IP 配置 . . . 21
1.2.1.8 hubble 账户与组配置 . . . 22
1.2.1.9 时间同步 . . . 22
1.2.1.10 NFS 配置 (如果有网络共享盘跳过次步) . . . 22
1.2.2 服务器之间配置 . . . 24
1.2.2.1 主机之间主机名同步 . . . 24
1.2.2.2 hubble 数据库用户免密配置 . . . 24
1.3 hubble 集群安装 . . . 25
1.3.1 安装 hubble . . . 25
1.3.2 创建 hubble 所需目录 . . . 25
1.3.3 hubble 数据库 license 注册激活 . . . 25
1.3.4 添加 hubble 数据库配置文件 . . . 26
1.3.5 添加普通用户启停特权 . . . 27
1.3.6 证书配置 . . . 27
1.3.7 启动服务 . . . 28
1.3.8 初始化服务 . . . 29
1.3.9 连接数据库，创建用户 . . . 29
1.3.10 测试验证 . . . 29
1.3.11 设置开机自启动 . . . 29
1.4 hubble 单机安装 . . . 29

HUBBLE

1.4.1 安装 hubble . . . 30
1.4.2 创建 hubble 所需目录 . . . 30
1.4.3 hubble 数据库 license 注册激活 . . . 30
1.4.4 添加 hubble 数据库配置文件配置 _1 . . . 30
1.4.5 添加 hubble 数据库配置文件配置 _2 . . . 31
1.4.6 添加 hubble 数据库配置文件配置 _3 . . . 32
1.4.7 添加 hubble 用户启停特权 . . . 33
1.4.8 证书配置 . . . 33
1.4.9 启动服务 . . . 34
1.4.10 初始化服务 . . . 34
1.4.11 连接数据库，创建用户 . . . 34
1.4.12 测试验证 . . . 35
1.4.13 设置开机自启动 . . . 35

2 SQL 35

2.1 数据类型 . . . 35
2.1.1 支持的类型 . . . 35
2.1.2 数据类型转换 . . . 36
2.1.3 ARRAY . . . 36
2.1.3.1 语法 . . . 36
2.1.3.2 大小 . . . 37
2.1.3.3 示例 . . . 37
2.1.3.4 支持的转换 . . . 39
2.1.4 BIT . . . 39
2.1.4.1 别名 . . . 39
2.1.4.2 语法 . . . 40
2.1.4.3 大小 . . . 40
2.1.4.4 示例 . . . 40
2.1.4.5 支持的转换 . . . 41
2.1.5 BOOL . . . 41
2.1.5.1 别名 . . . 41
2.1.5.2 语法 . . . 41
2.1.5.3 大小 . . . 41
2.1.5.4 示例 . . . 42
2.1.5.5 支持的转换 . . . 42
2.1.6 BYTES . . . 42
2.1.6.1 别名 . . . 42
2.1.6.2 语法 . . . 43
2.1.6.3 大小 . . . 43
2.1.6.4 示例 . . . 43
2.1.6.5 支持的转换 . . . 43
2.1.7 COLLATE . . . 44
2.1.7.1 细节 . . . 44
2.1.7.2 支持的排序 . . . 45
2.1.7.3 SQL 句法 . . . 45
2.1.7.4 示例 . . . 45
2.1.8 DATE . . . 47

HUBBLE

2.1.8.1 语法 . . . 47
2.1.8.2 大小 . . . 47
2.1.8.3 示例 . . . 47
2.1.8.4 支持的转换 . . . 48
2.1.9 DECIMAL . . . 48
2.1.9.1 别名 . . . 48
2.1.9.2 精度和刻度 . . . 48
2.1.9.3 语法 . . . 48
2.1.9.4 大小 . . . 49
2.1.9.5 示例 . . . 49
2.1.9.6 支持的转换 . . . 50
2.1.10 FLOAT . . . 50
2.1.10.1 别名 . . . 50
2.1.10.2 语法 . . . 50
2.1.10.3 大小 . . . 50
2.1.10.4 示例 . . . 50
2.1.10.5 支持的转换 . . . 51
2.1.11 INET . . . 51
2.1.11.1 语法 . . . 51
2.1.11.2 大小 . . . 52
2.1.11.3 示例 . . . 52
2.1.11.4 支持的转换 . . . 53
2.1.12 INT . . . 53
2.1.12.1 名称和别名 . . . 53
2.1.12.2 语法 . . . 53
2.1.12.3 大小 . . . 53
2.1.12.4 示例 . . . 54
2.1.12.5 支持的转换 . . . 55
2.1.13 INTERVAL . . . 56
2.1.13.1 别名 . . . 56
2.1.13.2 语法 . . . 56
2.1.13.3 大小 . . . 56
2.1.13.4 示例 . . . 56
2.1.13.5 支持的转换 . . . 57
2.1.14 JSONB . . . 57
2.1.14.1 别名 . . . 57
2.1.14.2 注意事项 . . . 58
2.1.14.3 语法 . . . 58
2.1.14.4 大小 . . . 58
2.1.14.5 JSONB 函数 . . . 58
2.1.14.6 JSONB 操作符 . . . 59
2.1.14.7 示例 . . . 59
2.1.14.8 支持的转换 . . . 62
2.1.15 SERIAL . . . 62
2.1.15.1 操作方式 . . . 62
2.1.15.2 示例 . . . 63
2.1.16 STRING . . . 65

HUBBLE

2.1.16.1 别名 . . . 65
2.1.16.2 长度 . . . 66
2.1.16.3 语法 . . . 66
2.1.16.4 大小 . . . 66
2.1.16.5 示例 . . . 66
2.1.16.6 支持的转换 . . . 67
2.1.17 TIME . . . 68
2.1.17.1 别名 . . . 69
2.1.17.2 语法 . . . 69
2.1.17.3 大小 . . . 69
2.1.17.4 示例 . . . 69
2.1.17.5 支持的转换 . . . 70
2.1.18 TIMESTAMP / TIMESTAMPTZ . . . 70
2.1.18.1 变体 . . . 70
2.1.18.2 最佳实践 . . . 70
2.1.18.3 别名 . . . 70
2.1.18.4 语法 . . . 70
2.1.18.5 大小 . . . 71
2.1.18.6 示例 . . . 71
2.1.18.7 支持的转换 . . . 71
2.1.19 UUID . . . 72
2.1.19.1 句法 . . . 72
2.1.19.2 大小 . . . 72
2.1.19.3 示例 . . . 72
2.1.19.4 支持的转换 . . . 75
2.2 函数 . . . 75
2.2.1 特殊的语法形式 . . . 75
2.2.2 条件运算符和类函数运算符 . . . 76
2.2.3 数组函数 . . . 78
2.2.4 比较函数 . . . 87
2.2.5 时间函数 . . . 88
2.2.6 ID 函数 . . . 91
2.2.7 数学类函数 . . . 92
2.2.8 字符串函数 . . . 93
2.2.9 聚合函数 . . . 95
2.2.10 JSON 函数 . . . 102
2.3 数据库信息 . . . 103
2.3.1 包含的数据信息 . . . 103
2.3.2 administrable_role_authorizations . . . 103
2.3.3 applicable_roles . . . 104
2.3.4 check_constraints . . . 104
2.3.5 columns . . . 104
2.3.6 column_privileges . . . 105
2.3.7 constraint_column_usage . . . 105
2.3.8 enabled_roles . . . 105
2.3.9 key_column_usage . . . 105
2.3.10 referential_constraints . . . 106

HUBBLE

2.3.11 role_table_grants . . . 106
2.3.12 schema_privileges . . . 106
2.3.13 schemata . . . 107
2.3.14 sequences . . . 107
2.3.15 statistics . . . 107
2.3.16 table_constraints . . . 108
2.3.17 table_privileges . . . 108
2.3.18 tables . . . 108
2.3.19 user_privileges . . . 109
2.3.20 views . . . 109
2.4 事务 . . . 109
2.4.1 事务隔离级别 . . . 109
2.4.2 事务开启、提交与回滚 . . . 110
2.4.3 事务操作 . . . 110
2.4.4 事务优先级 . . . 110
2.5 约束 . . . 110
2.5.1 支持的约束 . . . 110
2.5.2 约束条件的使用 . . . 110
2.5.2.1 添加约束条件 . . . 110
2.5.3 创建主键时定义约束的名称 . . . 111
2.5.4 列出约束条件 . . . 111
2.5.5 移除约束条件 . . . 111
2.5.6 改变约束条件 . . . 111
2.6 视图 . . . 112
2.6.1 创建视图 . . . 112
2.6.2 列表视图 . . . 112
2.6.3 查询视图 . . . 113
2.6.4 重命名视图 . . . 113
2.6.5 删除视图 . . . 113
2.7 SQL 语法手册 . . . 113
2.7.1 DCL . . . 113
2.7.1.1 USER . . . 113
2.7.1.2 ROLE . . . 115
2.7.1.3 PRIVILEGE . . . 115
2.7.2 DDL . . . 118
2.7.2.1 ADD COLUMN . . . 118
2.7.2.2 ADD CONSTRAINT . . . 122
2.7.2.3 CREATE DATABASE . . . 124
2.7.2.4 CREATE TABLE . . . 124
2.7.2.5 CREATE INDEX . . . 127
2.7.2.6 CREATE VIEW . . . 128
2.7.2.7 CREATE SEQUENCE . . . 129
2.7.2.8 COMMENT ON . . . 130
2.7.2.9 DROP DATABASE . . . 131
2.7.2.10 DROP TABLE . . . 131
2.7.2.11 DROP COLUMN . . . 132
2.7.2.12 DROP INDEX . . . 133

HUBBLE

2.7.2.13 DROP VIEW . . . 133
2.7.2.14 DROP SEQUENCE . . . 134
2.7.2.15 DROP CONSTRAINT . . . 134
2.7.2.16 RENAME DATABASE . . . 135
2.7.2.17 RENAME TABLE . . . 135
2.7.2.18 RENAME COLUMN . . . 136
2.7.2.19 RENAME SEQUENCE . . . 136
2.7.2.20 RENAME CONSTRAINT . . . 136
2.7.2.21 SHOW DATABASES . . . 137
2.7.2.22 SHOW TABLES . . . 137
2.7.2.23 SHOE COLUMNS . . . 137
2.7.2.24 SHOW INDEX . . . 137
2.7.2.25 SHOW SCHEMAS . . . 137
2.7.2.26 SHOW SEQUENCE . . . 137
2.7.2.27 SHOW CONSTRAINT . . . 137
2.7.2.28 SHOW PARTITIONS . . . 138
2.7.2.29 SHOW RANGES . . . 138
2.7.2.30 SHOW CREATE TABLE . . . 138
2.7.2.31 SHOW CREATE VIEW . . . 138
2.7.2.32 SHOW CREATE SEQUENCE . . . 138
2.7.3 DML . . . 138
2.7.3.1 SELECT . . . 138
2.7.3.2 INSERT . . . 141
2.7.3.3 DELETE . . . 143
2.7.3.4 UPDATE . . . 143
2.7.3.5 TRUNCATE . . . 144
2.7.3.6 UPSERT . . . 145
2.7.4 TRANSACTION . . . 146
2.7.5 SYSTEM . . . 148
2.7.5.1 SHOW SESSIONS . . . 148
2.7.5.2 SHOW QUERIES . . . 149
2.7.5.3 CANCEL QUERY . . . 151
2.7.5.4 SHOW ALL . . . 151
2.7.5.5 SET . . . 151
2.7.5.6 SHOW CLUSTER SETTING . . . 152
2.7.5.7 SHOW CLUSTER SETTING . . . 152
2.7.5.8 EXPERIMENTAL_AUDIT . . . 153
2.7.6 JOBS . . . 153
2.7.6.1 EXPLAIN . . . 153
2.7.6.2 EXPORT . . . 153
2.7.6.3 IMPORT . . . 154
2.7.6.4 IMPORT INTO . . . 155
2.7.6.5 SHOW JOBS . . . 157
2.7.6.6 CANCEL JOB . . . 157
2.7.6.7 PAUSE JOB . . . 158
2.7.6.8 RESUME JOB . . . 158
2.8 窗口函数 . . . 158

HUBBLE

2.8.1 窗口的定义 . . . 158
2.8.2 窗口函数机制 . . . 159
2.8.3 提示说明: . . . 160
2.9 主键 . . . 161
2.9.1 注意事项 . . . 161
2.9.2 定义 . . . 161
2.9.3 目的 . . . 161
2.9.4 示例 . . . 161

3 运维 162
3.1 数据备份 . . . 162
3.1.1 备份语法 . . . 162
3.1.1.1 参数说明 . . . 163
3.1.2 备份数据路径 destination 说明 . . . 163
3.1.3 全量备份示例 . . . 163
3.1.4 增量备份 . . . 163
3.1.5 指定时间戳备份 . . . 164
3.2 数据恢复 . . . 164
3.2.1 依赖对象 . . . 164
3.2.2 备份数据路径 . . . 164
3.2.3 参数说明 . . . 165
3.2.4 示例 . . . 165
3.2.4.1 恢复单个表 . . . 165
3.2.4.2 恢复多个表 . . . 165
3.2.4.3 恢复整个数据库 . . . 165
3.2.4.4 指定恢复时间点 . . . 165
3.2.4.5 从增量备份恢复数据 . . . 165
3.2.4.6 恢复到另一个数据库 . . . 166
3.2.4.7 在恢复之前删除外键 . . . 166
3.2.4.8 从系统中恢复用户。用户备份 . . . 166
3.3 监控告警 . . . 166
3.3.1 管理员界面区域 . . . 166
3.3.2 管理员界面访问 . . . 167
3.3.3 集群概况页面 . . . 167
3.3.3.1 集群概况面板 . . . 167
3.3.3.2 节点列表 . . . 168
3.3.3.3 LIVE 节点列表 . . . 168
3.3.4 汇总面板 . . . 168
3.3.5 事件面板 . . . 169
3.3.6 监控-概况仪表板 . . . 169
3.3.6.1 SQL 查询数 . . . 169
3.3.7 运行延迟：SQL，99% . . . 171
3.3.7.1 每节点的副本数 . . . 171
3.3.7.2 容量 . . . 172
3.3.8 硬件概览 . . . 172
3.3.8.1 CPU 使用率 . . . 173
3.3.8.2 内存使用情况 . . . 173

HUBBLE

3.3.8.3 磁盘读取字节 . . . 173
3.3.8.4 磁盘写入字节 . . . 174
3.3.8.5 磁盘读取操作 . . . 174
3.3.8.6 磁盘写操作 . . . 174
3.3.8.7 磁盘 IOPS 进行中 . . . 175
3.3.8.8 可用磁盘容量 . . . 175
3.3.8.9 收到的网络字节 . . . 176
3.3.8.10 发送的网络字节 . . . 176
3.3.9 运行时仪表板 . . . 176
3.3.9.1 活动节点数 . . . 176
3.3.9.2 内存使用情况 . . . 178
3.3.9.3 并发数 . . . 178
3.3.9.4 GC 运行数 . . . 178
3.3.9.5 GC 暂停时间 . . . 178
3.3.9.6 CPU 时间 . . . 178
3.3.9.7 时钟偏移 . . . 180
3.3.10 SQL 仪表板 . . . 180
3.3.10.1 SQL 连接数 . . . 180
3.3.10.2 SQL 字节流量 . . . 180
3.3.10.3 SQL 查询数 . . . 181
3.3.10.4 SQL 查询失败数 . . . 181
3.3.10.5 事务数 . . . 182
3.3.10.6 其他图表 . . . 182
3.3.11 存储仪表板 . . . 182
3.3.11.1 容量 . . . 182
3.3.11.2 文件描述符 . . . 184
3.3.11.3 其他图像 . . . 184
3.3.12 复制仪表板 . . . 184
3.3.12.1 分片 . . . 184
3.3.12.2 副本静止数 . . . 186
3.3.12.3 快照数 . . . 186
3.3.12.4 其他图形 . . . 187
3.3.13 数据变动仪表板 . . . 187
3.3.13.1 最大数据变动延迟 . . . 187
3.3.13.2 接收字节流量 . . . 187
3.3.13.3 接收数 . . . 188
3.3.13.4 接收计时 . . . 188
3.3.14 数据库页面 git fetch upstream . . . 188
3.3.14.1 表视图 . . . 188
3.3.14.2 GRANTS 视图 . . . 189
3.3.15 任务页面 . . . 189
3.3.15.1 任务细节 . . . 189
3.3.15.2 过滤结果 . . . 189
3.3.16 语句页面 . . . 189
3.3.16.1 有效期 . . . 189
3.3.16.2 按应用过滤 . . . 189
3.3.16.3 参数 . . . 189

HUBBLE

3.3.17 语句明细页面 . . . 193
3.3.17.1 逻辑计划 . . . 193
3.3.17.2 阶段 . . . 193
3.3.17.3 网关节点统计 . . . 193
3.3.17.4 执行计数 . . . 194
3.3.17.5 受影响行 . . . 194
3.3.17.6 统计信息 . . . 194
3.4 集群扩容 . . . 194
3.4.1 新节点准备 . . . 194
3.4.2 目录准备 . . . 198
3.4.3 证书配置 . . . 198
3.4.4 添加 hubble 数据库配置文件 . . . 199
3.4.5 添加普通用户启停特权 . . . 199
3.4.6 启动服务 . . . 200
3.4.7 测试验证 . . . 200
3.4.8 设置开机自启动 . . . 200
3.5 集群缩容 . . . 200
3.5.1 删除单个活动节点 (安全模式) . . . 200
3.5.1.1 前提条件 . . . 200
3.5.1.2 步骤 1. 停用之前检查节点 . . . 201
3.5.1.3 步骤 2. 退役并删除节点 . . . 201
3.5.1.4 步骤 3. 停用后检查节点和群集 . . . 201
3.5.2 安全删除单个已停止节点 . . . 202
3.5.2.1 步骤 1. 识别已停止节点的 ID . . . 202
3.5.2.2 步骤 2. 将失效节点标记为已停用 . . . 202
3.5.3 删除多个节点 . . . 203
3.5.3.1 开始之前 . . . 203
3.5.3.2 步骤 1. 确定要停用的节点的 ID . . . 203
3.5.3.3 步骤 2. 停用节点 . . . 203
3.5.3.4 步骤 4. 停用后检查节点和群集 . . . 204
3.5.3.5 步骤 5. 删除退役的节点 . . . 204
3.5.4 重新调试节点（安全模式） . . . 205
3.5.4.1 步骤 1. 识别退役节点的 ID . . . 205
3.5.4.2 步骤 2. 重新调试节点 . . . 205
3.5.4.3 步骤 3. 重新启动重新使用的节点 . . . 205
3.5.5 总结 . . . 205
3.6 常见问题 . . . 206
3.6.1 导入问题 . . . 206
3.6.2 副本数 . . . 206
3.6.3 错误日志 . . . 206
3.7 日常巡检 . . . 206
3.7.1 巡检意义 . . . 206
3.7.2 巡检范围 . . . 206
3.7.3 登录的网址 . . . 206
3.7.4 页面服务 . . . 207
3.7.4.1 概况 . . . 207
3.7.4.2 监控 . . . 207

HUBBLE

3.7.4.3 语句 . . . 207
3.7.4.4 任务 . . . 208
3.7.5 异常处理检测 . . . 208
3.7.5.1 节点实例查看 . . . 208
3.7.5.2 运行任务处理 . . . 208
3.7.5.3 运行 sql 语句处理 . . . 209
3.7.6 服务启停 . . . 209
3.7.7 服务器端 . . . 209
3.8 紧急预案 . . . 210
3.8.1 节点重启 (注意使用 hubble 账户操作) . . . 210
3.8.2 节点可疑/宕机 . . . 210
3.8.2.1 语句 . . . 210
3.8.2.2 任务 . . . 212
3.8.2.3 终结办法对可疑/宕机的节点进行重启操作 . . . 212
3.8.3 服务器节点故障 . . . 213
3.9 时区调整 . . . 213
3.9.1 JAVA 中设置时区 . . . 213
3.9.2 在 Dbeaver 设置时区 . . . 214
3.9.3 SQL 中设置时区 . . . 214

4 安全 215
4.1 审计日志 . . . 215
4.1.1 所需权限 . . . 215
4.1.2 参数 . . . 215
4.1.3 审计日志文件存储位置 . . . 215
4.1.4 开启审计日志 . . . 215
4.1.5 关闭审计日志 . . . 216
4.2 更新安全证书 . . . 216
4.2.1 说明 . . . 216
4.2.2 何时轮换证书 . . . 216
4.2.3 操作步骤 . . . 216
4.2.3.1 查看证书信息 . . . 216
4.2.3.2 备份证书文件 . . . 217
4.2.3.3 关闭服务 . . . 217
4.2.3.4 数据库 license 注册激活 . . . 217
4.2.3.5 证书更新 . . . 218
4.2.3.6 重启服务 . . . 219
4.2.3.7 连接测试 . . . 219

5 优化 219
5.1 索引 . . . 219
5.1.1 索引工作原理 . . . 219
5.1.1.1 索引创建 . . . 219
5.1.1.2 索引选择 . . . 220
5.1.1.3 索引存储 . . . 220
5.1.1.4 性能 . . . 220
5.1.2 最佳实践 . . . 220

HUBBLE

5.1.2.1 索引列 . . . 220
5.1.2.2 存储字段 . . . 220
5.1.2.3 示例 . . . 220
5.2 向量引擎 . . . 223
5.2.1 配置方式 . . . 223
5.2.2 支持的数据类型 . . . 223
5.2.3 示例 . . . 223
5.3 开发使用规范 . . . 224
5.3.1 开发设计 . . . 224
5.3.1.1 命名规范 . . . 224
5.3.1.2 处理特定的错误码 `40001` . . . 224
5.3.1.3 多节点进行写入 . . . 224
5.3.1.4 查询可适用任何节点 . . . 224
5.3.1.5 尽可能确保表有主键 . . . 224
5.3.1.6 索引设计 . . . 224
5.3.1.7 尽量避免大表后建索引的情况 . . . 225
5.3.1.8 连接数的合理规划 . . . 225
5.3.1.9 建表复制数据 `CREATE TABLE AS SELECT` . . . 225
5.3.2 SQL 规范建议 . . . 225
5.3.2.1 使用 UUID 生成唯一主键 . . . 225
5.3.2.2 插入同时返回数据 . . . 225
5.3.2.3 批量写入速度优化 . . . 226
5.3.2.4 使用多行的 DML . . . 226
5.3.2.5 执行计划分析 . . . 226
5.3.2.6 大数据量的分析如何提升性能 . . . 226
5.3.2.7 列族使用 . . . 226
5.3.2.8 优雅的清空表 . . . 227
5.3.2.9 避免使用 CTAS 与 SFU（select for update） . . . 227
5.3.2.10 合理的使用视图 . . . 227
5.3.2.11 删除大量数据 . . . 227
5.3.2.12 删除对业务没有帮助的索引 . . . 227
5.3.3 部署规范 . . . 227
5.3.3.1 交换分区 . . . 227

6 导入 228
6.1 csv 导入 . . . 228
6.1.1 步骤一 CSV 格式数据准备 . . . 228
6.1.2 步骤二：将数据文件放置在集群可访问到的位置 . . . 228
6.1.3 步骤三：将数据迁移到 Hubble 数据库中 . . . 228
6.1.3.1 hubble 库中不存在的表的数据迁移 . . . 228
6.1.3.2 hubble 库中已存在表的数据迁移 . . . 229
6.1.3.3 多文件导入 . . . 229
6.1.4 配置选项 . . . 229
6.1.4.1 列定界符 . . . 229
6.1.4.2 跳过注释行 . . . 230
6.1.4.3 跳过标题行 . . . 230
6.1.4.4 空字符串 . . . 231

HUBBLE

6.1.4.5 文件压缩格式 . . . 231
6.2 MySQL 导入 . . . 231
6.2.1 MySQL 单表迁移至 Hubble . . . 231
6.2.1.1 步骤一：导出需要迁移的表 . . . 231
6.2.1.2 步骤二：将导出的数据文件放置于集群可访问到的位置 . . . 231
6.2.1.3 步骤三：导入 MySQL 导出的文件 . . . 232
6.2.2 MySQL 整库迁移至 Hubble . . . 232
6.2.2.1 步骤一：导出需要迁移的整库 . . . 232
6.2.2.2 步骤二：将导出的数据文件放置于集群可访问到的位置 . . . 232
6.2.2.3 步骤三：导入 MySQL 导出的整库数据 . . . 233
6.3 Oracle 导入 . . . 233
6.3.1 步骤 1: 导出 Oracle 模式 . . . 233
6.3.2 步骤 2: 将 Oracle 模式转换为 SQL . . . 233
6.3.3 步骤 3: 导出表数据 . . . 233
6.3.4 步骤 4: 配置表数据并将其转换为 CSV . . . 234
6.3.5 步骤 5:Oracle 映射到 hubble 数据类型 . . . 235
6.3.5.1 数据类型映射 . . . 235
6.3.6 步骤 6: 数据导入 . . . 236
6.4 Postgesql 导入 . . . 236
6.4.1 单表迁移 . . . 236
6.4.1.1 步骤一：导出需要迁移的表 . . . 236
6.4.1.2 步骤二：将导出的数据文件放置于集群可访问到的位置 . . . 236
6.4.1.3 步骤三：导入 PostgreSQL 导出的整库数据 . . . 236
6.4.2 PostgreSQL 整库迁移至 Hubble . . . 236
6.4.2.1 步骤一：导出需要迁移的整库 . . . 236
6.4.2.2 步骤二：将导出的数据文件放置于集群可访问到的位置 . . . 236
6.4.2.3 步骤三：导入 PostgreSQL 导出的整库数据 . . . 236
6.5 DB2 导入 . . . 237
6.5.1 步骤一: 导出建表语句 . . . 237
6.5.2 步骤二: 导出 csv 文件 . . . 237
6.5.3 步骤三:DB2 映射 hubble 数据类型 . . . 238
6.5.4 步骤四: 将导出的数据文件放置于集群可访问到的位置 . . . 239
6.5.5 步骤五: 导入数据 . . . 240

7 开发文档 240
7.1 go 编码指南 . . . 240
7.1.1 介绍 . . . 240
7.1.2 准则 . . . 241
7.1.2.1 指向接口的指针 . . . 241
7.1.2.2 接收器和接口 . . . 241
7.1.2.3 零值互斥锁是有效的 . . . 242
7.1.2.4 当需要时拷贝 Map 和 Slices . . . 243
7.1.2.5 延迟清理 . . . 245
7.1.2.6 Channel 的大小为 1 或者 None . . . 246
7.1.2.7 枚举类型从 1 开始 . . . 246
7.1.2.8 Error Types . . . 247
7.1.2.9 性能 . . . 251

HUBBLE

7.1.2.10 代码风格 . . . 252
7.1.2.11 Patterns . . . 265
7.2 go 的测试方案 . . . 269
7.2.1 单元测试 . . . 269
7.2.1.1 编写测试 . . . 269
7.2.1.2 编写示例 . . . 271
7.2.1.3 Mocking . . . 271
7.2.2 go-fuzz . . . 276
7.2.3 数据库测试体系 . . . 277
7.2.3.1 测试工具 . . . 277
7.2.3.2 异常测试 . . . 277
7.2.3.3 模糊测试 . . . 277
7.2.3.4 回归测试 . . . 278
7.2.3.5 资源泄漏测试 . . . 278
7.3 交互式测试脚本开发 . . . 278
7.3.1 expect 基础 . . . 278
7.3.1.1 基本命令 . . . 278
7.3.1.2 简单示例 . . . 279
7.3.1.3 模式-动作 . . . 279
7.3.1.4 传参 . . . 280
7.3.2 hubble 中的实际测试例子 . . . 280
7.3.3 计划 . . . 281
7.4 generate 命令 . . . 281
7.4.1 用法 . . . 283
7.4.1.1 自动化 . . . 283
7.4.2 枚举 . . . 284
7.5 cobra 包的使用 . . . 285
7.5.1 概念 . . . 285
7.5.1.1 Commands . . . 286
7.5.1.2 Flags . . . 286
7.5.2 安装 . . . 286
7.5.3 使用 . . . 286
7.5.4 后续 . . . 288
7.6 go 中使用 web . . . 288
7.6.1 web 工作方式的几个概念 . . . 289
7.6.1.1 Conn 的 goroutine . . . 289
7.6.1.2 ServeMux 的自定义 . . . 289
7.6.2 静态资源的使用 . . . 294
7.6.2.1 安装 . . . 294
7.6.2.2 使用 . . . 294
7.7 数据库工作原理 . . . 295
7.7.1 基础知识 . . . 296
7.7.1.1 O(1) 和 O(n2) . . . 296
7.7.1.2 归并排序 . . . 298
7.7.1.3 Array, Tree and Hash table . . . 301
7.7.2 全局概述 . . . 306
7.7.3 客户端管理器 . . . 307

HUBBLE

7.7.4 查询管理器 . . . 307
7.7.4.1 查询解析 . . . 308
7.7.4.2 查询改写器 . . . 308
7.7.4.3 统计 . . . 309
7.7.4.4 查询优化器 . . . 310
7.7.4.5 查询执行器 . . . 322
7.7.5 数据管理器 . . . 322
7.7.5.1 缓存管理器 . . . 322
7.7.5.2 预读取 . . . 323
7.7.5.3 写缓存 . . . 325
7.7.6 事务管理器 . . . 325
7.7.6.1 ACID . . . 325
7.7.6.2 并发控制 . . . 327
7.7.6.3 锁管理器 . . . 327
7.7.6.4 日志管理器 . . . 330
7.7.7 结论 . . . 335
7.8 分布式 ID 生成器 . . . 335
7.8.1 worker_id 的分配 . . . 336
7.8.2 开源实现 . . . 336
7.8.3 总结 . . . 337
7.9 分布式追踪系统 . . . 337
7.9.1 1. 介绍 . . . 337
7.9.1.1 为什么需要 tracing . . . 337
7.9.1.2 为什么需要 OpenTracing . . . 338
7.9.1.3 什么是一个 Trace . . . 338
7.9.1.4 一个典型的 Trace 案例 . . . 338
7.9.2 2. 快速入门 . . . 341
7.9.2.1 创建一个 web 工程 . . . 342
7.9.2.2 监控应用程序 . . . 342
7.9.2.3 连接到追踪系统 . . . 344
7.9.3 3. 概念和术语 . . . 346
7.9.3.1 平台无关的 API 语义 . . . 349
7.9.4 4.OpenTracing API . . . 351
7.9.4.1 OpenTracing 多语言支持 . . . 351
7.9.4.2 Data Conventions 数据约定 . . . 351
7.9.4.3 跨进程追踪 . . . 353
7.9.5 5.Best Practices 最佳实践 . . . 356
7.9.5.1 常见用例 . . . 356
7.9.5.2 追踪大型分布式系统 . . . 361
7.10 工具连接 . . . 366
7.10.1 下载 . . . 366
7.10.2 安装步骤 . . . 366
7.10.3 工具连接案例 . . . 366
7.10.3.1 步骤一: 新建连接 . . . 366
7.10.3.2 步骤二: 选择 SQL 下的 postgresql . . . 366
7.10.3.3 步骤三: 测试 . . . 366
7.10.3.4 步骤四: 新建模式 . . . 366

HUBBLE

7.10.3.5 步骤五: 新建 SQL 编辑器 . . . 370

8 论文 370

8.1 raft . . . 370
8.1.1 摘要 . . . 370
8.1.2 介绍 . . . 370
8.1.3 复制状态机 . . . 372
8.1.4 Paxos 算法的问题 . . . 373
8.1.5 为了可理解性的设计 . . . 374
8.1.6 Raft 一致性算法 . . . 374
8.1.6.1 Raft 基础 . . . 377
8.1.6.2 领导人选举 . . . 380
8.1.6.3 日志复制 . . . 381
8.1.6.4 安全性 . . . 384
8.1.6.5 跟随者和候选人崩溃 . . . 387
8.1.6.6 时间和可用性 . . . 387
8.1.7 集群成员变化 . . . 388
8.1.8 日志压缩 . . . 390
8.1.9 客户端交互 . . . 393
8.1.10 算法实现和评估 . . . 393
8.1.10.1 可理解性 . . . 393
8.1.10.2 正确性 . . . 394
8.1.10.3 性能 . . . 396
8.1.11 相关工作 . . . 397
8.1.12 结论 . . . 398
8.1.13 感谢 . . . 398
8.1.14 参考 . . . 398
8.2 calvin . . . 399
8.2.1 摘要 . . . 399
8.2.2 数据和定义 . . . 399
8.2.3 关键字 . . . 399
8.2.4 背景和简介 . . . 399
8.2.4.1 分布式事务的成本 . . . 399
8.2.4.2 副本一致性 . . . 400
8.2.4.3 在不增加争用的情况下达成协议 . . . 400
8.2.5 确定性数据库系统 . . . 400
8.2.6 系统架构 . . . 401
8.2.6.1 序列和复制 . . . 402
8.2.6.2 调度和并发控制 . . . 403
8.2.7 基于磁盘存储的 Calvin . . . 405
8.2.7.1 磁盘 I/O 延迟预测 . . . 406
8.2.7.2 全局热点记录追踪 . . . 406
8.2.8 检查点 . . . 406
8.2.9 性能和可扩展性 . . . 407
8.2.9.1 TPC-C benchmark . . . 407
8.2.9.2 处理高争用 . . . 410
8.2.10 相关工作 . . . 411

HUBBLE

8.2.11 未来工作 . . . 412
8.2.12 结论 . . . 413
8.2.13 致谢 . . . 413
8.2.14 引用 . . . 413
8.3 缓存和并发控制 . . . 414
8.3.1 摘要 . . . 414
8.3.2 介绍 . . . 414
8.3.3 背景和相关工作 . . . 415
8.3.3.1 分布式并发控制 . . . 415
8.3.3.2 多核并发控制 . . . 416
8.3.3.3 数据复制和缓存一致性 . . . 416
8.3.3.4 整合并发控制和缓存一致性 . . . 416
8.3.4 SUNDAIL 并发控制 . . . 416
8.3.4.1 逻辑租约 . . . 417
8.3.5 冲突处理 . . . 417
8.3.6 协议阶段 . . . 418
8.3.6.1 执行阶段 . . . 419
8.3.6.2 准备阶段 . . . 420
8.3.6.3 提交阶段 . . . 421
8.3.7 索引和幻读 . . . 422
8.3.8 容错性 . . . 422

HUBBLE

1 部署

1.1 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。

1.1.1 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

1.1.2 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 8 核
单台内存	64G
单台磁盘	480G*2 SSD + 1T*2 HDD
网络	千兆网卡
NFS 网盘	2T SSD

1.1.3 生产环境-推荐配置

名称	要求
机器数量	6+
单台 CPU	2 路主频 2.4GHz，每路至少 16 核
单台内存	128G+
单台磁盘	6 块 SSD 960G+
网络	2 块万兆网卡
NFS 网盘	2T SSD

注意:

CPU、内存:

- 建议:32核 CPU为比较好的性能状态

HUBBLE

- 建议:128G内存为比较好的性能状态

- 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 8 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，建议提供 32vcores 为比较好的性能选择。如还要进一步提高吞吐量，我们建议增加节点，而非继续增加物理硬件配置

存储:
+ hubble 对于硬盘随机读写要求比较高，强烈推荐 SSD 类型，建议每块磁盘 <=1T
+ 单台节点最大的数据存储建议不超过 1TiB，无论这个节点的 vcore 是多少 + 存储 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 + 建议永远保持磁盘空闲空间大于 50%，这样可以适应日常数据库的维护操作波动
+ 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
+ 数据盘以 /data1 /data2 方式命名
+ NFS 网盘用于数据还原备份指定的共享盘，可以使用共享存储，也可用在集群中，使用 1 台服务器中的一块硬盘作为 NFS 共享盘

1.1.4 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

1.1.5 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

1.2 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

1.2.1 系统级别配置

1.2.1.1 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

1.2.1.2 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
```

1.2.1.3 关闭透明大页

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

1.2.1.4 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

HUBBLE

1.2.1.5 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
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
```

HUBBLE

```
sysctl -p
```

1.2.1.6 ulimit 配置

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

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

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

1.2.1.7 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
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
```

HUBBLE

注意：
ifcfg-xx需要根据实际地址进行改动

1.2.1.8 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
```

1.2.1.9 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

1.2.1.10 NFS 配置（如果有网络共享盘跳过次步）

1.2.1.10.1 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.servic
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
```

1.2.1.10.2 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

1.2.2 服务器之间配置

1.2.2.1 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

1.2.2.2 hubble 数据库用户免密配置

> 规划的 Hubble 启动用户做免密登录

```
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
```

1.3 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

1.3.1 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
```

1.3.2 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 /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

1.3.3 hubble 数据库 license 注册激活

选取任意一个节点执行 license 注册

- 获取 license 机器码

```
cd /var/lib/hubble
/usr/local/bin/hubble license  machinecode --certs-dir=certs  #执行后会生成/var/
    ↪ lib/hubble/certs/BDPMACHINE.txt文件机器码
```

- 将/var/lib/hubble/certs/BDPMACHINE.txt 文件机器码发送到公司申请 license 激活码

- 激活 license
  1> 将 license 激活码写入到/var/lib/hubble/certs/BDPLIC.txt 文件
  2> 执行 license 注册激活

```
/usr/local/bin/hubble license register --certs-dir=/var/lib/hubble/certs
```

1.3.4 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=hubble01
    ↪ :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,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 --sql-audit-dir=/data/hubbledir/
    ↪ auditdir --log-dir=/data/hubbledir/logs --log-dir-max-size=1GiB --max-disk
    ↪ -temp-storage=10GiB --external-io-dir=/data_shares
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

HUBBLE

--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
--sql-audit-dir=/data/hubbledir/auditdir # 审计日志目录
--log-dir=/data/hubbledir/logs # 日志输出目录
--log-dir-max-size=1GiB # 设置日志文件生成大小
--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

1.3.5 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!
```

1.3.6 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/
```

> **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  # root的，也可以生成其他账户的
```

> **注意：**
> 如果在执行 create-client 客户端授权时，有的 license 申请期限时常较少，会出现”Error: failed to generate client certificate and key: xxx key: CA lifetime is 2364.193159h, shorter than the requested 43920.000000h. Renew CA certificate, or rerun with `--lifetime=2364h` for a shorter duration” 需要后面在执行命令后面添加 hubble cert create-client root --certs-dir=certs --ca-key=mysafedirectory/ca.key --overwrite `--lifetime=2364h` 在每个节点都执行一次

1.3.7 启动服务

> 使用 hubble 用户操作

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

1.3.8 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

1.3.9 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    #连接
show databases;
create user hubble with password 'Sjk-Qkl^2021379';
grant admin to hubble with admin option;

set cluster setting kv.snapshot_rebalance.max_rate='10GiB';
set cluster setting kv.snapshot_recovery.max_rate='10GiB';
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

1.3.10 测试验证

- 页面访问

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

- 查看进程

```
systemctl status hubble
```

1.3.11 设置开机自启动

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

1.4 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

1.4.1 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/
```

1.4.2 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

1.4.3 hubble 数据库 license 注册激活

- 获取 license 机器码

```
cd /var/lib/hubble
/usr/local/bin/hubble license  machinecode --certs-dir=certs  #执行后会生成/var/
    ↪ lib/hubble/certs/BDPMACHINE.txt文件机器码
```

- 将/var/lib/hubble/certs/BDPMACHINE.txt 文件机器码发送到公司申请 license 激活码

- 激活 license
  1> 将 license 激活码写入到/var/lib/hubble/certs/BDPLIC.txt 文件
  2> 执行 license 注册激活

```
/usr/local/bin/hubble license register --certs-dir=/var/lib/hubble/certs
```

注意:
有部分测试环境不带 license，可以跳过 license 注册激活

1.4.4 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,attrs=ssd,size=350GiB,rocksdb=write_buffer_size
    ↪ =134217728 --http-addr=0.0.0.0:48080 --sql-audit-dir=/data/hubbledir/
    ↪ auditdir1 --log-dir=/data/hubbledir/logs1 --log-dir-max-size=1GiB --
    ↪ 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
```

1.4.5 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,attrs=ssd,size=350GiB,rocksdb=write_buffer_size
    ↪ =134217728 --http-addr=0.0.0.0:48081 --sql-audit-dir=/data/hubbledir/
    ↪ auditdir2 --log-dir=/data/hubbledir/logs2 --log-dir-max-size=1GiB --
    ↪ external-io-dir=/data_shares
TimeoutStopSec=60
Restart=always
RestartSec=10
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=hubble2
User=hubble
```

HUBBLE

```
LimitNOFILE=1000000
[Install]
WantedBy=default.target
```

1.4.6 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,attrs=ssd,size=350GiB,rocksdb=write_buffer_size
    ↪ =134217728 --http-addr=0.0.0.0:48082 --sql-audit-dir=/data/hubbledir/
    ↪ auditdir3 --log-dir=/data/hubbledir/logs3 --log-dir-max-size=1GiB --
    ↪ 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
```

HUBBLE

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
--sql-audit-dir=/data/hubbledir/auditdir # 审计日志目录
--log-dir=/data/hubbledir/logs # 日志输出目录
--log-dir-max-size=1GiB # 设置日志文件生成大小
--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

1.4.7 添加 hubble 用户启停特权

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

##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!
```

1.4.8 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  # root的，也可以生成其他账户的
```

> **注意：**
> 如果在执行 create-client 客户端授权时，有的 license 申请期限时常较少，会出现"Error: failed to generate client certificate and key: xxx key: CA lifetime is 2364.193159h, shorter than the requested 43920.000000h. Renew CA certificate, or rerun with `--lifetime=2364h` for a shorter duration" 需要后面在执行命令后面添加 hubble cert create-client root --certs-dir=certs --ca-key=mysafedirectory/ca.key --overwrite `--lifetime=2364h`

1.4.9 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

1.4.10 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

1.4.11 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 'Sjk-Qkl^2021379';
grant admin to hubble with admin option;

set cluster setting kv.snapshot_rebalance.max_rate='10GiB';
set cluster setting kv.snapshot_recovery.max_rate='10GiB';
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

1.4.12 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
```

1.4.13 设置开机自启动

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

注意：
使用 root 操作

2 SQL

2.1 数据类型

2.1.1 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

HUBBLE

类型	描述	样例	向量量化执行
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-91e0-56 ↪ a2d5a246ec	支持

2.1.2 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。

2.1.3 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

2.1.3.1 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。

2.1.3.2 大小

ARRAY 的值大小是可变的，为确保性能最好 <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。

2.1.3.3 示例

2.1.3.3.1 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)
```

2.1.3.3.2 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)
```

2.1.3.3.3 使用数组索引访问数组元素

```
> SELECT * FROM c;
```

```
+-------------+
|      d      |
+-------------+
| {10,20,30} |
+-------------+
(1 row)
```

HUBBLE

```
> SELECT d[2] FROM c;
```

```
+------+
| d[2] |
+------+
|   20 |
+------+
(1 row)
```

2.1.3.3.4 将元素追加到数组

使用array_append函数

```
> SELECT * FROM c;
```

```
+------------+
|     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)
```

2.1.3.3.5 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;
```

HUBBLE

```
+------------------+
|        d         |
+------------------+
| {10,20,30,40,50} |
+------------------+
(1 row)
```

2.1.3.4 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[];
```

```
   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，以与 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)
```

2.1.4 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。

2.1.4.1 别名

名称BIT VARYING是VARBIT的别名

HUBBLE

2.1.4.2 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。

2.1.4.3 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 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。

2.1.4.4 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;
```

HUBBLE

```
  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)
```

2.1.4.5 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'。

2.1.5 BOOL

BOOL数据类型存储布尔值false或true。

2.1.5.1 别名

BOOLEAN是BOOL的别名。

2.1.5.2 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)

2.1.5.3 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。

HUBBLE

2.1.5.4 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)
```

2.1.5.5 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	--

2.1.6 BYTES

BYTES数据类型存储可变长度的二进制字符串。

2.1.6.1 别名

BYTEA和BLOB是BYTES的别名。

HUBBLE

2.1.6.2 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'...'。

2.1.6.3 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。

2.1.6.4 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)
```

2.1.6.5 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'\x48AA'。
- 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'0\252'。

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。

HUBBLE

2.1.6.5.1 STRING vs. 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

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。

2.1.7 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）。

2.1.7.1 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。

2.1.7.2 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 BCP 47 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

2.1.7.3 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);
```

2.1.7.4 示例

2.1.7.4.1 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)
```

2.1.7.4.2 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;
```

HUBBLE

```
    name
+----------+
  Backhaus
  Baz
  Bär
(3 rows)
```

2.1.7.4.3 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)
```

2.1.7.4.4 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);
```

HUBBLE

```
  pg_collation_for
+------------------+
  de
(1 row)
```

2.1.8 DATE

该DATE 数据类型储存年月日信息。

2.1.8.1 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。

2.1.8.2 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。

2.1.8.3 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;
```

HUBBLE

```
+----------------------------+-------+
                a                |   b
+----------------------------+-------+
  2016-03-26 00:00:00+00:00 | 12345
  2016-03-27 00:00:00+00:00 | 12345
+----------------------------+-------+
(2 rows)
```

2.1.8.4 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 时间至今的天数（1970 年 1 月 1 日）
FLOAT	转换为 Unix 时间至今的天数（1970 年 1 月 1 日）
TIMESTAMP	将结果时间戳记中的时间设置为 00:00（午夜）。
INT	转换为 Unix 时间至今的天数（1970 年 1 月 1 日）
STRING	--

2.1.9 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。

2.1.9.1 别名

DEC和NUMRIC是DECIMAL的别名

2.1.9.2 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。

2.1.9.3 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:

HUBBLE

语法	值
inf, infinity, +inf, +infinity	+∞
-inf, -infinity	-∞
nan	(Not-a-Number)

举例:

- DECIMAL '+Inf'
- '-Inf'::DECIMAL
- CAST('NaN' AS DECIMAL)

2.1.9.4 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。

2.1.9.5 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.
```

HUBBLE

2.1.9.6 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	--

2.1.10 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。

2.1.10.1 别名

Name	Aliases
FLOAT	None
REAL	FLOAT4
DOUBLE PRECISION	FLOAT8

2.1.10.2 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	+∞
-inf, -infinity	-∞
nan	(Not-a-Number)

举例:

- DECIMAL '+Inf'
- '-Inf'::DECIMAL
- CAST('NaN' AS DECIMAL)

2.1.10.3 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。

2.1.10.4 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, 2.0123456789012345678, CAST('+Inf'
  ↪  AS FLOAT));
```

```
> SELECT * FROM floats;
```

```
        a        |         b          |  c
+----------------+--------------------+------+
  1.012345678901 | 2.0123456789012346 | +Inf
(1 row)
# Note that the value in "b" has been limited to 17 digits.
```

2.1.10.5 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
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	--

2.1.11 INET

INET数据类型存储 IPv4 或 IPv6 地址。

2.1.11.1 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。

HUBBLE

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

2.1.11.2 大小

使用INET值存储 IPv4 占用 32 位，存储 IPv6 占用 128 位。

2.1.11.3 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');
```

HUBBLE

```
> SELECT * FROM computers;
```

```
                   ip                    |    user_email     |
                      ↪ registration_date
+--
   ↪ ----------------------------------+-------------------+-------------------------
   ↪
  192.168.0.2/10                         | lauren@google.com | 2018-01-31
     ↪ 00:00:00+00:00
  192.168.0.1                            | info@google.com   | 2018-01-31
     ↪ 00:00:00+00:00
  2001:4f8:3:ba:2e0:81ff:fe22:d1f1/120 | test@google.com   | 2018-01-31
     ↪ 00:00:00+00:00
(3 rows)
```

2.1.11.4 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。

2.1.12 INT

支持各种有符号整数数据类型。

2.1.12.1 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

2.1.12.2 语法

数值文本可以作为INT类型的输入。例如：42，-1234或0xCAFE

2.1.12.3 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）。

2.1.12.3.1 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。

2.1.12.4 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)
```

HUBBLE

2.1.12.5 支持的转换

INT值可以强制转换为以下数据类型:

HUBBLE

Type	Details
DECIMAL	—
FLOAT	如果INT值的大小大于 2 ^ 53，则会失去精度。
BIT	转换为整数值的二进制表示形式。如果该值为负，则在左侧复制符号位以填充整个位数组。
BOOL	0 转换为 false；所有其他值都将转换为 true。
DATE	转换为自 Unix 时间至今的日子（1970 年 1 月 1 日）
TIMESTAMP	转换为自 Unix 时间至今的日子（1970 年 1 月 1 日）
INTERVAL	转换为微秒。
STRING	—

2.1.13 INTERVAL

INTERVAL数据类型存储一个表示时间跨度的值。

2.1.13.1 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。

2.1.13.2 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。

2.1.13.3 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。

2.1.13.4 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)
```

2.1.13.5 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h-m-s格式（微秒精度）
TIME	转换为HH：MM：SS.SSSSSS，该时间等于午夜后的时间间隔（微秒精度）

2.1.14 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。

2.1.14.1 别名

HUBBLE

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。

2.1.14.2 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。

2.1.14.3 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。

2.1.14.4 大小

JSONB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。

2.1.14.5 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。

HUBBLE

有关受支持的JSONB函数的完整列表。

2.1.14.6 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。

2.1.14.7 示例

2.1.14.7.1 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;
```

HUBBLE

```
                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)
```

2.1.14.7.2 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)
```

2.1.14.7.3 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
+------------+------------+
  "Wang"     | "ShangHai"
  "Zhang"    | "Beijing"
(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
+------------+----------+
  Wang       | ShangHai
  Zhang      | Beijing
(2 rows)
```

2.1.14.7.4 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"}
```

HUBBLE

```
(3 rows)
```

主键id作为profile列中的字段进行计算。

2.1.14.8 支持的转换

JSONB值可以转换为以下数据类型:

- STRING

2.1.15 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。

2.1.15.1 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。

2.1.15.1.1 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。

2.1.15.1.2 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。

2.1.15.2 示例

2.1.15.2.1 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)
```

2.1.15.2.2 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);
```

2. 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;
```

3. 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。

HUBBLE

2.1.15.2.3 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。

2.1.16 STRING

STRING 数据类型存储一串 Unicode 字符。

2.1.16.1 别名

STRING的别名

- CHARACTER
- CHAR

HUBBLE

- VARCHAR
- TEXT

STRING(N)的别名：

- CHARACTER(n)
- CHARACTER VARYING(n)
- CHAR(n)
- CHAR VARYING(n)
- VARCHAR(n)

2.1.16.2 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。

2.1.16.3 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。

2.1.16.3.1 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。

2.1.16.4 大小

STRING值的大小是可变的，但建议将值保持在 64 KB 以下以确保性能。超过该阈值，写放大和其他考虑因素可能导致性能显着下降。

2.1.16.5 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)
```

2.1.16.6 支持的转换

可以将STRING值强制转换为以下任何数据类型：

HUBBLE

类型	细节
BIT	需要支持BIT字符串类型, 如: '101001'.
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: '1h2m3s4ms5us6ns'.
TIME	需要支持TIME字符串类型, 如: '01:22:12' (微秒精度).
TIMESTAMP	需要支持TIMESTAMP字符串类型, 如: '2019-01-28 10:10:10.222222'.

`STRING` vs. `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。

2.1.17 TIME

TIME数据类型以 UTC 格式存储一天中的时间。

2.1.17.1 别名

TIME WITHOUT TIME ZONE

2.1.17.2 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。

2.1.17.3 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。

2.1.17.4 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 | 0000-01-01 08:40:00+00:00
        2 | 0000-01-01 08:41:40+00:00
(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);
```

HUBBLE

```
(SELECT time_val FROM "time"  WHERE time_id = 1) < (SELECT  time_val FROM "time"
    ↪  WHERE  time_id = 2)
+--
    ↪ ------------------------------------------------------------------------------
    ↪
              true
(1 row)
```

2.1.17.5 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.SSSSSS”（微秒精度）

2.1.18 TIMESTAMP / TIMESTAMPTZ

TIMESTAMP和TIMESTAMPTZ数据类型以 UTC 格式存储日期和时间对。

2.1.18.1 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()。

2.1.18.2 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。

2.1.18.3 别名

- TIMESTAMP, TIMESTAMP WITHOUT TIME ZONE
- TIMESTAMPTZ, TIMESTAMP WITH TIME ZONE

2.1.18.4 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：

HUBBLE

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。

2.1.18.5 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。

2.1.18.6 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 10:10:10-05:00'),
   ↪ (2, TIMESTAMPTZ '2019-03-25');
```

```
> SELECT * FROM timestamps;
```

```
  a |             b
+---+---------------------------+
  1 | 2019-03-25 15:10:10+00:00
  2 | 2019-03-25 00:00:00+00:00
(2 rows)
```

2.1.18.7 支持的转换

可以将TIMESTAMP值强制转换为以下任何数据类型：

HUBBLE

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	--

2.1.19 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。

2.1.19.1 句法

UUID值可以使用以下格式表示：

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

2.1.19.2 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。

2.1.19.3 示例

2.1.19.3.1 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)
```

2.1.19.3.2 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,
```

HUBBLE

```
        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。

HUBBLE

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+）。

2.1.19.4 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	需要支持BYTES 字符串格式，如：b'\141\061\142\062\143\063'.

2.2 函数

2.2.1 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

HUBBLE

语法	相当于	描述
CURRENT_TIMESTAMP	current_timestamp()	现在的时间戳
CURRENT_USER	current_user()	现在的用户
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

2.2.2 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>)
```

HUBBLE

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 ... )
```

HUBBLE

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。

2.2.3 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[]

HUBBLE

函数	说明	返回值类型
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[]
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

HUBBLE

函数	说明	返回值类型
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
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[]

HUPBLIC

函数	说明	返回值类型
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[]
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[]

HUBBLE

函数	说明	返回值类型
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[]
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[]

HUBBLE

函数	说明	返回值类型
`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[]

HUBBLE

函数	说明	返回值类型
array_to_string(input: anyelement[], delim: string)	用定界符将数组连接到字符串中。	string| | 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(string, delimiter: string)| 将字符

HUBBLE

函数	说明	返回值类型

案例:

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}
```

- append(array: x[], elem: y) → bool[]

数组拼接单个元素

```
select array_append(a,'Thursday') from test_array;
             array_append
+-------------------------------------+

  {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
```

HUBBLE

- 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
```

- 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}
```

2.2.4 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

HUBBLE

函数	说明	返回值类型
least(anyelement...)	返回具有最小值的元素。	anyelement
ltrim('m','n')	返回具有最左边的元素。	string

案例:

- greatest(anyelement...)

取元素中的最大值

```
select greatest(1,4,6,9,12,5,7);

  greatest
+----------+
        12
```

- least(anyelement...)

取元素中的最小值

```
select least(8,1,4,6,9,12,5,7);

  least
+-------+
      1
```

取值取左边的字符串

```
select ltrim('m','n');
  ltrim
+-------+
  m
```

2.2.5 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

HUBBLE

函数	说明	返回值类型
`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
`to_char(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
`to_char(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
`to_char(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
`to_date(input: string, format: string)`	input使用format（使用标准strptime格式）作为时间戳返回。	timestamp
`to_datetz(input: string, format: string)`	input使用format（使用标准strptime格式）作为时间戳返回。	timestamptz
`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
`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

HUBBLE

函数	说明	返回值类型
`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
`cast()`	返回当前交易的时间。将时间转换成字符串类型并截取出来。	string

案例:

- age(end: timestamptz, begin: timestamptz)

计算 end 和 begin 之间的时间间隔

```
select age(TIMESTAMPTZ '2019-03-26 14:22:10+08:00',TIMESTAMPTZ '2019-03-26
    ↪ 10:10:10-05:00');

     age
+-----------+
  -08:48:00
```

- age(val: timestamptz)

与当前时间的间隔

```
select age(TIMESTAMPTZ '2019-03-26 14:22:10+08:00');

         age
+--------------------+
  6531:32:02.512077
```

- current_date()

当前日期

- current_timestamp()

当前时间

- date_trunc(x,input:date)

HUBBLE

截断时间，后边清零，x(year, quarter, month, week, hour, minute, second, millisecond, microsecond)

```
select date_trunc('month',TIMESTAMPTZ '2019-03-26 14:22:10+08:00');

         date_trunc
+---------------------------+
  2019-03-01 00:00:00+08:00
(1 row)
```

- extract(x,input:date)

input 中提取 x(year, quarter, month, week, hour, minute, second, millisecond, microsecond)

```
select extract('month',TIMESTAMPTZ '2019-03-26 14:22:10+08:00');

  extract
+---------+
        3
```

timestamp 类型怎么能转成字符串并截串

```
select cast('20220107' as string);
    text
+----------+
  20210107
```

2.2.6 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

(1 row)
```

- unique_rowid()

```
 select unique_rowid();
     unique_rowid
+--------------------+
```

HUBBLE

```
  514553266254184454
(1 row)
```

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。

2.2.7 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	decimal	decimal	计算立方根
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

HUBBLE

函数	返回类型	参数类型	说明
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
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

2.2.8 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
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

HUBBLE

函数	返回类型	参数	说明
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。如果 v 参数为空，则返回 “NULL”。
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
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。

HUBBLE

函数	返回类型	参数	说明
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

2.2.9 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

HUBBLE

函数	说明	返回值类型
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
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

HUBBLE

函数	说明	返回值类型
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
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

HUDT

函数	说明	返回值类型
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
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

HUBBLE

函数	说明	返回值类型
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
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

HUBBLE

函数	说明	返回值类型
`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
`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

HUBBLE

函数	说明	返回值类型
`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
`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

HUBBLE

函数	说明	返回值类型
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

2.2.10 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
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

HUDBLE

函数	说明	返回值类型
`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

2.3 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。

2.3.1 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

2.3.2 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

列名	描述
is_grantable	是否可授权

2.3.3 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

2.3.4 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。

2.3.5 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

HUBBLE

列名	描述
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

2.3.6 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
table_name	表明
column_name	列名
privilege_type	授权类型
is_grantable	保留字段

2.3.7 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

2.3.8 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

2.3.9 key_column_usage

使用主键、惟一键或外键/引用约束的列信息

HUBBLE

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)

2.3.10 referential_constraints

列名	描述
constraint_catalog	包含约束的数据库的名称
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

2.3.11 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

2.3.12 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

2.3.13 schemata

数据库的 schema 信息

列名	描述
table_catalog	数据库名称
table_schema	schema 名称
default_character_set_name	保留字段
sql_path	保留字段

2.3.14 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

2.3.15 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。

HUBBLE

列名	描述
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

2.3.16 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

2.3.17 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

2.3.18 tables

表与视图在数据库中的信息

HUBBLE

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

2.3.19 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

2.3.20 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

HUBBLE

2.4 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，它通过可 serializable isolation 事务级别，将其与其他他并发事务中隔离出来

2.4.1 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。

2.4.2 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 ROLLBAC 语句即可中止事务。

2.4.3 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。

2.4.4 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 PROIORITY <LOW | NORMAL | HIGH>;
```

客户端可以查看当前事务的优先级

```
SHOW TRANSACTION PRIORITY;
```

2.5 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。

2.5.1 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。

2.5.2 约束条件的使用

2.5.2.1 添加约束条件

HUBBLE

1. 创建表并在单列上添加主键约束

```
CREATE TABLE foo (a INT PRIMARY KEY);
```

2. 创建表并在多列上添加主键约束

```
CREATE TABLE bar (a INT, b INT, PRIMARY KEY (a,b));
```

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 baz ADD CONSTRAINT id_unique UNIQUE (id);
```

```
ALTER TABLE baz ALTER COLUMN bool SET DEFAULT true;
```

备注：PRIMARY KEY 和 NOT NULL 约束只能在创建时添加。

2.5.3 创建主键时定义约束的名称

```
CREATE TABLE foo (a INT CONSTRAINT another_name PRIMARY KEY);
```

```
CREATE TABLE bar (a INT, b INT, CONSTRAINT yet_another_name PRIMARY KEY (a,b));
```

2.5.4 列出约束条件

```
SHOW CONSTRAINTS or SHOW CREATE.
```

2.5.5 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。

2.5.6 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。

HUBBLE

约束类型	操作
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）。

2.6 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。

2.6.1 创建视图

要创建视图，请使用以下 CREATE VIEW 语句：

```
CREATE VIEW tpcc.user_accounts
  AS SELECT type, email
  FROM tpcc.accounts;
```

2.6.2 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 tpcc;
```

```
+---------------+
|     Table     |
+---------------+
| accounts      |
| user_accounts |
+---------------+
(2 rows)
```

要仅列出视图，可以 views 在信息模式中查询表：

```
SELECT * FROM tpcc.information_schema.views;
```

```
+---------------+------------------+---------------------+-------------------------
    ↪
| table_catalog |   table_schema   |      table_name     |
    ↪ view_definition                  | check_option | is_updatable |
    ↪ is_insertable_into | is_trigger_updatable | is_trigger_deletable |
    ↪ is_trigger_insertable_into |
+---------------+------------------+---------------------+-------------------------
    ↪
| tpcc          | public           | user_accounts       | SELECT type, email
    ↪ FROM tpcc.accounts       | NULL         | NULL         | NULL
    ↪               | NULL                | NULL                | NULL
    ↪                       |
+---------------+------------------+---------------------+-------------------------
    ↪
```

HUBBLE

```
(1 row)
```

2.6.3 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 tpcc.user_accounts;
```

SELECT 视图执行后将执行该视图的存储 SELECT 语句，该语句从基础表返回相关数据。要检查 SELECT 视图执行的语句，请使用以下 SHOW CREATE 语句：

```
SHOW CREATE tpcc.user_accounts;
```

2.6.4 重命名视图

要重命名视图，请使用以下 ALTER VIEW 语句：

```
ALTER VIEW tpcc.user_accounts RENAME TO tpcc.user_accts;
```

2.6.5 删除视图

要删除视图，请使用以下 DROP VIEW 语句

```
DROP VIEW tpcc.user_accounts
```

2.7 SQL 语法手册

2.7.1 DCL

2.7.1.1 USER

2.7.1.1.1 查看所有用户

所需权限：

> 需要用 admin 权限

```
SHOW USERS;
```

2.7.1.1.2 创建用户

CREATE USER 语句创建库用户，使你可以控制数据库的表的权限。

> **注意事项：**
>
> - 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

HUBBLE

- 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

用户必须在 system.users 表上具有insert和update权限

创建一个用户

```
CREATE USER beagledata;
```

注意

创建用户后，必须授予用户对数据库相关权限。

使用密码创建用户

```
CREATE USER beagledata WITH PASSWORD 'abcd1234';
```

2.7.1.1.3 删除一个或多个用户

所需权限

用户必须拥有system.users表的 delete 权限。

必须先撤销用户的所有权限，然后才能删除该用户

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)
```

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;
```

2.7.1.1.4 修改和新增用户名密码

仅在非root用户的 secure 集群中支持密码创建和更改

所需权限

用户需要对system.users表有INSERT和UPDATE权限

```
ALTER USER beagledata WITH PASSWORD "Pass@Word";
```

2.7.1.2 ROLE

2.7.1.2.1 SHOW ROLES

查看所有数据库上的角色，需要admin权限。

```
SHOW ROLES;
```

2.7.1.2.2 CREATE ROLE

创建一个角色，需要超管或 admin 权限，root 用户默认是 admin 权限。

角色的所有特权均由其所有成员继承。

```
CREATE ROLE dev;
```

2.7.1.2.3 DROP ROLE

删除一个角色，需要 admin 权限。

```
DROP ROLE dev;
```

2.7.1.3 PRIVILEGE

2.7.1.3.1 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
```

HUBBLE

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;
```

2.7.1.3.2 GRANT

用于添加一个角色或者角色中的一个用户。

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。

权限	级别
ALL	Database,Table
CREATE	Database,Table
DROP	Database,Table
GRANT	Database,Table
SELECT	Table
INSERT	Table
DELETE	Table
UPDATE	Table

给用户授予数据库的权限

```
GRANT CREATE ON DATABASE hubbletest TO beagledata;
```

查看数据库权限

```
SHOW GRANTS ON DATABASE hubbletest;
```

```
+---------------+--------------------+---------+----------------+
| database_name |    schema_name     | grantee | privilege_type |
+---------------+--------------------+---------+----------------+
| database_0903 | hubble_internal    | admin   | ALL            |
| database_0903 | hubble_internal    | beagle  | CREATE         |
| database_0903 | hubble_internal    | root    | ALL            |
| database_0903 | information_schema | admin   | ALL            |
+---------------+--------------------+---------+----------------+
(4 rows)
```

给角色授予数据库的权限

```
GRANT CREATE ON DATABASE hubbletest TO dev;
```

查看角色权限

```
SHOW GRANTS FOR dev;
```

给用户授予表的权限

```
GRANT DELETE ON TABLE hubbletest.orders TO beagledata;
```

查看表权限

```
SHOW GRANTS ON TABLE hubbletest.orders;
```

```
+---------------+-------------+------------+---------+----------------+
| database_name | schema_name | table_name | grantee | privilege_type |
+---------------+-------------+------------+---------+----------------+
| database_0903 | public      | person     | admin   | ALL            |
| database_0903 | public      | person     | beagle  | CREATE         |
| database_0903 | public      | person     | beagle  | DELETE         |
+---------------+-------------+------------+---------+----------------+
(3 rows)
```

2.7.1.3.3 REVOKE

用于回收一个用户的角色。

所需权限

> 授予角色成员资格的用户必须拥有GRANT权限。

回收给用户的数据库的权限

```
REVOKE CREATE ON DATABASE hubbletest FROM beagledata;
```

```
SHOW GRANTS ON DATABASE hubbletest;
```

```
+----------------------+--------------------+---------+----------------+
|    database_name     |    schema_name     | grantee | privilege_type |
+----------------------+--------------------+---------+----------------+
| hubble_database_0902 | hubble_internal    | admin   | ALL            |
| hubble_database_0902 | hubble_internal    | root    | ALL            |
+----------------------+--------------------+---------+----------------+
(2 rows)
```

回收给角色的数据库的权限

```
REVOKE CREATE ON DATABASE hubbletest FROM dev;
SHOW GRANTS ON DATABASE hubbletest FOR dev;
```

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;
```

2.7.2 DDL

2.7.2.1 ADD COLUMN

在表内添加一个新列

所需权限

> 用户必须有表的CREATE权限。

参数

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。

添加单列

```
ALTER TABLE person ADD COLUMN age INT;
```

```
  column_name |  data_type  | is_nullable | column_default |
    ↪ generation_expression | indices   |is_hidden
+-------------+-------------+-------------+----------------+----------------------+------
  ↪
  name        | STRING      |    true     | NULL           |
    ↪                       | {}          |   false
  id          | INT8        |    false    | NULL           |
    ↪                       | {primary}   |   false
  email       | STRING      |    true     | NULL           |
    ↪                       |             |   false
  adress      | STRING      |    true     | NULL           |
    ↪                       | {}          |   false
  age         | INT8        |    true     | NULL           |
    ↪                       | {}          |   false
(5 rows)
```

添加多列

```
ALTER TABLE company ADD COLUMN adress STRING, ADD COLUMN zczb INT;
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
    ↪  |  indices  | is_hidden
+-------------+-----------+-------------+----------------+-----------------------+--------
  ↪
  id          | INT8      |    false    | NULL           |
    ↪                       | {primary} |   false
  name        | STRING    |    true     | NULL           |
    ↪                       | {}        |   false
  clrq        | DATE      |    true     | NULL           |
    ↪                       | {}        |   false
  adress      | STRING    |    true     | NULL           |
    ↪                       | {}        |   false
  zczb        | INT8      |    true     | NULL           |
    ↪                       | {}        |   false
(5 rows)
```

添加具有 NOT NULL 约束和 DEFAULT 值的列

```
ALTER TABLE company ADD COLUMN money DECIMAL NOT NULL DEFAULT (DECIMAL '
  ↪ 123456789');
```

```
  column_name | data_type | is_nullable |      column_default        |
    ↪ generation_expression |  indices  | is_hidden
+-------------+-----------+-------------+----------------------------+-------------------
  ↪
  id          | INT8      |    false    | NULL                       |
    ↪                       | {primary} |   false
  name        | STRING    |    true     | NULL                       |
    ↪                       | {}        |   false
  clrq        | DATE      |    true     | NULL                       |
    ↪                       | {}        |   false
  adress      | STRING    |    true     | NULL                       |
    ↪                       | {}        |   false
  zczb        | INT8      |    true     | NULL                       |
    ↪                       | {}        |   false
  money       | DECIMAL   |    false    | 123456789:::DECIMAL::DECIMAL |
    ↪                       | {}        |   false
(6 rows)
```

添加带有 UNIQUE 约束列，并且值非空

```
ALTER TABLE company ADD COLUMN tel DECIMAL UNIQUE NOT NULL;
```

```
  column_name | data_type | is_nullable |        column_default        |
    ↪ generation_expression |          indices          | is_hidden
+-------------+-----------+-------------+------------------------------+-------------------
  ↪
```

```
  id          | INT8      |    false    | NULL                          |
     ↪                       | {primary,company_tel_key} |   false
  name        | STRING    |    true     | NULL                          |
     ↪                       | {}                        |   false
  clrq        | DATE      |    true     | NULL                          |
     ↪                       | {}                        |   false
  adress      | STRING    |    true     | NULL                          |
     ↪                       | {}                        |   false
  zczb        | INT8      |    true     | NULL                          |
     ↪                       | {}                        |   false
  money       | DECIMAL   |    false    | 123456789:::DECIMAL::DECIMAL |
     ↪                       | {}                        |   false
  tel         | DECIMAL   |    false    | NULL                          |
     ↪                       | {company_tel_key}         |   false
(7 rows)
```

添加带有排序规则的列

```
ALTER TABLE company ADD COLUMN email STRING COLLATE en;
```

```
  column_name |     data_type     | is_nullable |        column_default        |
     ↪  generation_expression |          indices          | is_hidden
+-------------+-------------------+-------------+------------------------------+----------
   ↪
  id          | INT8              |    false    | NULL                          |
     ↪                       | {primary,company_tel_key} |   false
  name        | STRING            |    true     | NULL                          |
     ↪                       | {}                        |   false
  clrq        | DATE              |    true     | NULL                          |
     ↪                       | {}                        |   false
  adress      | STRING            |    true     | NULL                          |
     ↪                       | {}                        |   false
  zczb        | INT8              |    true     | NULL                          |
     ↪                       | {}                        |   false
  money       | DECIMAL           |    false    | 123456789:::DECIMAL::DECIMAL |
     ↪                       | {}                        |   false
  tel         | DECIMAL           |    false    | NULL                          |
     ↪                       | {company_tel_key}         |   false
  email       | STRING COLLATE en |    true     | NULL                          |
     ↪                       | {}                        |   false
(8 rows)
```

添加一列并将其分给新的列族

```
ALTER TABLE company ADD COLUMN creat_time DATE CREATE FAMILY new_time;
```

```
SHOW CREATE TABLE company;  // 查看建表语句
```

```
  table_name |                              create_statement
+------------+-------------------------------------------------------------------------+
   ↪
  company    | CREATE TABLE company (
             |     id INT8 NOT NULL,
             |     name STRING NULL,
             |     clrq DATE NULL,
             |     adress STRING NULL,
             |     zczb INT8 NULL,
             |     money DECIMAL NOT NULL DEFAULT 123456789:::DECIMAL::DECIMAL,
             |     tel DECIMAL NOT NULL,
             |     email STRING COLLATE en NULL,
             |     creat_time DATE NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (id ASC),
             |     UNIQUE INDEX company_tel_key (tel ASC),
             |     FAMILY "primary" (id, name, clrq, adress, zczb, money, tel,
                ↪ email),
             |     FAMILY new_time (creat_time)
             | )
(1 row)
```

添加一列分给现有列族

```
ALTER TABLE company ADD COLUMN time string FAMILY new_time;
```

```
  table_name |                              create_statement
+------------+-------------------------------------------------------------------------+
   ↪
  company    | CREATE TABLE company (
             |     id INT8 NOT NULL,
             |     name STRING NULL,
             |     clrq DATE NULL,
             |     adress STRING NULL,
             |     zczb INT8 NULL,
             |     money DECIMAL NOT NULL DEFAULT 123456789:::DECIMAL::DECIMAL,
             |     tel DECIMAL NOT NULL,
             |     email STRING COLLATE en NULL,
             |     creat_time DATE NULL,
             |     "time" STRING NULL,
             |     CONSTRAINT "primary" PRIMARY KEY (id ASC),
             |     UNIQUE INDEX company_tel_key (tel ASC),
             |     FAMILY "primary" (id, name, clrq, adress, zczb, money, tel,
                ↪ email),
             |     FAMILY new_time (creat_time, "time")
             | )
(1 row)
```

HUBBLE

如果列族不存在，则添加列并创建列族

```
ALTER TABLE company ADD COLUMN new_time STRING CREATE IF NOT EXISTS FAMILY
    ↪ n_time;
```

```
  table_name |                                create_statement
+------------+-------------------------------------------------------------------------------+
    ↪
  company    | CREATE TABLE company (
             |     id INT8 NOT NULL,
             |     name STRING NULL,
             |     clrq DATE NULL,
             |     adress STRING NULL,
             |     zczb INT8 NULL,
             |     money DECIMAL NOT NULL DEFAULT 123456789:::DECIMAL::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 (id ASC),
             |     UNIQUE INDEX company_tel_key (tel ASC),
             |     FAMILY "primary" (id, name, clrq, adress, zczb, money, tel,
                ↪ email),
             |     FAMILY new_time (creat_time, "time"),
             |     FAMILY n_time (new_time)
             | )
(1 row)
```

> 提示：FAMILY语句见COLUMN FAMILY内容

2.7.2.2 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

所需权限

> 需要有表的CREATE权限。

添加约束：UNIQUE

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);
```

```
  column_name | data_type | is_nullable | column_default | generation_expression
     ↪  |        indices        | is_hidden
+-------------+-----------+-------------+----------------+-----------------------+--------
   ↪
  id          | INT8      |    false    | NULL           |
     ↪                       | {primary,id_unique} |   false
(1 row)
```

添加约束：check

```
ALTER TABLE shop ADD CONSTRAINT money_check CHECK (money > 0);
```

```
  table_name | constraint_name | constraint_type |       details        |
     ↪ validated
+------------+-----------------+-----------------+----------------------+-----------+
   ↪
  shop       | id_unique       | UNIQUE          | UNIQUE (id ASC)      |   true
  shop       | money_check     | CHECK           | CHECK ((money > 0))  |   true
  shop       | primary         | PRIMARY KEY     | PRIMARY KEY (id ASC) |   true
(3 rows)
```

添加FOREIGN KEY约束

> **注意**：
>
> 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 shipments1 ADD CONSTRAINT fk_customers FOREIGN KEY (customer_id)
   ↪ REFERENCES customers1(id);
```

```
SHOW CONSTRAINTS FROM shipments;
```

2.7.2.3 CREATE DATABASE

创建数据库

```
CREATE DATABASE IF NOT EXISTS my_hubble_database;
```

2.7.2.4 CREATE TABLE

参数

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 identifier rules. 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。
column_def	定义列的逗号分隔列表。每列需要名称/标识符和数据类型; 列级约束或其他列限定（例如计算列）的指定是可选项。列名在表中必须是唯一的，但可以与索引或约束具有相同的名称。
index_def	可选项，定义索引的逗号分隔列表。对于每个索引，必须指定要索引的列; 可选择指定名称。索引名称在表中必须是唯一的，并遵循标识符规则。请参阅下面的创建具有辅助索引和倒排索引的表示例。
family_def	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。
table_constraint	可选项，表级约束的逗号分隔列表。约束名称在表中必须是唯一的，但可以与列，列族或索引具有相同的名称
opt_interleave	你可以通过交错表来优化查询性能，这会更改 hubble 存储数据的方式。
opt_partition_by	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。

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;
```

```
+-------------+--------+-------+---------+---------------------------------+
|    Field    |  Type  | Null  | Default |             Indices             |
+-------------+--------+-------+---------+---------------------------------+
| user_id     | INT    | false | NULL    | {primary,logoff_user_email_key} |
| user_email  | STRING | true  | NULL    | {logoff_user_email_key}         |
| logoff_date | DATE   | true  | NULL    | {}                              |
+-------------+--------+-------+---------+---------------------------------+
(3 rows)
```

```
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)
```

CREATE TABLE AS

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。

HUBBLE

参数	详情
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。
name	要使用的列的名称，而不是 select_stmt 中列的名称
select_stmt	用于提供数据的选择查询

> 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。

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 logtable_copy (
                |     user_id INT8 NULL,
                |     user_name STRING NULL,
                |     log_date DATE NULL,
                |     FAMILY "primary" (user_id, user_name, log_date, rowid)
                | )
(1 row)
```

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    |                          CreateTable
    ↪ |
+-------------+-------------------------------------------------------------+
    ↪
```

```
| logoff_copy | CREATE TABLE logoff_copy (
    ↪ |
|             |     user_id INT NULL,
    ↪ |
|             |     user_email STRING NULL,
    ↪ |
|             |     logoff_date DATE NULL,
    ↪ |
|             |     INDEX logoff_copy_id_idx (user_id ASC),
    ↪ |
|             |     FAMILY "primary" (user_id, user_email, logoff_date, rowid)
    ↪ |
|             | )
    ↪ |
+-------------+-----------------------------------------------------------------+
    ↪
```

2.7.2.5 CREATE INDEX

CREATE INDEX 语句为表创建索引。

PRIMARY KEY 和 UNIQUE列将会自动创建索引。

所需权限

> 要求拥有表的 CREATE 权限。

创建索引

```
CREATE TABLE cust_info (
    id INT DEFAULT unique_rowid(),
    name STRING,
    age INT,
    region STRING,
    address JSONB
);
```

创建索引

```
CREATE INDEX ON cust_info (age);
```

创建二级索引

```
CREATE INDEX ON cust_info (name, region);
```

创建唯一索引

```
CREATE UNIQUE INDEX ON cust_info (name);
```

或者

HUBBLE

```
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);
```

```
\d cust_info  // show columns from cust_info
```

```
column_name|data_type|is_nullable|column_default|       indices         |
    ↪ is_hidden
-----------+---------+-----------+--------------+-----------------------+---------
    ↪
id          |INT8     |   true    |unique_rowid()|{cust_info_name_id_key}|  false
name        |STRING   |   true    |NULL          |{cust_info_name_id_key}|  false
age         |INT8     |   true    |NULL          |{}                     |  false
region      |STRING   |   true    |NULL          |{}                     |  false
address     |JSONB    |   true    |NULL          |{cust_info_address_idx}|  false
rowid       |INT8     |   false   |unique_rowid()|{primary,              |  true
            |         |           |              | cust_info_name_id_key,|
            |         |           |              | cust_info_address_idx}|
```

2.7.2.6 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，并且对视图引用的热和表具有SELECT权限

参数

参数	详情
view_name	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。
name_list	可选项，视图的逗号分隔的列名列表。如果指定，则将在 response 中使用这些名称，而不是AS select_stmt中指定的列。
as select_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。

```
CREATE VIEW general_view AS SELECT id, name FROM cust_info;
```

> **注意：**
>
> 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 general_view;
```

```
 table_name |          create_statement
------------+-------------------------------------------------
general_view|CREATE VIEW general_view ( ID, NAME ) AS SELECT
            |ID,
            |NAME
            |FROM
            |    defaultdb.PUBLIC.cust_info
```

2.7.2.7 CREATE SEQUENCE

所需权限

> 执行此操作需要用户拥有CREATE权限。

SEQUENCE 支持一下函数操作

- nextval('seq_name')
- currval('seq_name')
- lastval()
- setval('seq_name',value,is_called)

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,
  cre_date date
);
```

插入几条测试数据

HUBBLE

```
insert into user_table (phone_number, cre_date)
values
('13900010002','2018-10-01'),
('13654559807','2018-12-01'),
('18800001111','2016-10-01');
```

```
  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
```

> **注意**
>
> 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。

2.7.2.8 COMMENT ON

添加注释

数据库添加注释

```
COMMENT ON DATABASE database_0922 IS '这是测试数据库';

SHOW databases WITH COMMENT;
```

```
     database_name     |      comment
+----------------------+-------------------+
     aideng            |      NULL
     database_0903     |      NULL
     database_0922     |      这是测试数据库
     db2               |      NULL
```

表添加注释

```
COMMENT ON TABLE person IS '用户信息表';

SHOW tables WITH COMMENT;
```

```
 table_name | comment
+------------+---------+
  t         |
  t2        |
  v2        |
  view_test |
  person    | 人员表
  company   |
```

HUBBLE

```
  shop        |
  user_table  |
(8 rows)
```

字段添加注释

```
COMMENT ON COLUMN person.id IS '唯一id';

SHOW COLUMNS FROM person WITH COMMENT;
```

```
  column_name |  data_type   | is_nullable | column_default |
    ↪ generation_expression |          indices           | is_hidden | comment
+-------------+--------------+-------------+----------------+-----------------------+------
  ↪
  name        | STRING       |    true     | NULL           |
    ↪                        | {}                           |   false   | NULL
  comment     | VARCHAR(20)  |    true     | NULL           |
    ↪                        | {}                           |   false   | NULL
  phone       | STRING       |    true     | NULL           |
    ↪                        | {}                           |   false   | NULL
  money       | INT8         |    true     | NULL           |
    ↪                        | {}                           |   false   | NULL
  id          | INT8         |    false    | NULL           |
    ↪                        | {primary,person_email_key}   |   false   | 唯一id
(5 rows)
```

2.7.2.9 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 views。

- 禁止删除一个非空的数据库（RESTRICT）

  当一个数据库不是空的时候，RESTRICT会阻止数据库的删除。

```
DROP DATABASE hubble_database_0902 RESTRICT;
```

```
pq: database "hubbledatabase" is not empty and CASCADE was not specified
```

2.7.2.10 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。

```
SHOW tables FROM hubbletest;
DROP TABLE hubbletest.branches;
DROP TABLE hubbletest.orders CASCADE;
```

注意

CASCADE删除依赖于表的所有对象（例如约束和视图）。CASCADE不会列出它掉落的对象，因此应谨慎使用。

2.7.2.11 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。

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

```
ALTER TABLE t2 DROP COLUMN customer;
```

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

- sql_safe_updates会话变量必须设置为false

```
// 设置会话变量
SET sql_safe_updates=false;
```

注意：删除完成后请还原sql_safe_updates为true

- 删除时检查依赖：

```
ALTER TABLE t2 DROP COLUMN name RESTRICT;
```

```
  pq: cannot drop column "name" because view "v2" 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 t2 DROP COLUMN name CASCADE;
```

2.7.2.12 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。

查看索引

```
SHOW INDEX FROM person;
```

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 person@person_email_key;
```

```
pq: index "company_tel_key" is in use as unique constraint (use CASCADE if you
    ↪ really want to drop it)
```

如果一个表字段是外键类型时，删除索引时要加关键字 CASCADE。(该索引为默认索引格式，可直接删除)

```
DROP INDEX cust_info_name_region_idx CASCADE;
```

2.7.2.13 DROP VIEW

删除数据库中的view

所需权限

> 用户必须拥有删除视图的权限

```
SELECT * FROM information_schema.tables WHERE table_type = 'VIEW';
```

```
+---------------+------------------+--------------------+------------+---------+
    ↪
| TABLE_CATALOG |   TABLE_SCHEMA   |     TABLE_NAME     | TABLE_TYPE | VERSION
    ↪ |
+---------------+------------------+--------------------+------------+---------+
    ↪
| def           | bank             | user_accounts      | VIEW       |       1
    ↪ |
| def           | bank             | user_emails        | VIEW       |       1
    ↪ |
+---------------+------------------+--------------------+------------+---------+
    ↪
(2 rows)
```

```
DROP VIEW bank.user_accounts;
```

```
pq: cannot drop view "user_accounts" because view "user_emails" depends on it
```

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 bank.user_accounts CASCADE;
```

2.7.2.14 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

```
DROP SEQUENCE userid_seq;
```

如果有表依赖此序列不允许删除:

```
pq: cannot drop sequence userid_seq because other objects depend on it
```

2.7.2.15 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。

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。

```
SHOW CONSTRAINTS FROM orders;
```

```
+--------+--------------------------+-------------+-----------+----------------+
    ↪
| Table  |           Name           |    Type     | Column(s) |    Details
    ↪ |
+--------+--------------------------+-------------+-----------+----------------+
    ↪
| orders | fk_customer_ref_customers | FOREIGN KEY | customer  | customers.[id]
    ↪ |
| orders | primary                  | PRIMARY KEY | id        | NULL
    ↪ |
+--------+--------------------------+-------------+-----------+----------------+
    ↪
```

删除约束

```
ALTER TABLE orders DROP CONSTRAINT fk_customer_ref_customers;
```

```
SHOW CONSTRAINTS FROM orders;
```

```
+--------+---------+-------------+-----------+---------+
| Table  |  Name   |    Type     | Column(s) | Details |
+--------+---------+-------------+-----------+---------+
| orders | primary | PRIMARY KEY | id        | NULL    |
+--------+---------+-------------+-----------+---------+
```

删除唯一约束

> UNIQUE constraint 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

```
  table_name | constraint_name  | constraint_type |       details        |
    ↪ validated
+------------+------------------+-----------------+----------------------+----------
  ↪
  person     | person_email_key | UNIQUE          | UNIQUE (email ASC)   |
    ↪ true
  person     | primary          | PRIMARY KEY     | PRIMARY KEY (id ASC) |
    ↪ true
(2 rows)
```

```
DROP INDEX person@person_email_key CASCADE;
```

2.7.2.16 RENAME DATABASE

修改数据库名称

所需权限

> 要求拥有数据库管理员权限

> **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;
```

2.7.2.17 RENAME TABLE

修改表名称

所需权限

HUBBLE

要求拥有表管理员权限

```
ALTER TABLE t2 RENAME TO sys_role;
```

2.7.2.18 RENAME COLUMN

修改表的字段名称

所需权限

要求拥有表的 CREATE 权限

```
SHOW COLUMNS FROM user_tab;
ALTER TABLE user_tab RENAME COLUMN phone_number TO tel_no;
```

2.7.2.19 RENAME SEQUENCE

修改数据库的 SEQUENCE 名称

所需权限

要求拥有库的 CREATE 权限

查看当前库中的 SEQUENCE:

```
SELECT * FROM information_schema.sequences;
```

修改 SEQUENCE:

```
ALTER SEQUENCE database_0922.userid_seq RENAME TO database_0922.uid_seq;
```

注意:

需要确认是否有表依赖与此 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。

2.7.2.20 RENAME CONSTRAINT

修改表的约束名称

所需权限

要求拥有表的 CREATE 权限

查询表的约束:

```
SHOW CONSTRAINTS FROM table_with_index;
```

修改表的约束名称:

```
ALTER TABLE table_with_index RENAME CONSTRAINT table_with_index_user_email_key
    ↪ TO unique_email;
```

2.7.2.21 SHOW DATABASES

查看集群包含的所有数据库

```
SHOW DATABASES;
```

2.7.2.22 SHOW TABLES

查看一个 schema 或者数据库中的表和视图

```
SHOW TABLES;
SHOW TABLES FROM my_hubble_database;
```

2.7.2.23 SHOE COLUMNS

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 my_hubble_table;
```

2.7.2.24 SHOW INDEX

查看表的索引信息

```
SHOW INDEX FROM my_hubble_table;
```

2.7.2.25 SHOW SCHEMAS

查看一个数据库中所有的 schemas 信息

```
SHOW SCHEMAS;
```

2.7.2.26 SHOW SEQUENCE

显示当前可用的 SEQUENCE

```
SHOW SEQUENCES;
```

```
  sequence_name
+----------------+
  userid_seq
```

2.7.2.27 SHOW CONSTRAINT

显示表的约束情况

```
SHOW CONSTRAINT from company;
```

HUBBLE

```
  table_name | constraint_name | constraint_type |        details         |
     ↪ validated
+------------+-----------------+-----------------+------------------------+-----------+
   ↪
  company    | primary         | PRIMARY KEY     | PRIMARY KEY (id ASC) |   true
(1 row)
```

2.7.2.28 SHOW PARTITIONS

检索表分区

```
SHOW PARTITIONS FROM { TABLE <table> | INDEX <index> | DATABASE <database> };
```

2.7.2.29 SHOW RANGES

查看 range 分区情况

```
SHOW RANGES FROM TABLE customers_by_range;
```

2.7.2.30 SHOW CREATE TABLE

显示创建表的语句

```
SHOW CREATE TABLE my_hubble_table;
```

2.7.2.31 SHOW CREATE VIEW

显示创建 view 的 create view 语句

```
SHOW CREATE VIEW my_hubble_viewname;
```

2.7.2.32 SHOW CREATE SEQUENCE

显示特定序列的创建语句

```
SHOW CREATE <sequencesName>;
```

结果

```
table_name|                                        create_statement
----------+-------------------------------------------------------------------------------
   ↪
userid_seq|CREATE SEQUENCE userid_seq MINVALUE 1 MAXVALUE 9223372036854775807
   ↪ INCREMENT 1 START 1
```

2.7.3 DML

2.7.3.1 SELECT

SELECT语句用于读取和处理现有数据。

所需权限

HUBBLE

要求用户拥有表的SELECT权限。

检索所有列

```
SELECT * FROM accounts;
```

```
    id     |    name    | balance | flag  | is_disabled | is_dis
+---------+-----------+---------+-------+-------------+--------+
         6 | zack       |   66666 | false | true        |  NULL
         8 | zhangsan   | 1001.45 | NULL  | NULL        |  NULL
        99 | feina      |     555 | false | false       |  NULL
       123 | xuejs      |  123.45 | NULL  | NULL        |  NULL
       321 | alian      |      76 | false | true        |  NULL
       876 | jana       |      76 | false | true        |  NULL
      1234 | lyp        |   88888 | NULL  | NULL        |  NULL
     98790 | ff         |   66666 | false | true        |  true
    123456 | xuejs      |  523.45 | NULL  | NULL        |  NULL
   6843484 | Jack       |   88888 | false | true        |  NULL
(10 rows)
```

检索特定列

```
SELECT name FROM accounts;
```

```
    name
+-----------+
  zhangsan
  xuejs
  lyp
  xuejs
(4 rows)
```

过滤

- 单个条件过滤

```
  SELECT name FROM accounts WHERE name='zhangsan';
```

```
     name
  +-----------+
    zhangsan
 (1 row)
```

- 多个条件过滤

```
  SELECT name FROM accounts WHERE name='zhangsan' AND balanc1e='1001.45';
```

```
     name
  +-----------+
```

HUBBLE

```
  zhangsan
(1 row)
```

- 列表过滤值

```
SELECT name
  FROM accounts
  WHERE balance
  IN ('1001.45', '523.45');
```

```
    name
+----------+
  zhangsan
  xuejs
(3 rows)
```

DISTINCT

```
select DISTINCT name FROM accounts;
```

```
   name
+----------+
  zhangsan
  xuejs
  lyp
(3 rows)
```

ALL

```
select DISTINCT name FROM accounts;
```

```
    name
+----------+
  zhangsan
  xuejs
  lyp
  xuejs
(4 rows)
```

GROUP BY 聚合函数

```
SELECT name, sum(balance)
    FROM accounts
    GROUP BY name;
```

```
    name    |   sum
+----------+---------+
  feina     |     555
  alian     |      76
```

HUBBLE

```
  Jack      |   88888
  zack      |   66666
  zhangsan  | 1001.45
  xuejs     |  646.90
  jana      |      76
  lyp       |   88888
  ff        |   66666
(9 rows)
```

HAVING 分组过滤

```
SELECT name, SUM(balance)
  FROM accounts
  GROUP BY name
  HAVING name='xuejs';
```

```
  name   |  sum
+-------+--------+
  xuejs | 646.90
(1 row)
```

2.7.3.2 INSERT

- 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 IDs 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

插入单行

```
INSERT INTO accounts (id, name, balance, flag, is_disabled) VALUES
  (6, 'zack', 66666, 'false', 'true');
```

```
SELECT * FROM accounts WHERE id=6;
```

```
root@poc-hubble01:15432/xuejs> SELECT * FROM accounts;
    id    |    name    | balance | flag  | is_disabled
+--------+----------+---------+-------+-------------+
       6 | zack       |   66666 | false | true
       8 | zhangsan   | 1001.45 | NULL  | NULL
     123 | xuejs      |  123.45 | NULL  | NULL
    1234 | lyp        |  123.45 | NULL  | NULL
  123456 | xuejs      |  523.45 | NULL  | NULL
(5 rows)
```

插入多行

```
INSERT INTO accounts (id, name, balance, flag, is_disabled) VALUES
  (876, 'sinal', 76, 'false', 'true'), (99, 'feina', 55, 'false', 'false');
```

```
SELECT * FROM accounts WHERE id IN (876, 99);
```

```
  id  | name  | balance | flag  | is_disabled
+-----+-------+---------+-------+-------------+
   99 | feina |     555 | false | false
  876 | jana  |      76 | false | true
(2 rows)
```

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 (6, 'zack', 66666, 'false', 'true')
  ON CONFLICT (id, name)
  DO UPDATE SET balancce=excluded.balance;
```

使用现有值更新

```
INSERT INTO accounts (id, name,balance, flag, is_disabled)
  VALUES (99, 'feina', 555, 'false', 'false')
  ON CONFLICT (id, name)
  DO UPDATE SET balance=accounts.balance+excluded.balance;
```

```
SELECT * FROM accounts WHERE id=99;
```

```
  id | name  | balance | flag  | is_disabled
+----+-------+---------+-------+-------------+
  99 | feina |     555 | false | false
(1 row)
```

ON CONFLICT语句下使用WHERE条件

```
INSERT INTO accounts (id, name,balance, flag, is_disabled)
  VALUES (99, 'feina', 555, 'false', 'false')
  ON CONFLICT (id, name)
  DO UPDATE SET balance=excluded.balance
  WHERE excluded.balance < accounts.balance;
```

```
SELECT * FROM accounts WHERE id=99;
```

```
  id | name  | balance | flag  | is_disabled
+----+-------+---------+-------+-------------+
  99 | feina |     555 | false | false
(1 row)
```

HUBBLE

ON CONFLICT 不更新值

- 当约束冲突时，不操作

```
INSERT INTO accounts (id, name, balance, flag, is_disabled) VALUES
  (6, 'zack', 66666, 'false', 'true');
```

```
pq: duplicate key value (id,name)=(6,'zack') violates unique constraint "
    ↪ primary"
```

```
INSERT INTO accounts (id, name, balance, flag, is_disabled)
  VALLUES (6, 'zack', 66666, 'false', 'true')
  ON CONFLICT (id, name)
  DO NOTHING;
```

- 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', 'true'), (99, 'feina', 555, 'false', '
     ↪ false')
  ON CONFLICT (id, name)
  DO NOTHING;
```

2.7.3.3 DELETE

DELETE 语句删除一个表中行的数据

> **注意**
>
> 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

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 tablename WHERE colname < 12333 RETURNING * ;
```

返回特定列的数据

```
DELETE FROM tablename WHERE colname < 12333 RETURNING colnameb,colnamec;
```

2.7.3.4 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。

```
UPDATE accounts SET balance=1234567890 WHERE ID=99;
```

```
select * from accounts WHERE id=99;
```

```
    id     |    name    |  balance    | flag  | is_disabled | is_dis
+---------+----------+------------+-------+-------------+--------+
       99 | feina     | 1234567890 | false | false       |  NULL
(1 rows)
```

使用SELECT语句更新

```
UPDATE accounts
  SET (name, balance)=
  (SELECT name, balance FROM accounts WHERE id=123)
  WHERE id=8;
```

```
select * from accounts WHERE id=8;
```

```
    id     | name   |  balance    | flag  | is_disabled | is_dis
+---------+-------+------------+-------+-------------+--------+
        8 | xuejs |     123.45 | NULL  | NULL        |  NULL
(1 rows)
```

更新后返回值

```
UPDATE accounts SET balance=83468 WHERE name='ff' RETURNING id;
```

```
   id
+-------+
  98790
(1 row)
```

2.7.3.5 TRUNCATE

TRUNCATE语句用于从表中删除所有行。

所需权限

用户需要拥有表的DROP权限

清空一张客户表

```
TRUVCATE dates
```

```
select * from dates;
```

HUBBLE

```
  a | b
+---+---+
(0 rows)
```

对于含有外键的表，需要使用CASCADE关键字删除数据。

```
TRUNCATE papers CASCADE;
```

```
select * from papers
```

```
  id | paper_key | paper_value | paper_dt
+----+-----------+-------------+----------+
(0 rows)
```

2.7.3.6 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。在指定值不违反唯一性约束的情况下，它会插入行，在值违反唯一性榆树的情况下，他会更新。

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。

示例数据：

```
SELECT * FROM customer;
```

```
          id          | name | region |         regis_date
+---------------------+------+--------+----------------------------+
  705266450755518465 | 李三 | 朝阳   | 2020-01-22 00:00:00+00:00
  705266450755649537 | 张三 | 海淀   | 2020-01-23 00:00:00+00:00
  705266450755682305 | 王丽 | 通州   | 2020-01-20 00:00:00+00:00
  705274504013611009 | 马五 | 重庆   | 2017-10-01 00:00:00+00:00
(4 rows)
```

```
INSERT INTO customer (name, region, regis_date)
  VALUES ('王艳', '朝阳', '2020-01-22');
```

使用UPSERT语句操作表数据：

当唯一性冲突不在主键中的列上时，UPSERT将不会更新。

上例中id列是主键，但是name列具有UPSERT约束。下面插入name='马超'与非PRIMARY的UNIQUE字段name值冲突，UPSERT将不会生效。

```
UPSERT INTO customer (name, region,regis_date)
  VALUES ('马五','安徽','2017-10-01');
```

```
pq: duplicate key value (name)=(e'\U00009A6C\U00004E94') violates unique
    ↪ constraint "customer_name_key"
```

由于name字段创建了唯一性约束，所以更新失败

```
SELECT * FROM customer WHERE name LIKE '马%';
```

```
          id          | name | region |        regis_date
+---------------------+------+--------+---------------------------+
  705274504013611009 | 马五 | 重庆   | 2017-10-01 00:00:00+00:00
(1 row)
```

> **注意：**
>
> 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。

2.7.4 TRANSACTION

事务的优先级分为LOW 、NORMAL 和 HIGH三种。

事务模式分为只读（READ ONLY）和读写（READ WRITE）模式两种。

目前仅支持SERIALIZABLE隔离。

事务流程

流程	说明	执行
BEGIN	BEGIN用于启动一个事务，事务成功则执行它包含的所有语句，若失败则不执行。权限取决于事务中每个语句需要的用户权限。你也可以使用BEGIN/BEGIN TRANSACTION/START TRANSACTION开启一个事务，他们是等效的	BEGIN;
SET TRANSACTION	SET TRANSACTION语句可以设定事务的优先级请在执行操作数据库的第一个语句之前设置事务优先级。	设定事务优先级： BEGIN;SET TRANSACTION PRIORITY LOW;
SAVEPOINT	SAVEPOINT语句用于注释在当前事务中建立保存点。事务结束后，所有保存点都自动释放。session 变量force_savepoint_restart设置为true时，可以自定义保存点名称，默认只能使用hubble_restart。	SAVEPOINT trans_restart;
RELEASE SAVEPOINT	如果事务中的语句生成了任何非重试错误，则RELEASE SAVEPOINT等效于ROLLBACK，将中止事务并丢弃事务语句中的所有更新。声明SAVEPOINT后，使用RELEASE SAVEPOINT提交事务，然后使用COMMIT为下一个事务准备连接。	

流程	说明	执行
COMMIT	COMMIT语句用于提交当前事务。使用高级客户端事务重试时，在发出RELEASE SAVEPOINT而不是COMMIT时，将提交在SAVEPOINT之后发出的语句。但是，您仍然必须发出COMMIT语句以清除下一个事务的连接。对于不可重试的事务，如果事务中的语句生成任何错误，则COMMIT等效于ROLLBACK，后者将中止事务并丢弃其语句进行的所有更新。	
ROLLBACK	用于中止当前的事务，放弃事务中所有的更新语句。	放弃事务操作: BEGIN;INSERT INTO customers1 (name, ↪ region,regis_date)VALUES ('黎明',' ↪ lishui','2018-10-01');ROLLBACK; ↪ COMMIT;也可以使 用ROLLBACK TO SAVEPOINT 语句，回退到保存点，进行事务的重新执行。 ROLLBACK TO SAVEPOINT trans_restart;

示例:

```
BEGIN;
SET TRANSACTION PRIORITY HIGH;
SAVEPOINT hubble_restart;
-- DML语句中有执行异常，该事务无法执行;下面2个DML语句不满足INSERT或者UPDATE的执
   ↪ 行条件，无法执行成功;事务回退
UPDATE customer SET region = '浙江' WHERE name = '网田';
INSERT INTO customer (name,region,regis_date) VALUES ('郝安', '南宁', '
   ↪ 2020-08-08');
RELEASE SAVEPOINT hubble_restart;
COMMIT;
END;
```

```
SELECT * FROM customer;
```

```
          id          |  name  | region |         regis_date
+---------------------+--------+--------+----------------------------+
  705266450755518465 | 李三   | 湖北   | 2020-01-22 00:00:00+00:00
  705266450755682305 | 王丽   | 通州   | 2020-01-20 00:00:00+00:00
  705269989533876225 | 王艳   | 朝阳   | 2020-01-22 00:00:00+00:00
  705274504013611009 | 马五   | 安徽   | 2017-10-01 00:00:00+00:00
  705337581353402369 | 张鹏   | 上海   | 2020-02-02 00:00:00+00:00
  705337753947865089 | 网田   | 浙江   | 2020-02-02 00:00:00+00:00
  705339388724019201 | 安浩   | 江苏   | 2020-06-06 00:00:00+00:00
  705339938462203905 | 郝安   | 南宁   | 2020-08-08 00:00:00+00:00
(8 rows)
```

HUBBLE

```
BEGIN;
SET TRANSACTION PRIORITY HIGH;
SAVEPOINT hubble_restart;
INSERT INTO customers1 (NAME,region,regis_date) VALUES ('李安', '贵阳', '
    ↪ 2019-09-09');
RELEASE SAVEPOINT hubble_restart;
COMMIT;
```

```
SELECT * FROM customer;
```

```
          id          |  name  | region |         regis_date
----------------------+--------+--------+----------------------------
  705266450755518465 | 李三   | 湖北   | 2020-01-22 00:00:00+00:00
  705266450755682305 | 王丽   | 通州   | 2020-01-20 00:00:00+00:00
  705269989533876225 | 王艳   | 朝阳   | 2020-01-22 00:00:00+00:00
  705274504013611009 | 马五   | 安徽   | 2017-10-01 00:00:00+00:00
  705337581353402369 | 张鹏   | 上海   | 2020-02-02 00:00:00+00:00
  705337753947865089 | 网田   | 浙江   | 2020-02-02 00:00:00+00:00
  705339388724019201 | 安浩   | 江苏   | 2020-06-06 00:00:00+00:00
  705339938462203905 | 郝安   | 南宁   | 2020-08-08 00:00:00+00:00
  705350799508963329 | 李安   | 贵阳   | 2019-09-09 00:00:00+00:00
(8 rows)
```

2.7.5 SYSTEM

2.7.5.1 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。

你也可以使用SHOW CLUSTER SESSIONS。

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';
```

2.7.5.2 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';
```

HUBBLE

2.7.5.3 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。

2.7.5.4 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)
```

2.7.5.5 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	说明	默认值	是否可用 SHOW 查看
application_name			
database			
default_transaction_isolation			
default_transaction_read_only			
sql_safe_updates			

变量名称	说明	默认值	是否可用 SHOW 查看
search_path			
server_version_num			
timezone			
tracing	跟踪记录状态	off	Yes
transaction_isolation	当前事务的隔离级别	SERIALIZABLE	Yes
transaction_priority			
transaction_read_only	访问模式		
transaction_status		NoTxn	Yes
client_encoding	客户端编码	N/A	No
client_min_messages			No
extra_floatdigits			No
standard_conforming_strings			No

指定当前数据库：

```
SET database=database_0922;
```

> **关于时区**
>
> 默认时区为 UTC，如果需要修改客户端显示的时区，可以使用SET timezone='Asia/Shanghai';，设置后显示的时间即为中国时间。

2.7.5.6 SHOW CLUSTER SETTING

查看一个或多个clister setting的值，也可以通过SET CLUSTER SETTING进行配置。

所需权限

> 查看所有Cluster Setting的值，需要 admin 权限

```
SHOW ALL CLUSTER SETTINGS;
```

2.7.5.7 SHOW CLUSTER SETTING

使用SET CLUSTER SETTING语句可以修改集群中变量的默认值。

所需权限

> 需要用户拥有 admin 权限

```
SET CLUSTER SETTING <variable>=<value>;
```

例如：

```
SET CLUSTER SETTING trace.debug.enable = true;
```

2.7.5.8 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;
```

2.7.6 JOBS

2.7.6.1 EXPLAIN

语句返回指定语句的查询计划，用户可以使用 explain 语句优化查询。

```
EXPLAIN SELECT * FROM customer;
```

```
  tree |    field     |   description
+------+--------------+------------------+
       | distributed  | true
       | vectorized   | false
  scan |              |
       | table        | customer@primary
       | spans        | ALL
(5 rows)
```

2.7.6.2 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。
>
> 建议导出得治为每个节点都可以访问的公共文件服务。

HUBBLE

导出数据到本地，数据存储区在数据目录的 extern 目录下：

```
EXPORT INTO
CSV 'nodelocal:///customers'
WITH DELIMITER ='|'
FROM
    SELECT * FROM users;
```

```
  filename | rows | bytes
+----------+------+-------+
  n5.0.csv |    3 |   414
(1 row)
```

> **说明**
>
> Tip：n1.0.csv 格式说明
>
> n1：节点 id，即文件保存在 n1 节点
>
> 0：文件序号，从 0 起
>
> csv：文件格式
>
> 例如：n3.12.csv 为EXPORT任务在 n3 节点上生成的编号 12 保存为 csv 格式的数据

查看本地目录：

```
# 当前在n1节点
cat ${HUBBLE_HOME}/data1/hubble_data/extern/customers/n1.0.csv
```

```
535970470514294785|李三|123@qq.com|chaoyang|2018-10-01
535970470514360321|王二|456@qq.com|haidian|2018-12-01
535970470514524161|钱途|666@qq.com|harbin|2017-10-01
535970470514393089|赵四|789@qq.com|pudong|2016-10-01
535970470514556929|贾六|888@qq.com|shenyang|2017-10-01
535970470514458625|马超|345@qq.com|tianhe|2016-10-01
535970470514589697|刘刘|777@qq.com|tianjin|2017-10-01
535970470514425857|王石|234@qq.com|xuhui|2016-10-01
535970470514491393|张良|457@qq.com|yuexiu|2016-10-01
535970470514622465|胡杨|119@qq.com|zhengzhou|2017-10-01
```

2.7.6.3 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

使用 http 搭建文件服务，必须使各节点可以访问到待导入文件

```
IMPORT TABLE customers2 (
    ID INT,
    NAME STRING,
    email STRING,
    region STRING,
    regis_date DATE,
    PRIMARY KEY ( ID )
) CSV DATA ( 'http://IP:Port/customers/n1.0.csv' )
WITH
    DELIMITER = '|'
;
```

```
        job_id       |  status   | fraction_completed | rows | index_entries |
           ↪ system_records | bytes
+--------------------+-----------+--------------------+------+---------------+-----------
   ↪
  705884786919309313 | succeeded |                  1 |   13 |             0 |
     ↪                0 |   507
(1 row)
```

2.7.6.4 IMPORT INTO

可以使用IMPORT INTO语句，导入数据到已经存在的表中。

说明：本文档适用于从 hubble 到 hubble 的 csv 导出导入

步骤：从 tt_image1 到 img 从 hubble 中导出数据

```
EXPORT INTO
CSV 'nodelocal:///customers'
WITH DELIMITER ='|'
FROM
SELECT * FROM tt_image1;
```

返回结果：

```
  filename | rows |    bytes
+----------+------+-----------+
  n1.0.csv |   59 | 477301967
```

导入一：img 表存在的情况，将数据导入到 img

```
IMPORT into img ( a , B
) CSV DATA ( 'nodelocal:///customers/n4.0.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
```

导入二: img 表不存在的情况

```
IMPORT table  img ( a int  , B  bytes
) CSV DATA ( 'nodelocal:///customers/n1.0.csv' )
WITH
    DELIMITER = '|'
;
```

返回结果:

```
       job_id       |  status   | fraction_completed | rows | index_entries |
          ↪ system_records |   bytes
+--------------------+-----------+--------------------+------+---------------+-----------
   ↪
  739024174094942209 | succeeded |                  1 |   59 |             0 |
     ↪              0 | 238652345
```

查看数据量

```
select count(1) from img;
```

```
  count
+-------+
     59
(1 row)
```

所需权限

> 需要用户拥有admin权限

建表:

```
CREATE TABLE customers_3 (
    id INT,
```

```
    name STRING,
    email STRING,
    region STRING,
    regis_date DATE,
    PRIMARY KEY (id)
);
```

导入数据:

> **注意**
>
> 使用 http 搭建文件服务，必须使各节点可以访问到待导入文件

```
IMPORT INTO customers_3 (id, name, email, region, regis_date)
CSV DATA (
    'http://IP:Port/customers/n1.0.csv'
) WITH delimiter = '|' ;
```

2.7.6.5 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;
```

2.7.6.6 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');
```

2.7.6.7 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');
```

2.7.6.8 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');
```

2.8 窗口函数

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。

2.8.1 窗口的定义

窗口使用 over 或 window 关键字定义

2.8.2 窗口函数机制

窗口功能通过以下方式工作:

1. 使用选择查询创建“虚拟表”。
2. 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。
3. 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 DISTINCT(city),
              SUM(revenue) OVER () AS total_revenue,
              SUM(revenue) OVER (PARTITION BY city) AS city_revenue
             FROM rides
         ORDER BY city_revenue DESC;
```

其操作可以描述如下 (此处列出的编号步骤对应于下图中的编号):

1. 外部使用 SELECT DISTINCT(city)... FROM RIDES 创建一个用于窗口函数操作的虚拟表
2. 窗口函数 SUM(revenue)OVER () 在包含查询输出的所有行的窗口框架上进行操作。
3. 窗口函数 SUM(revenue)OVER (PARTITION BY city)轮流操作多个窗框; 每个框架包含不同城市分区的revenue列

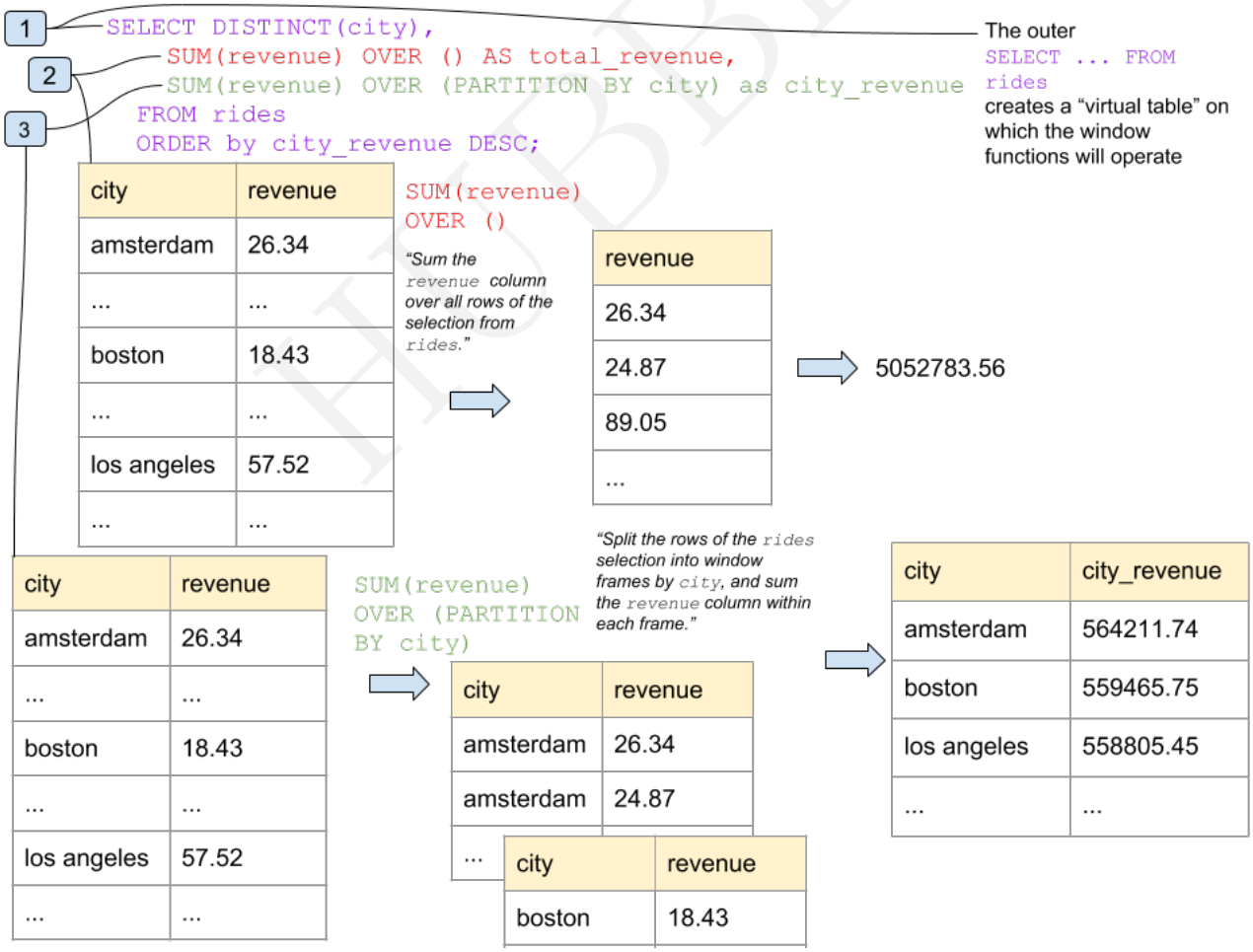


图 1: window-functions.png

2.8.3 提示说明:

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。

- 常见的开窗函数

rank() over()、dense_rank() over()、row_number() over() 比较

```
> 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);
```

```
> 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
```

```
> 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
```

```
> 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
```

2.9 主键

2.9.1 注意事项

> 在表数据迁移到 HUBBLE 过程中，如果目标表有主键，一定要在建表过程中维护主键后进行迁移，防止整条语句当成主键。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，之后无法通过 alter 语句建立主键。

2.9.2 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。

2.9.3 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，再说联合主键。

2.9.4 示例

单一主键：单个字段作为主键。

示例 1：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)
```

```
);
```

复合主键：2 个及其以上字段保证数据唯一。

示例 2：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)
);
```

联合主键：用 2 个字段 (或者多个字段) 来确定一条记录; 这 2 个字段都不是唯一的，2 个字段可以分别重复。一个简单的例子

```
> create table  paper(
     A   varchar(50)   not   null   primary      key,
     B   char(10)      not   null   constraint   unique_author_id   unique
);
```

字段 A 跟字段 B 组成联合主键

字段 A 跟字段 B 的数据可以完全相同，联合就在于 A 跟 B 形成的联合主键是唯一的。

下例字段 A 数据是 a，字段数据也是 a，联合主键其实是 aa，这个 aa 是唯一值，绝对不充许再出现 aa 这个唯一值。(这就是多对多关系)

字段 A 数据字段 B 数据

a　　a

b　　b

A 与 B 的联合主键值最多也就是

aa

ab

ba

bb

3　运维

3.1　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　备份语法

```
BACKUP <table_pattern [,table_pattern]>||<DATABASE> db_name TO destination
[AS OF SYSTEM TIME <TIMESTAMP>
[INCREMENTAL FROM <full_backup_location> ]
```

HUBBLE

3.1.1.1 参数说明

- destination 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
- INCREMENTAL FROM <full_backup_location> 增量备份

> **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。

3.1.2 备份数据路径 destination 说明

备份路径拥有 2 种模式，http 和 NFS 模式，推荐使用NFS稳定性和速度更优
每个备份的路径必须是唯一的，备份的目的地/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
NFS/Local	nodelocal	节点 ID 或为空	N/A	nodelocal:///path/mydatest

3.1.3 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。

- 备份整个数据库

```
backup database zl to 'nodelocal:///k2/f3_1';
```

- 备份某几张表

```
backup zl.tt,zl.f2 to 'nodelocal:///zl_t_bak'
```

3.1.4 增量备份

- 增量备份整个数据库

```
backup database zl to 'nodelocal:///k2/f3_3' incremental FROM 'nodelocal:///k2/
    ↪ f3_1';
```

- 增量备份某几张表

```
backup zl.tt,zl.f2 to 'nodelocal:///zl_t_bak_2' incremental from 'nodelocal:///
    ↪ zl_t_bak';
```

HUBBLE

3.1.5 指定时间戳备份

```
backup database zl to 'nodelocal:///k2/f3_5' AS OF SYSTEM TIME '-20s';  --备份
    ↪ 20s之前的数据
```

> **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

3.2 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。

3.2.1 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

3.2.2 备份数据路径

每个备份的路径必须是唯一的，备份的目的地/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:///path/mydatest,nodelocal://2/path/mydatest`

3.2.3 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
`into_db`	如果希望将表或视图还原到数据库中，而不是它原来所在的数据库中，则可以更改目标数据库。如果您希望恢复当前存在的表，但又不想删除它，那么这是非常有用的。	WITH into_db = 'newdb'
`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

3.2.4 示例

3.2.4.1 恢复单个表

```
RESTORE icheck.customers FROM 'http://database-icheck-2019-01-11-weekly';
```

3.2.4.2 恢复多个表

```
RESTORE icheck.customers, icheck.accounts FROM 'http://database-icheck
    ↪ -2019-01-11-weekly';
```

3.2.4.3 恢复整个数据库

```
RESTORE DATABASE icheck FROM 'http://database-icheck-2019-01-11-weekly';
```

3.2.4.4 指定恢复时间点

```
RESTORE icheck.customers FROM 'http://database-icheck-2019-01-11-weekly'
AS OF SYSTEM TIME '2019-01-11 10:00:00';
```

3.2.4.5 从增量备份恢复数据

```
RESTORE icheck.customers
FROM 'http://database-icheck-2019-01-11-weekly', 'http://database-icheck
    ↪ -2019-01-12-daily', 'http://database-icheck-2019-01-13-daily';
```

HUBBLE

3.2.4.6 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 icheck.customers
FROM 'http://database-icheck-2019-01-11-weekly'
WITH into_db = 'newdb';
```

3.2.4.7 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 icheck.accounts
FROM 'http://database-icheck-2019-01-11-weekly'
WITH skip_missing_foreign_keys;
```

3.2.4.8 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 'http://database-icheck-2019-01-11-weekly'
WITH into_db = 'newdb';
```

```
INSERT INTO system.users SELECT * FROM newdb.users;
```

```
DROP TABLE newdb.users;
```

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	概况	查看重要的 SQL 性能，复制和存储指标。监控-> 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-> 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-> 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-> 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-> 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-> 仪表盘：复制
	分布式	查看 batches，RPCs，kv 事务等相关指标。监控-> 仪表盘：分布式

HUBBLE

导航栏	区域	描述
	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-> 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-> 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-> 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-> 汇总
	事件	查看最近的集群事件列表。监控-> 事件
数据库		查看有关集群中系统和用户数据库的详细信息。
语句		识别频繁执行或高延迟的 SQL 语句
任务		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	条数据
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 集群概况页面

Admin UI 的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 集群概况面板

图 2: 输入图片说明

集群概况面板提供以下指标：

HUBBLE

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。
分片状态	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。

图 3: 输入图片说明

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 汇总面板

集群指标仪表板显示关键指标的汇总面板。要查看汇总面板，请单击左侧的监控。

汇总面板提供以下指标（不一定全部显示）：

图 4: 输入图片说明

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。

3.3.5 事件面板

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。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

图 5: 输入图片说明

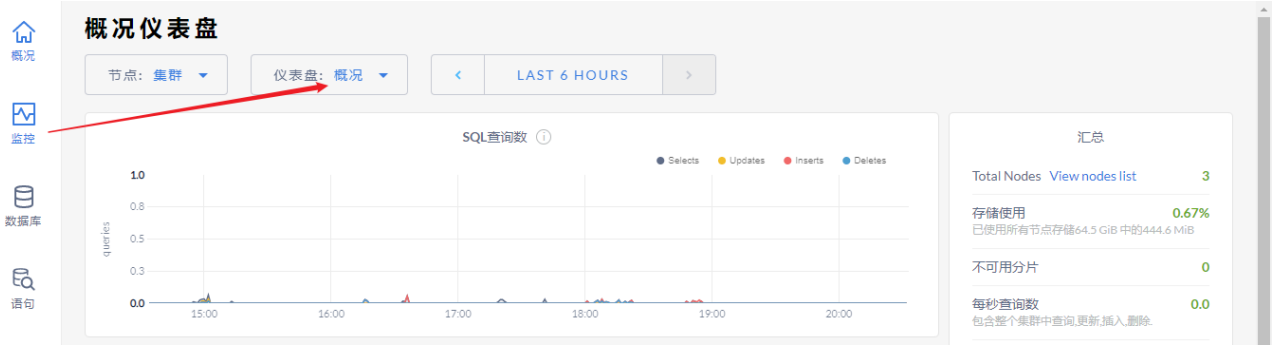


图 6: 输入图片说明

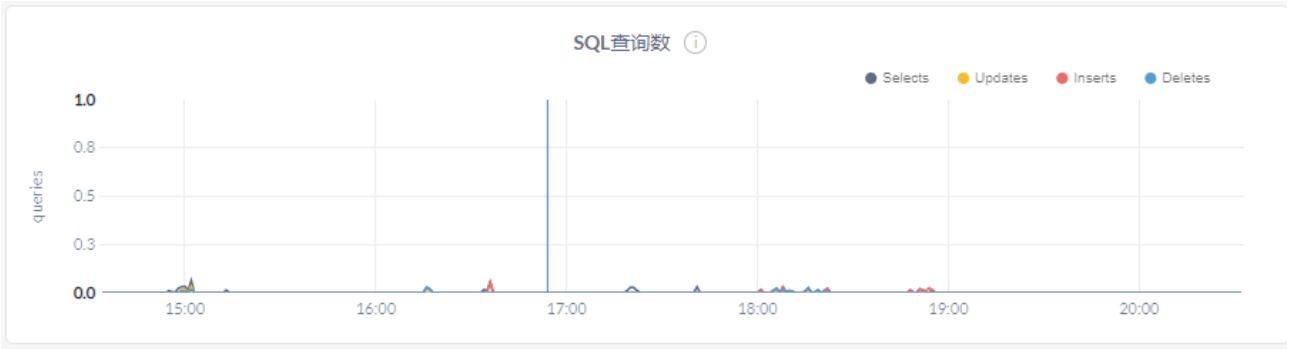


图 7: 输入图片说明

- 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。

3.3.7 运行延迟：SQL，99%

图 8: 输入图片说明

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。

3.3.7.1 每节点的副本数

图 9: 输入图片说明

- 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。

3.3.7.2 容量

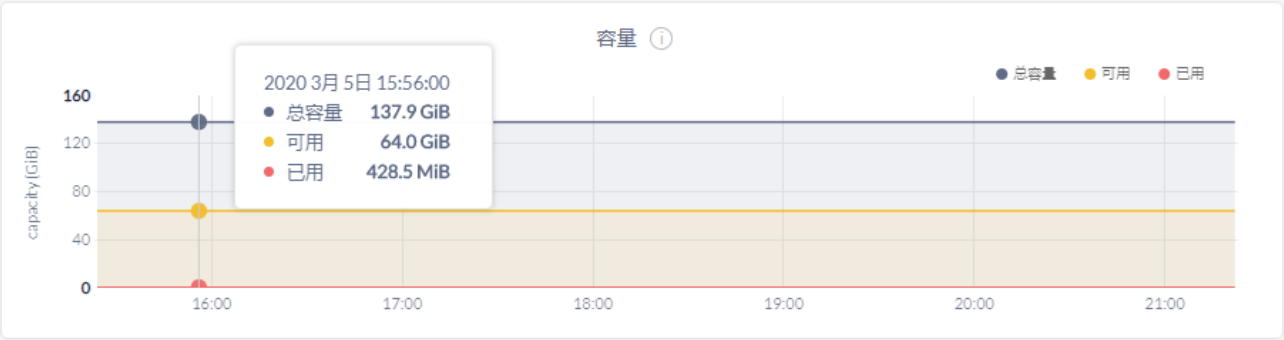


图 10: 输入图片说明

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。

3.3.8 硬件概览

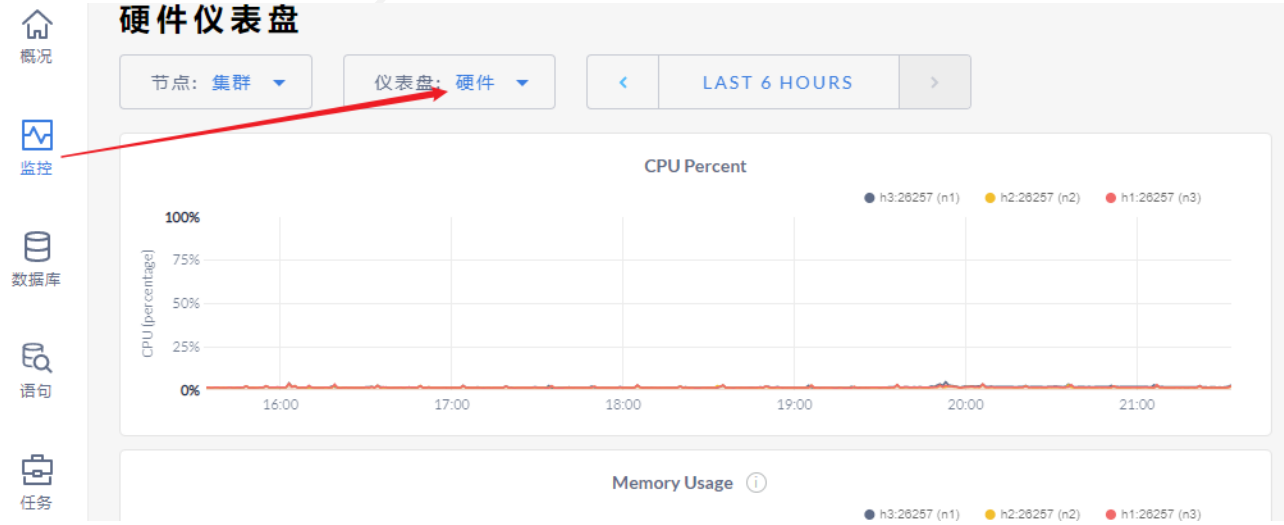


图 11: 输入图片说明

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，然后选择仪表板 > 硬件。

硬件仪表板显示以下时间序列图：

3.3.8.1 CPU 使用率

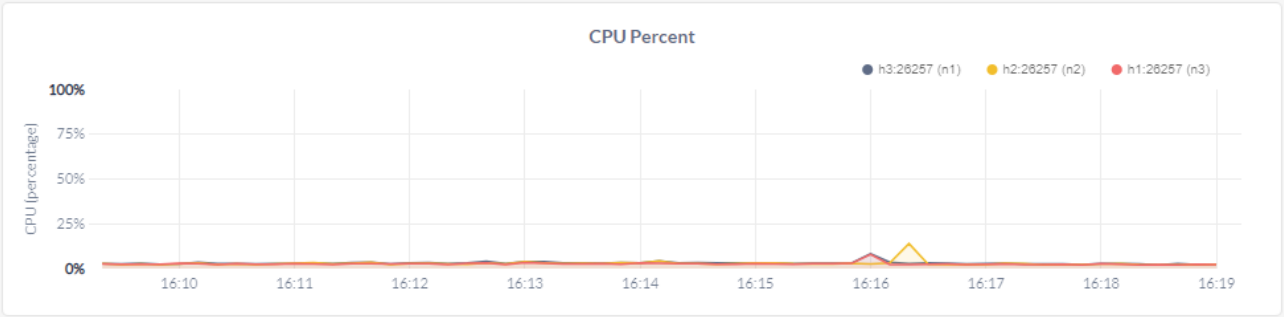


图 12: 输入图片说明

- 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。

3.3.8.2 内存使用情况

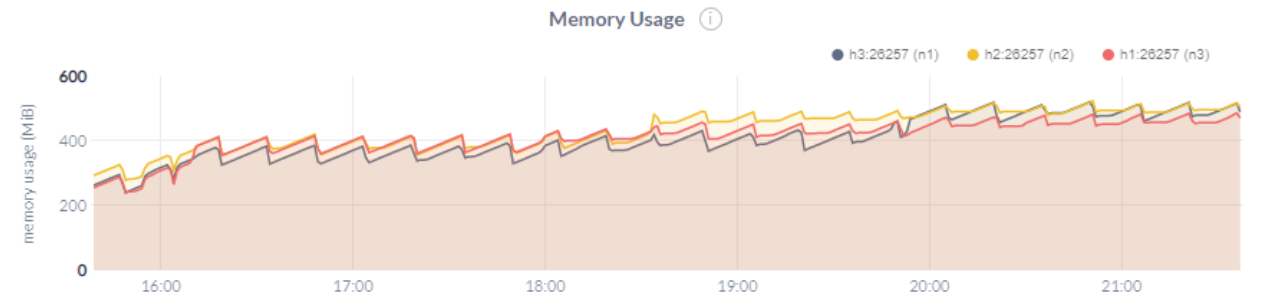


图 13: 输入图片说明

- 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。

3.3.8.3 磁盘读取字节

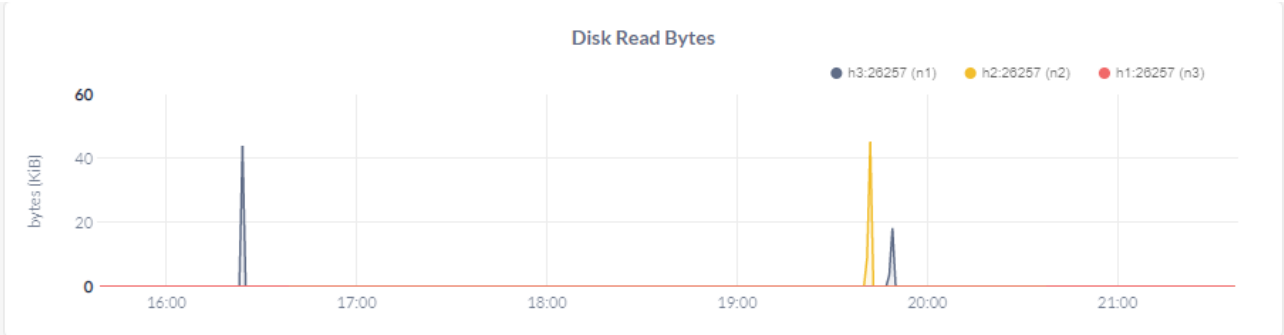


图 14: 输入图片说明

- 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。

3.3.8.4 磁盘写入字节

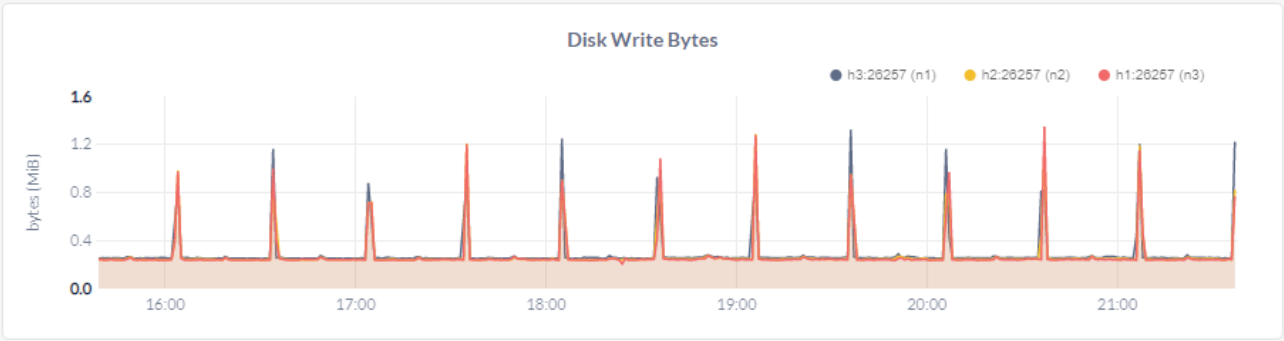


图 15: 输入图片说明

- 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

3.3.8.5 磁盘读取操作

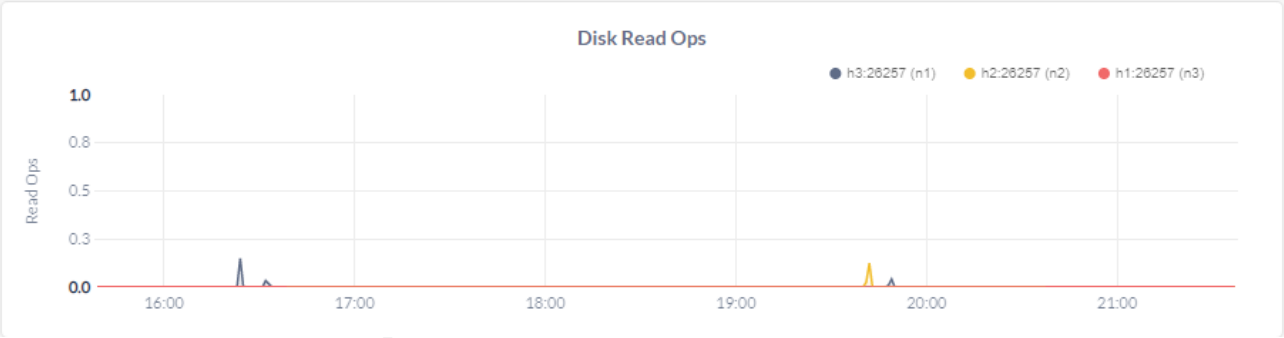


图 16: 输入图片说明

- 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

3.3.8.6 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

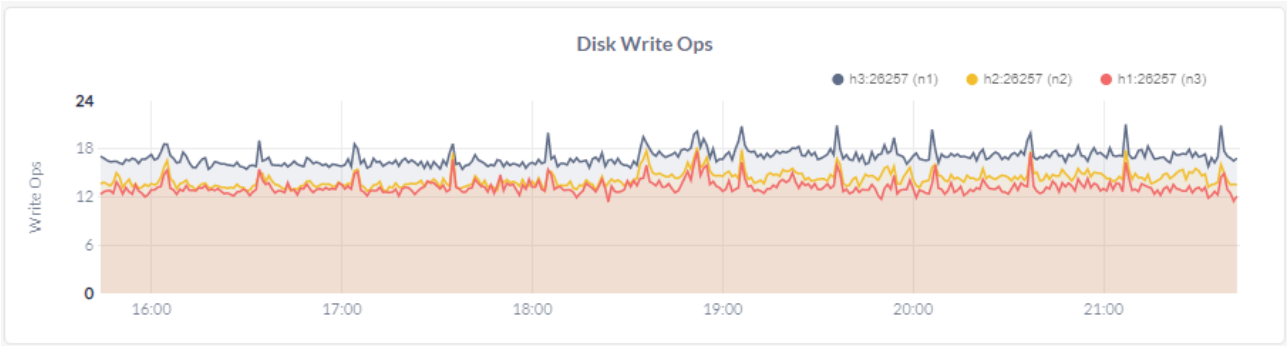


图 17: 输入图片说明

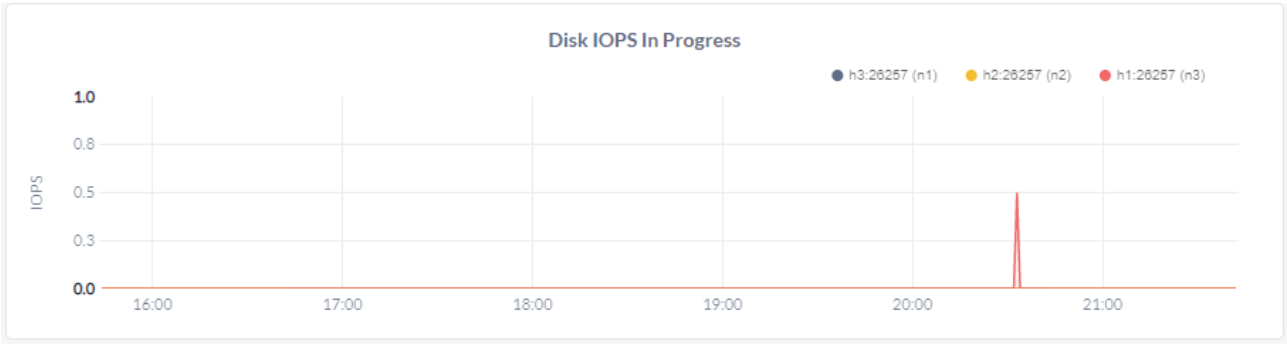


图 18: 输入图片说明

3.3.8.7 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。

3.3.8.8 可用磁盘容量

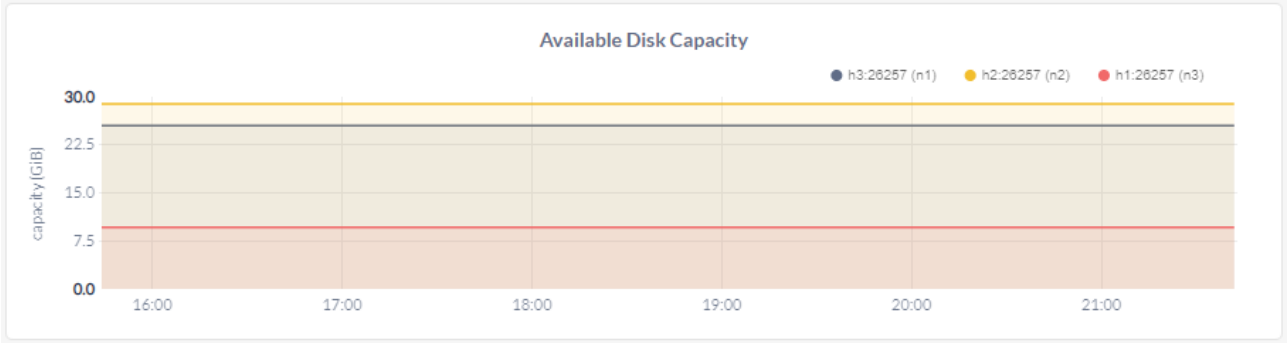


图 19: 输入图片说明

- 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。

3.3.8.9 收到的网络字节

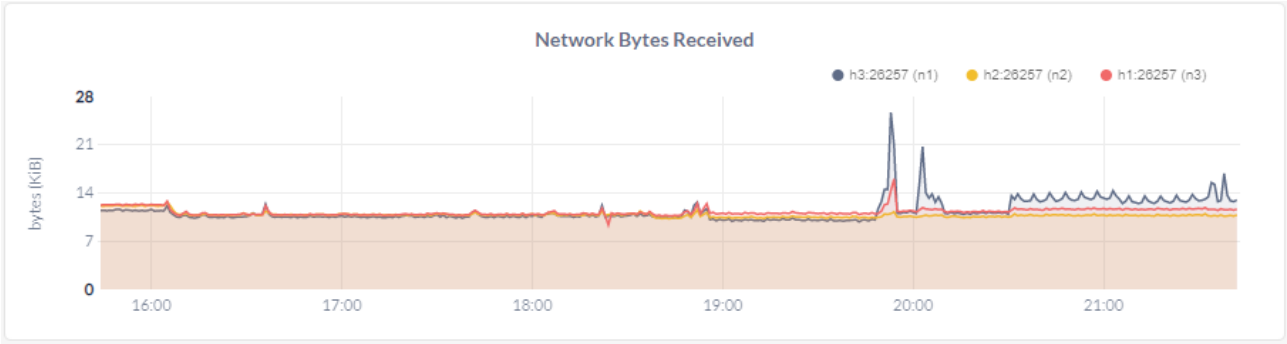


图 20: 输入图片说明

- 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。

3.3.8.10 发送的网络字节

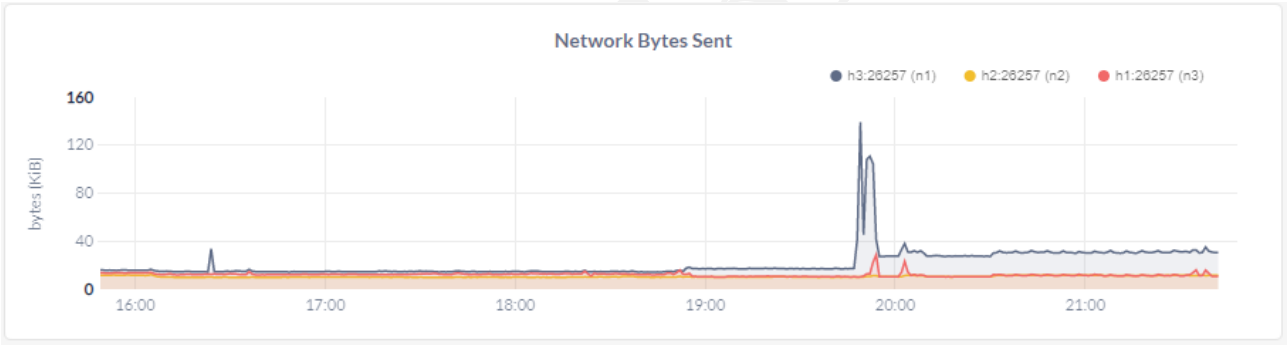


图 21: 输入图片说明

- 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。

3.3.9 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，然后选择仪表板 > 运行。

运行时仪表板显示以下时间序列图：

3.3.9.1 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。

图 22: 输入图片说明

HUBBLE

图 23: 输入图片说明

3.3.9.2 内存使用情况

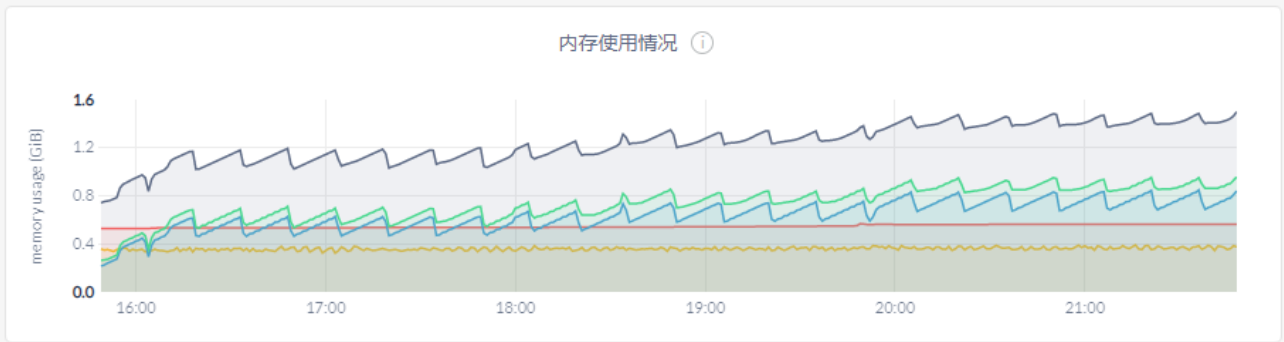


图 24: 输入图片说明

- 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：

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。

3.3.9.3 并发数

图 25: 输入图片说明

3.3.9.4 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。

3.3.9.5 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。

3.3.9.6 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。

图 26: 输入图片说明

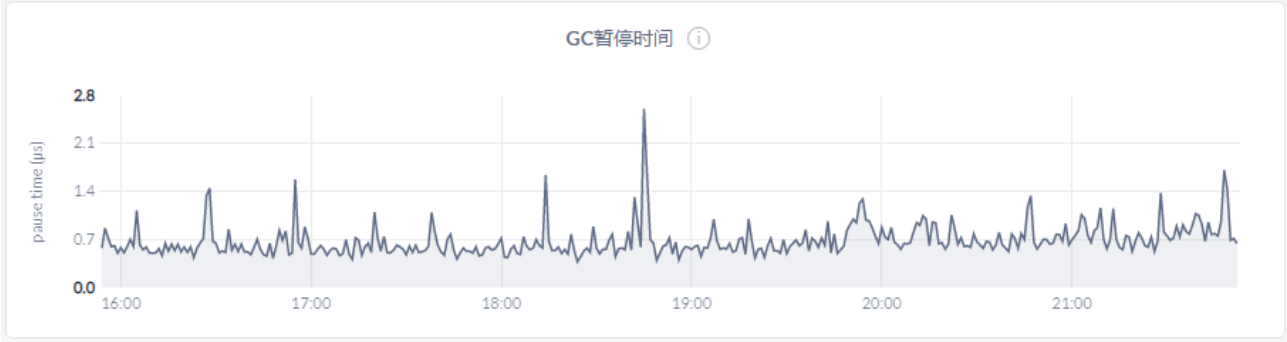


图 27: 输入图片说明

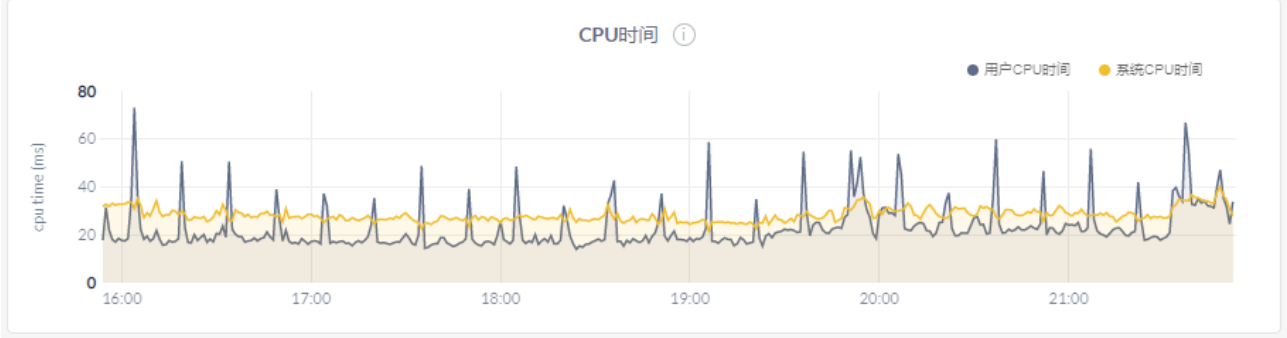


图 28: 输入图片说明

- 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。

3.3.9.7 时钟偏移

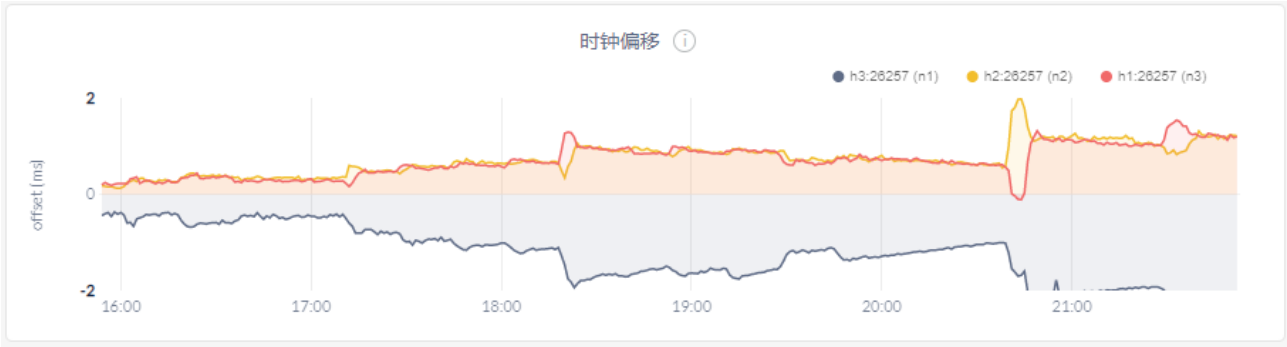


图 29: 输入图片说明

- 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。

3.3.10 SQL 仪表板

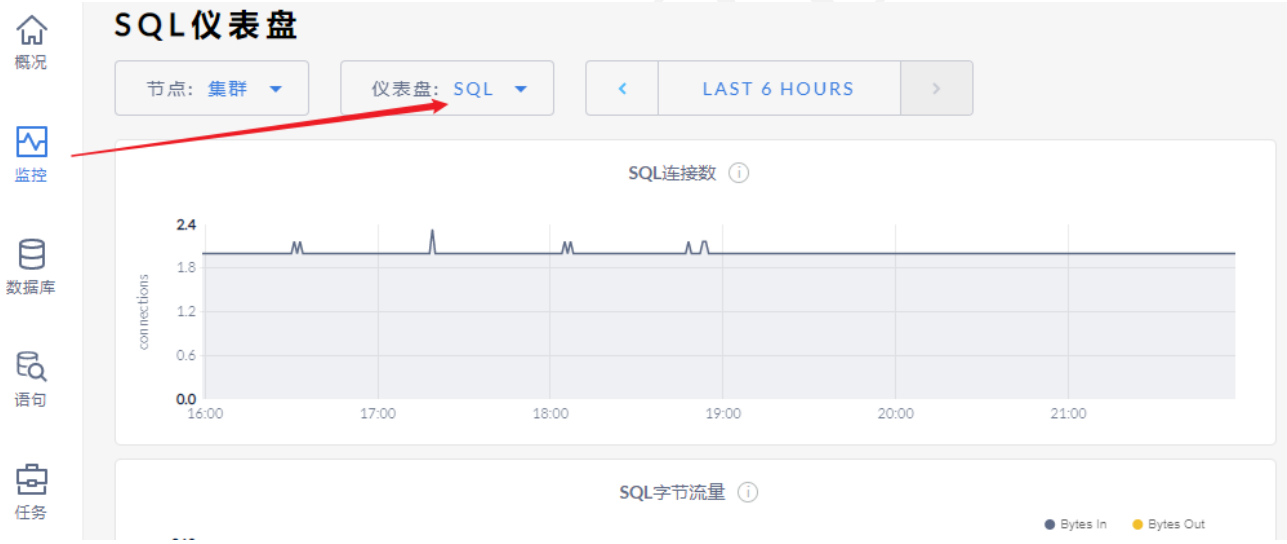


图 30: 输入图片说明

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，然后选择仪表板 >SQL。

SQL 仪表板显示以下时间序列图：

3.3.10.1 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。

3.3.10.2 SQL 字节流量

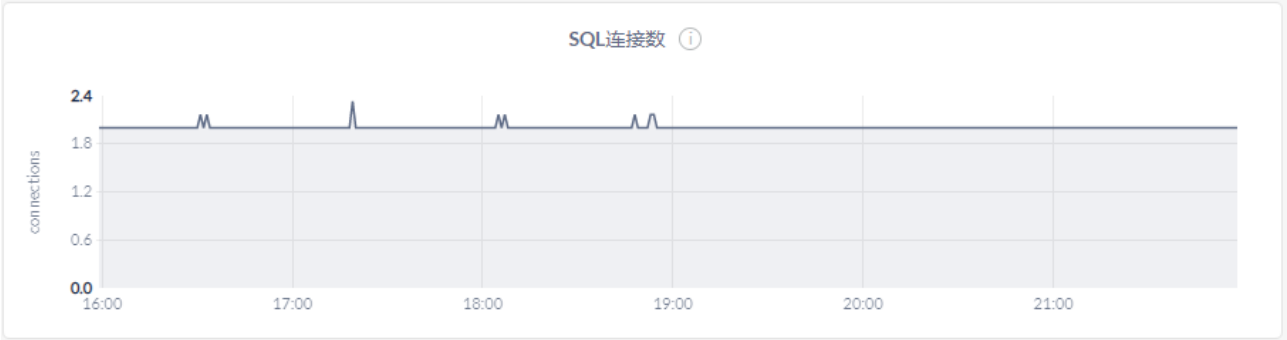


图 31: 输入图片说明

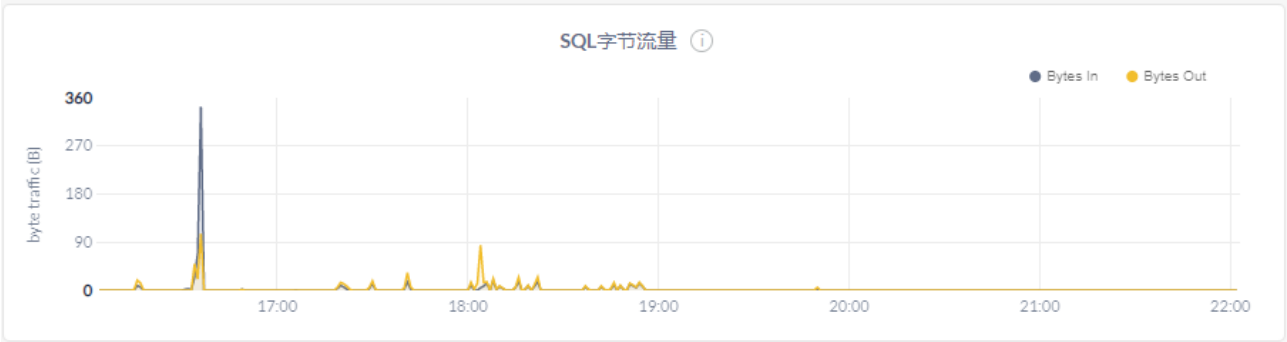


图 32: 输入图片说明

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。

3.3.10.3 SQL 查询数

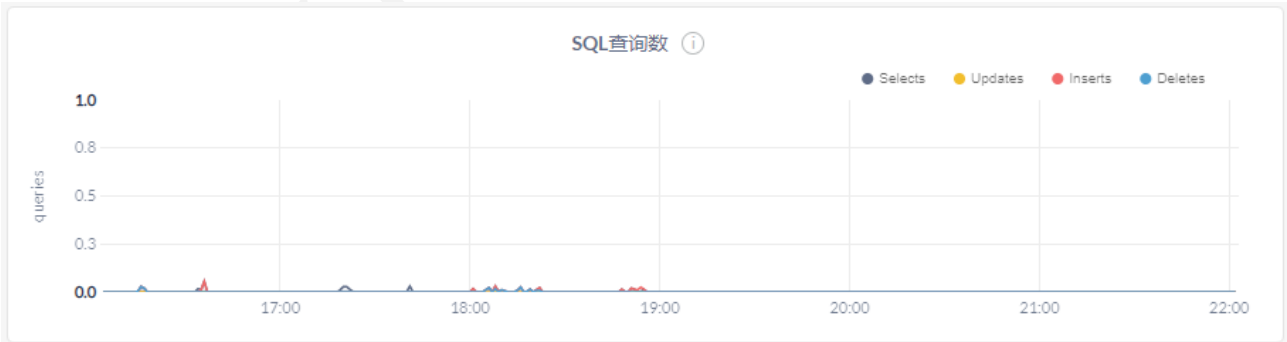


图 33: 输入图片说明

- 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。

3.3.10.4 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。

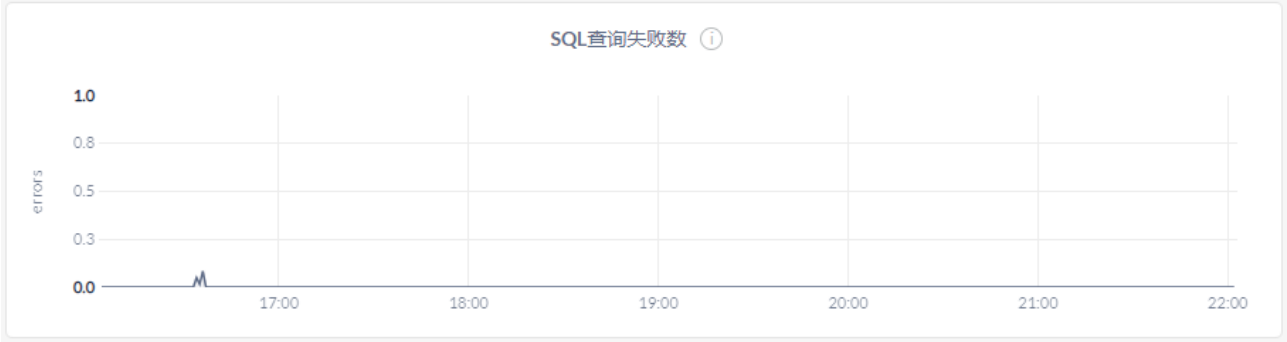


图 34: 输入图片说明

- 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。

3.3.10.5 事务数

图 35: 输入图片说明

- 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。

3.3.10.6 其他图表

主要用于开发者监控运行时使用

3.3.11 存储仪表板

通过 Admin UI 中的存储仪表板，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，单击左侧导航栏上的指标，然后选择仪表板 > 存储。

存储仪表板显示以下时间序列图：

3.3.11.1 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。

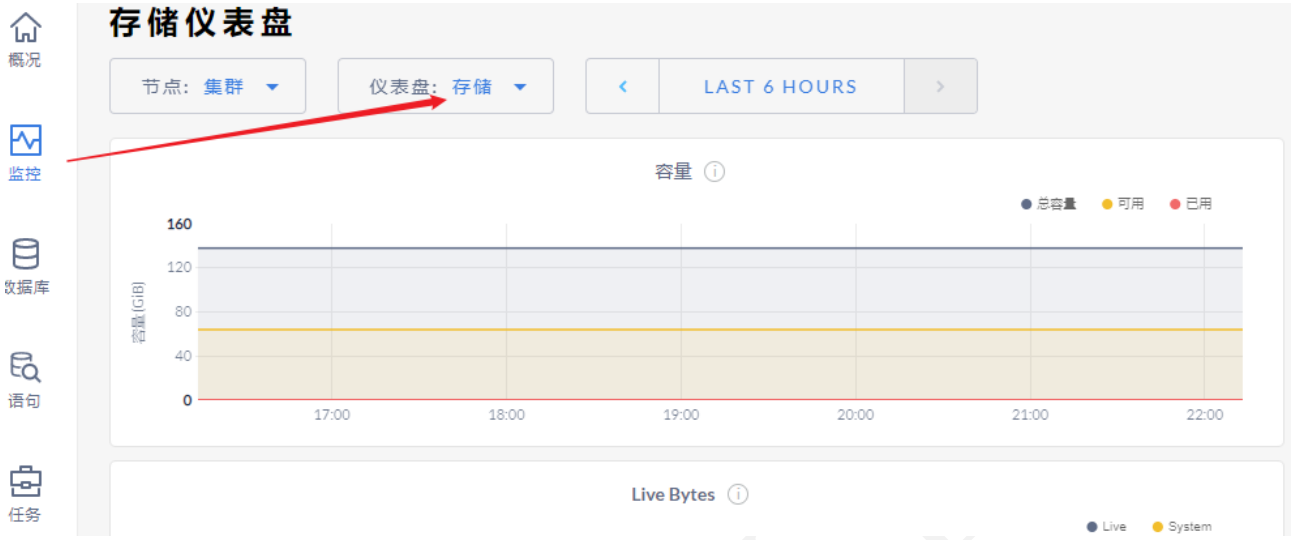


图 36: 输入图片说明

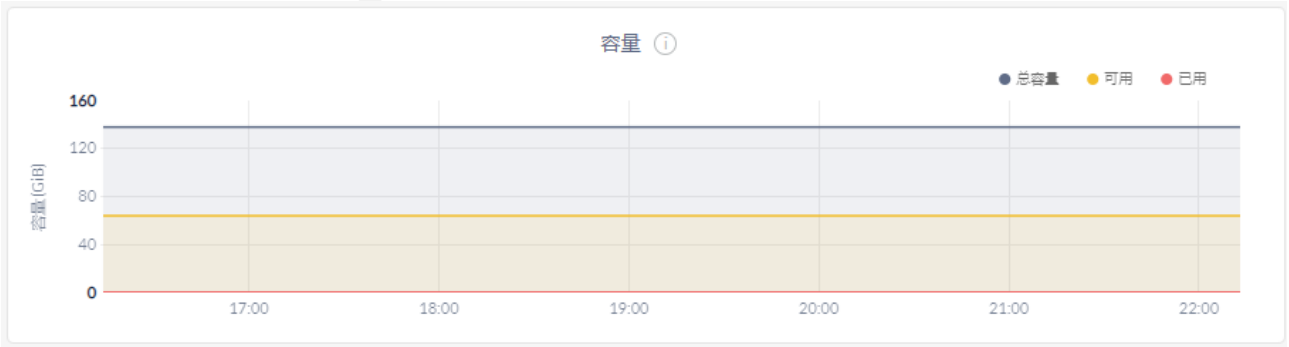


图 37: 输入图片说明

- 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。

3.3.11.2 文件描述符

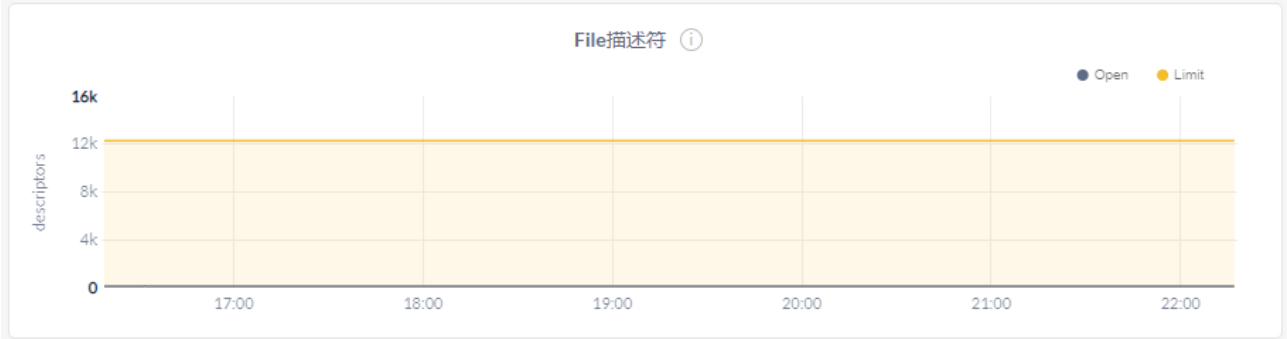


图 38: 输入图片说明

- 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。

3.3.11.3 其他图像

主要用于开发者监控运行时使用

3.3.12 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，然后选择仪表板 > 复制。

3.3.12.1 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：

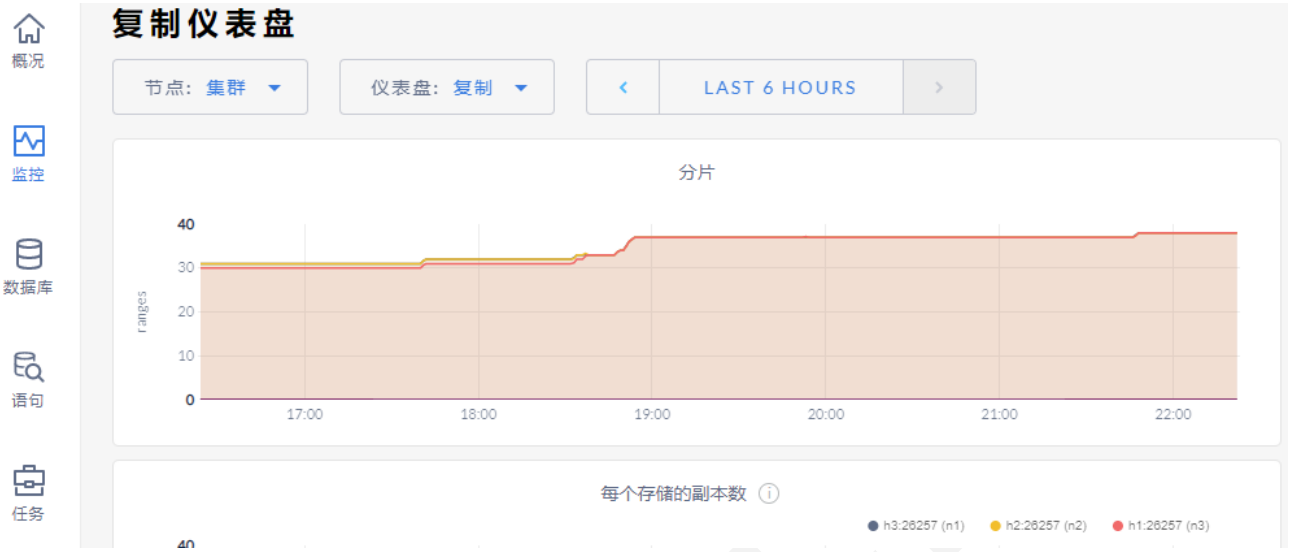


图 39: 输入图片说明

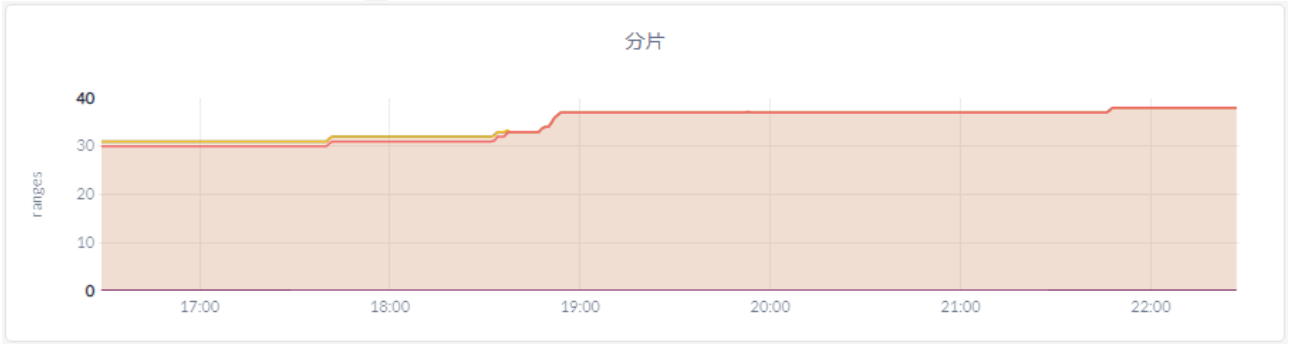


图 40: 输入图片说明

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。

3.3.12.2 副本静止数

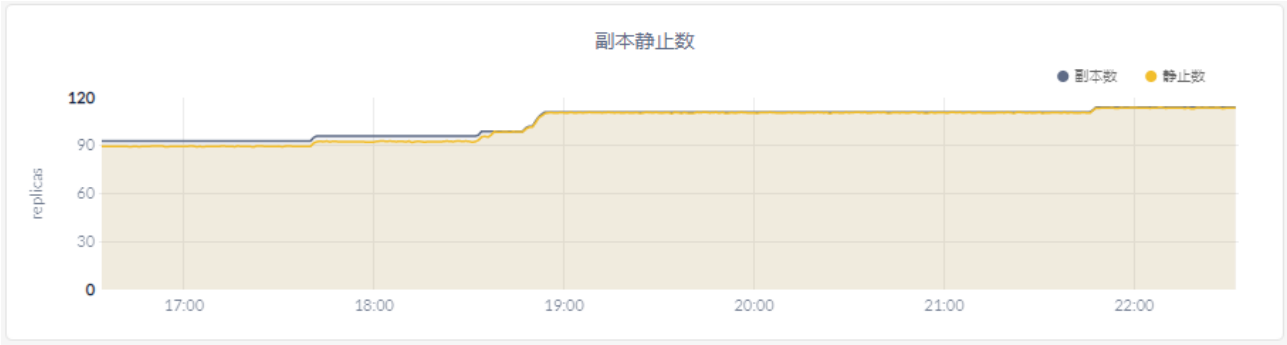


图 41: 输入图片说明

- 在节点视图中，该时间序列图显示的是指定节点的副本数量。
- 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	描述
Replicas	副本数
Quiescent	一段时间未访问的副本数量

3.3.12.3 快照数

图 42: 输入图片说明

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。

3.3.12.4 其他图形

主要用于开发者监控运行时使用

3.3.13 数据变动仪表板

图 43: 输入图片说明

通过 Admin UI 中的数据变动仪表板，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，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：

3.3.13.1 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。

3.3.13.2 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：

图 44: 输入图片说明

图 45: 输入图片说明

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。

3.3.13.3 接收数

图 46: 输入图片说明

- 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。

3.3.13.4 接收计时

图 47: 输入图片说明

TABLES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。

3.3.14.2 GRANTS 视图

GRANTS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，然后从左侧导航栏中选择数据库，从左侧导航栏中选择数据库，然后从VIEW菜单中选择GRANTS。

3.3.15 任务页面

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.

3.3.15.1 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。

3.3.15.2 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。

3.3.16 语句页面

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。

3.3.16.1 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。

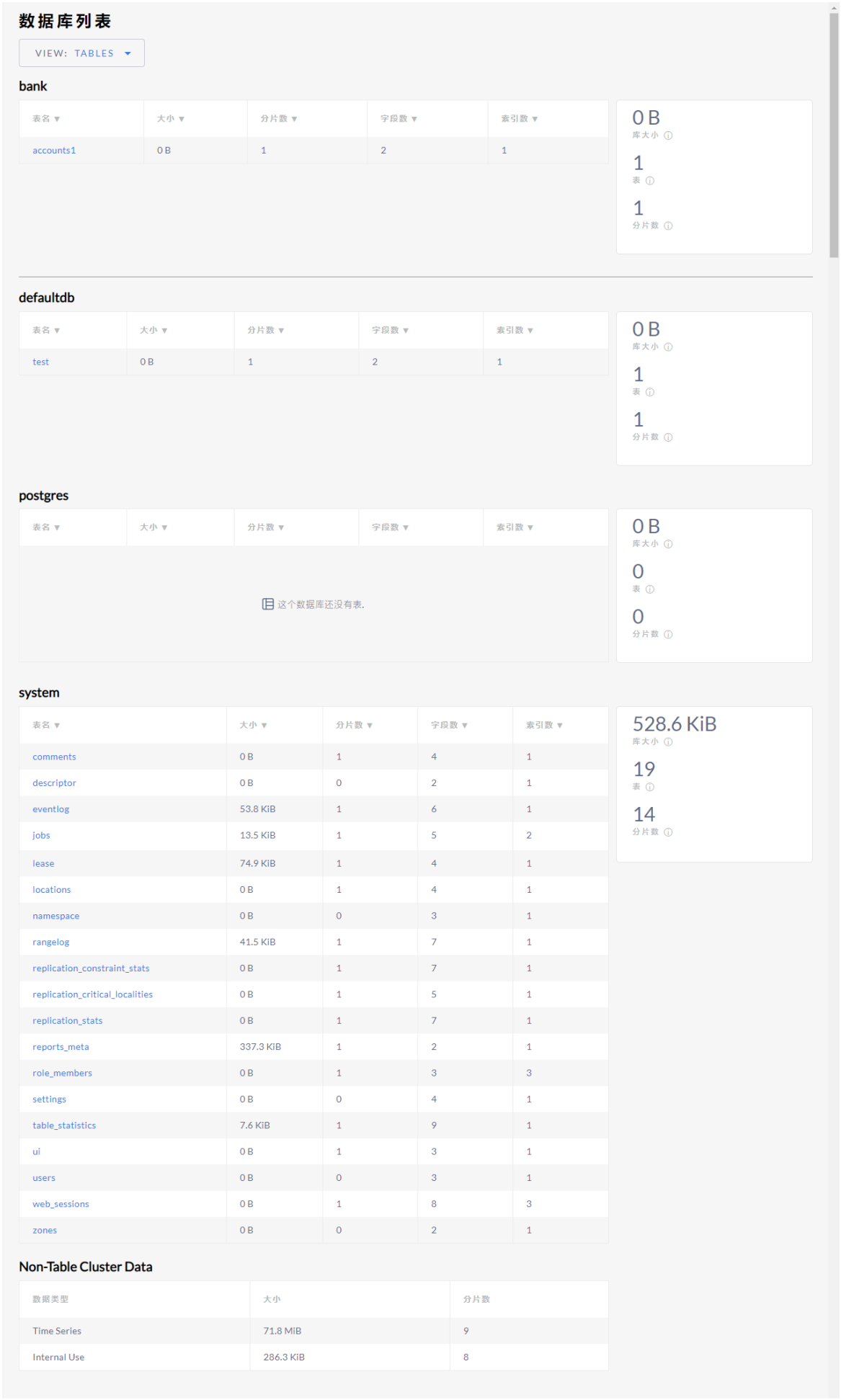

数据库列表
VIEW: TABLES
bank
表名 大小 分片数 字段数 索引数
accounts1 0 B 1 2 1
0 B 库大小
1 表
1 分片数
defaultdb
表名 大小 分片数 字段数 索引数
test 0 B 1 2 1
0 B 库大小
1 表
1 分片数
postgres
表名 大小 分片数 字段数 索引数
这个数据库还没有表.
0 B 库大小
0 表
0 分片数
system
表名 大小 分片数 字段数 索引数
comments 0 B 1 4 1
descriptor 0 B 0 2 1
eventlog 53.8 KiB 1 6 1
jobs 13.5 KiB 1 5 2
lease 74.9 KiB 1 4 1
locations 0 B 1 4 1
namespace 0 B 0 3 1
rangelog 41.5 KiB 1 7 1
replication_constraint_stats 0 B 1 7 1
replication_critical_localities 0 B 1 5 1
replication_stats 0 B 1 7 1
reports_meta 337.3 KiB 1 2 1
role_members 0 B 1 3 3
settings 0 B 0 4 1
table_statistics 7.6 KiB 1 9 1
ui 0 B 1 3 1
users 0 B 0 3 1
web_sessions 0 B 1 8 3
zones 0 B 0 2 1
528.6 KiB 库大小
19 表
14 分片数
Non-Table Cluster Data
数据类型 大小 分片数
Time Series 71.8 MiB 9
Internal Use 286.3 KiB 8

图 48: 输入图片说明

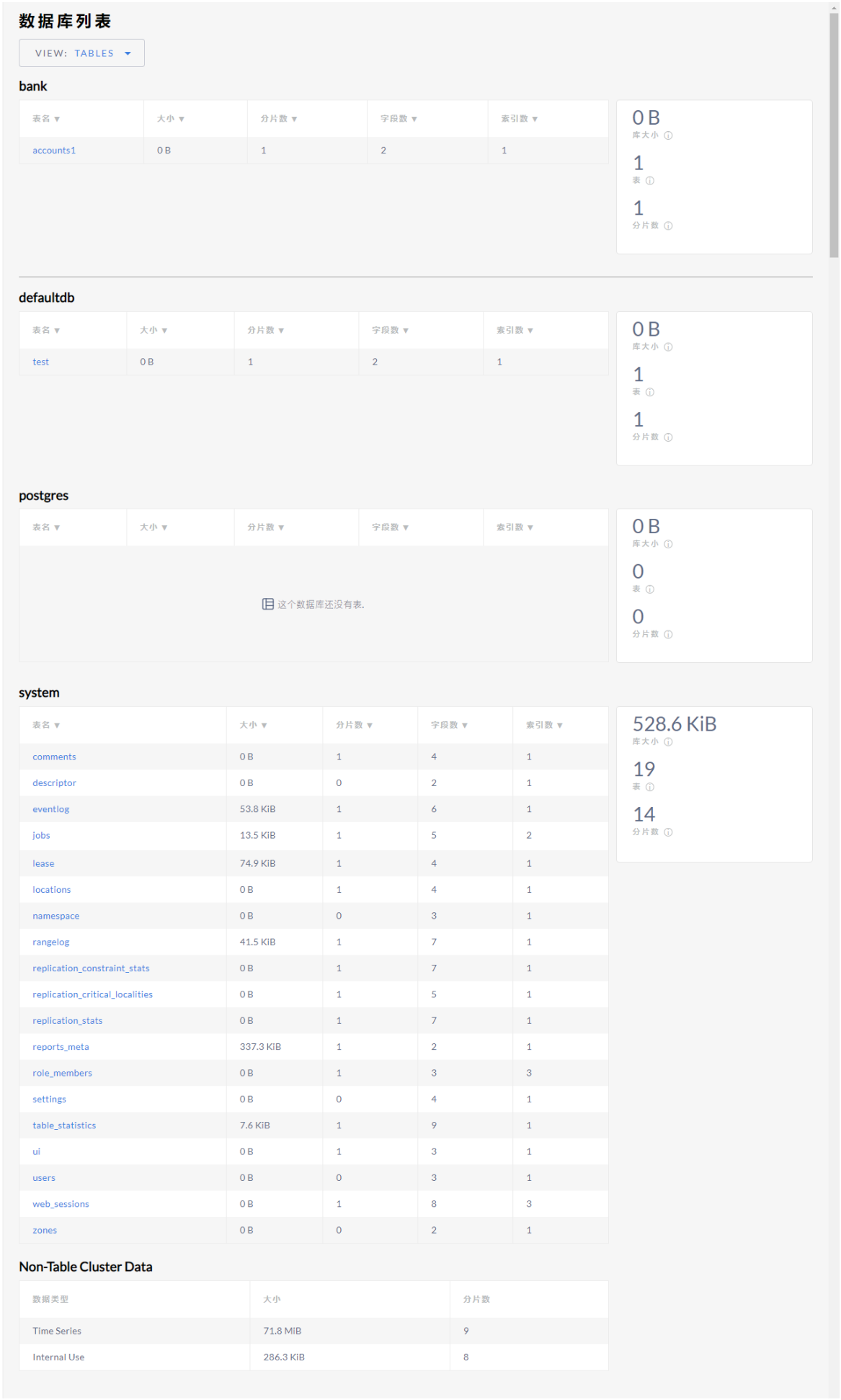

数据库列表
VIEW: TABLES
bank
表名 大小 分片数 字段数 索引数
accounts1 0 B 1 2 1
0 B 库大小
1 表
1 分片数
defaultdb
表名 大小 分片数 字段数 索引数
test 0 B 1 2 1
0 B 库大小
1 表
1 分片数
postgres
表名 大小 分片数 字段数 索引数
这个数据库还没有表.
0 B 库大小
0 表
0 分片数
system
表名 大小 分片数 字段数 索引数
comments 0 B 1 4 1
descriptor 0 B 0 2 1
eventlog 53.8 KiB 1 6 1
jobs 13.5 KiB 1 5 2
lease 74.9 KiB 1 4 1
locations 0 B 1 4 1
namespace 0 B 0 3 1
rangelog 41.5 KiB 1 7 1
replication_constraint_stats 0 B 1 7 1
replication_critical_localities 0 B 1 5 1
replication_stats 0 B 1 7 1
reports_meta 337.3 KiB 1 2 1
role_members 0 B 1 3 3
settings 0 B 0 4 1
table_statistics 7.6 KiB 1 9 1
ui 0 B 1 3 1
users 0 B 0 3 1
web_sessions 0 B 1 8 3
zones 0 B 0 2 1
528.6 KiB 库大小
19 表
14 分片数
Non-Table Cluster Data
数据类型 大小 分片数
Time Series 71.8 MiB 9
Internal Use 286.3 KiB 8

图 49: 输入图片说明

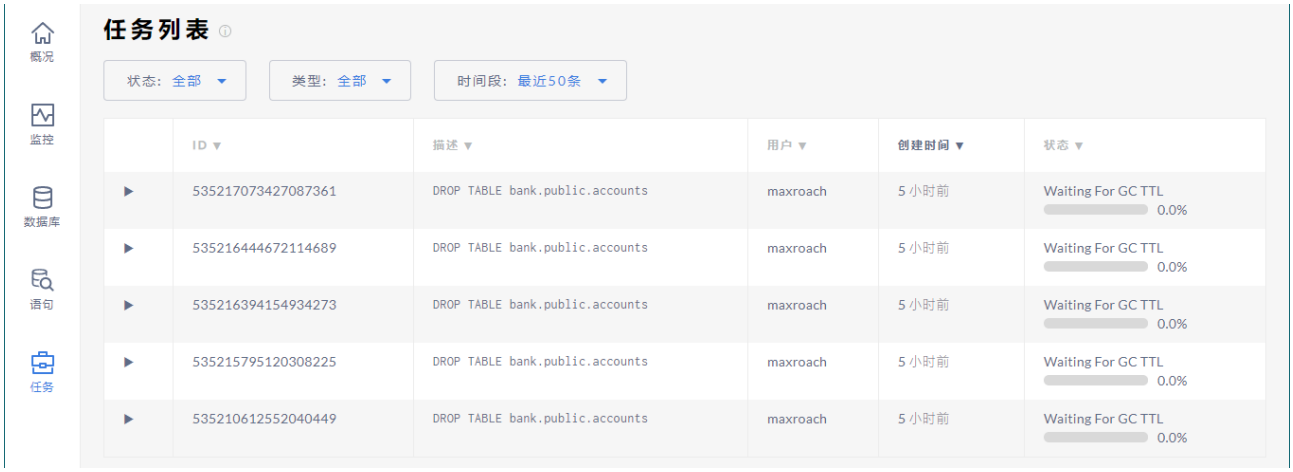


图 50: 输入图片说明

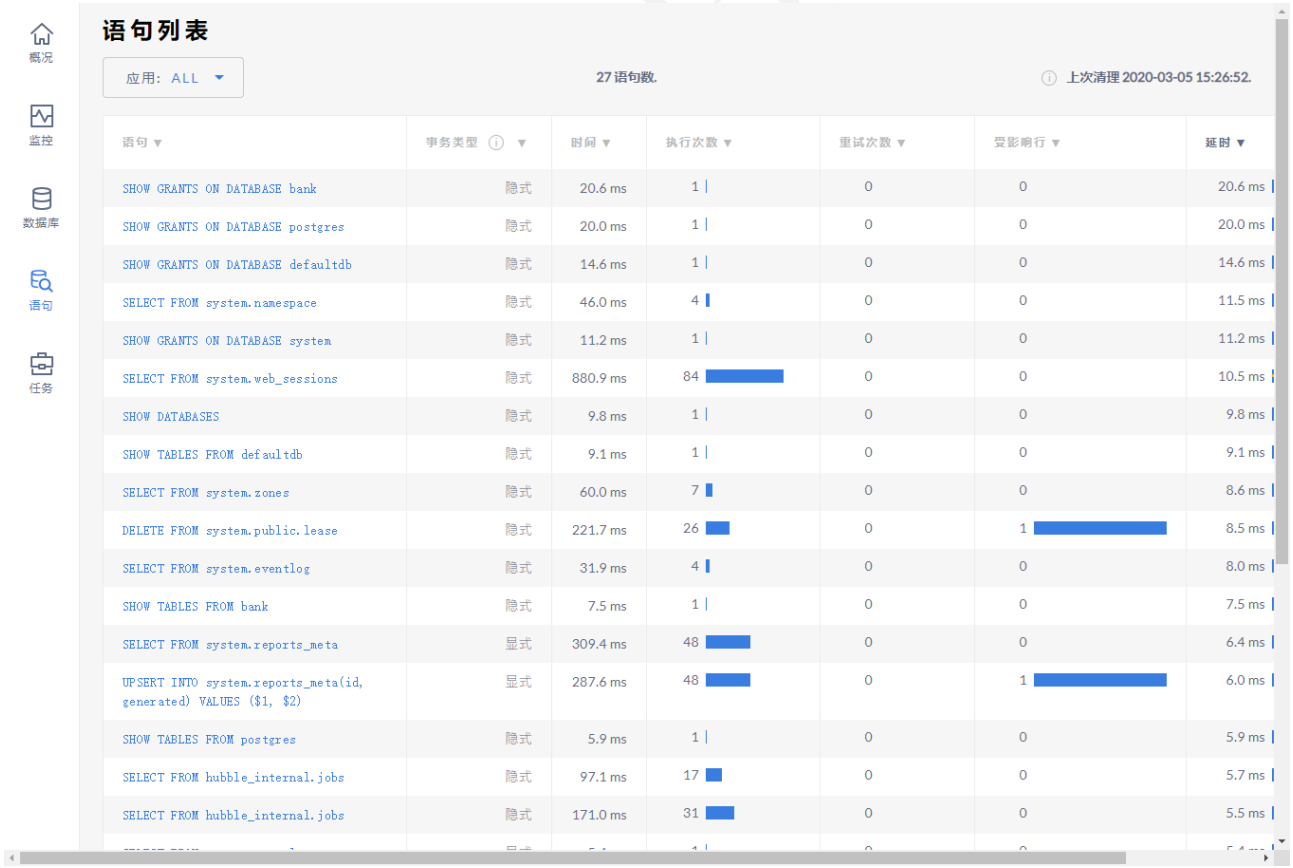


图 51: 输入图片说明

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。

3.3.17 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。

3.3.17.1 逻辑计划

3.3.17.2 阶段

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。

3.3.17.3 网关节点统计

提供了指定语句在每个网关节点上详细的数值指标。对于每个网关节点

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。
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。

HUBBLE

参数	描述
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。

3.3.17.4 执行计数

“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。

3.3.17.5 受影响行

行数表提供 SQL 语句返回的行的累积计数的平均值和平均值的一个标准偏差。该表以数字值和条形图提供了服务延迟详细信息。

3.3.17.6 统计信息

语句详细信息页面右侧的统计信息框提供了语句指纹的以下详细信息：

参数	简介
总时间	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句执行时间的累计值。
执行次数	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句执行的总次数。
执行时不重试	在最近 1 小时或是自定义时间间隔内，一个 SQL 语句中首次执行成功的占比。
平均服务时间	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。
平均行数	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。

统计信息框下方的表格提供以下详细信息：

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。
使用 CBO?	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。

3.4 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。

3.4.1 新节点准备

参考集群环境服务配置，做免密及用户配置

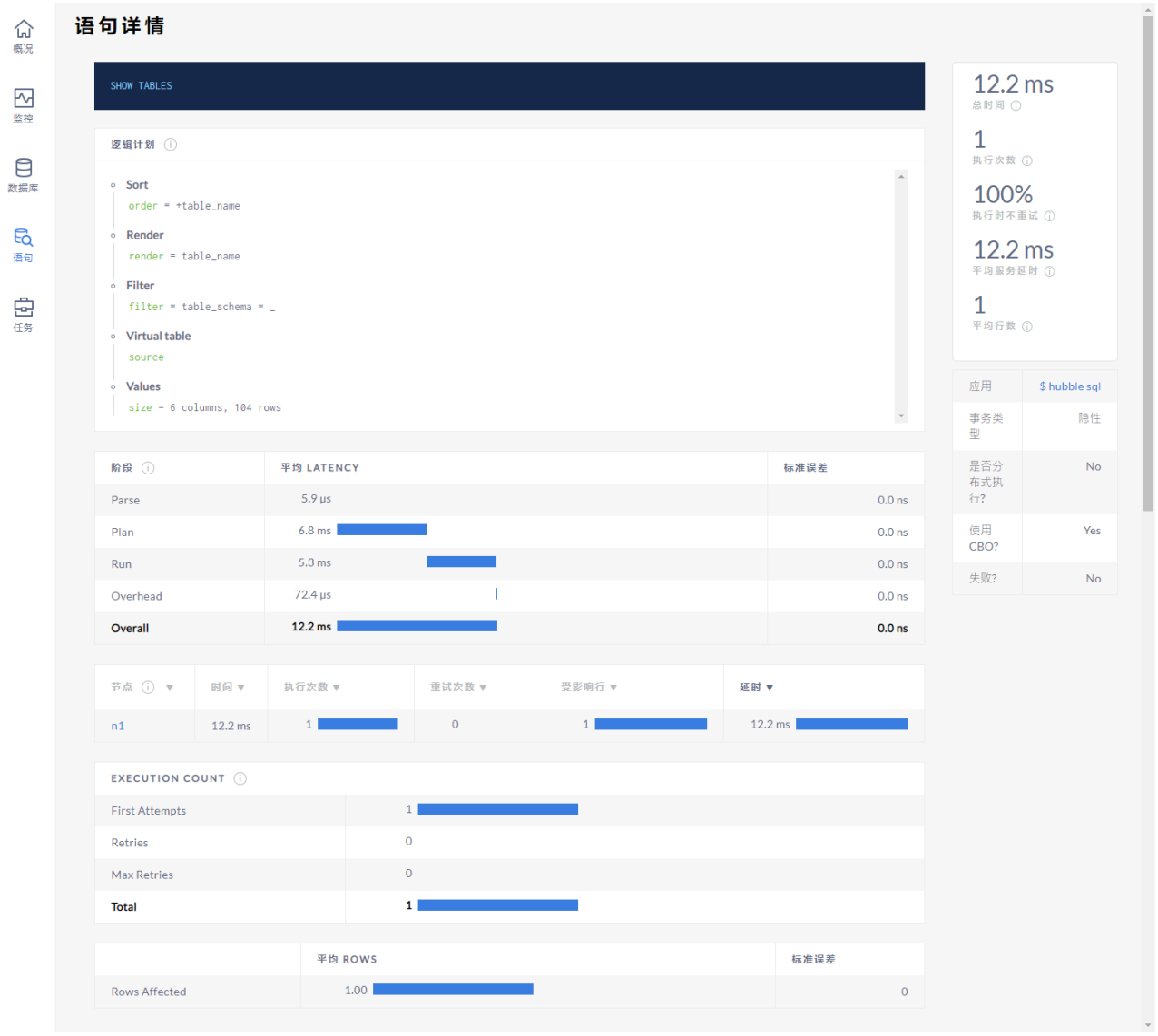


图 52: 输入图片说明

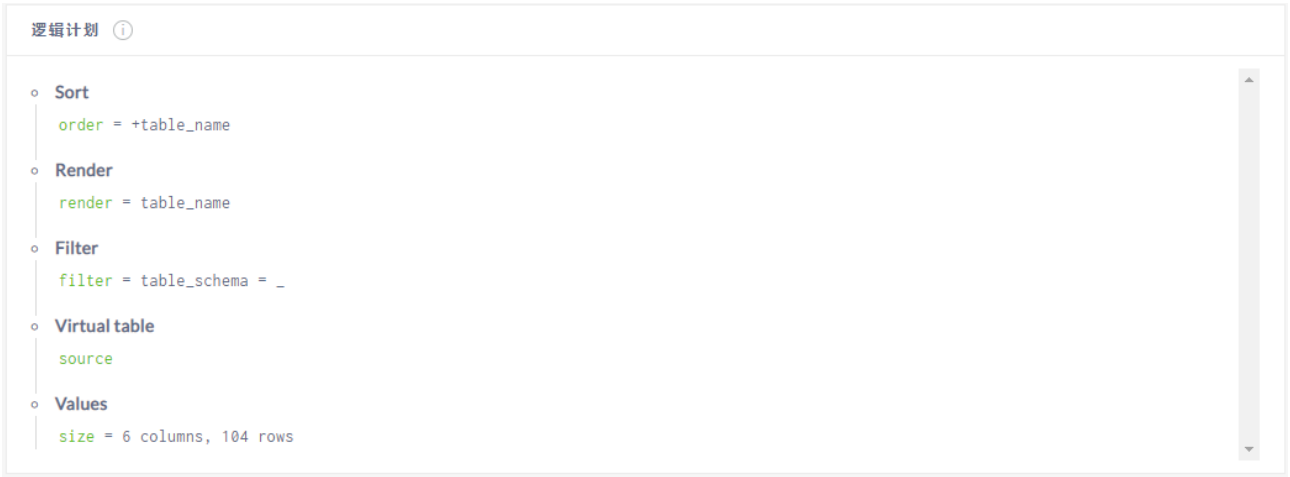


图 53: 输入图片说明

阶段	平均 LATENCY	标准误差
Parse	5.9 µs	0.0 ns
Plan	6.8 ms	0.0 ns
Run	5.3 ms	0.0 ns
Overhead	72.4 µs	0.0 ns
Overall	**12.2 ms**	**0.0 ns**

图 54: 输入图片说明

节点	时间	执行次数	重试次数	受影响行	延时
n1	12.2 ms	1	0	1	12.2 ms

图 55: 输入图片说明

EXECUTION COUNT	
First Attempts	1
Retries	0
Max Retries	0
Total	**1**

图 56: 输入图片说明

	平均 ROWS	标准误差
Rows Affected	1.00	0

图 57: 输入图片说明

图 58: 输入图片说明

应用	$ hubble sql
事务类型	隐性
是否分布式执行?	No
使用CBO?	Yes
失败?	No

图 59: 输入图片说明

3.4.2 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 /data/hubbledir    ## 数据存储目录，按集群存储规划创建
chown -R hubble:hubble /data/hubbledir
```

3.4.3 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
```

3.4.4 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=<new node>: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
    ↪  --sql-audit-dir=/data/hubbledir/auditdir --log-dir=/data/hubbledir/logs
    ↪ --log-dir-max-size=1GiB --max-disk-temp-storage=10GiB --external-io-dir=/
    ↪ data_shares
TimeoutStopSec=60
Restart=always
RestartSec=10
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=hubble
User=hubble
LimitNOFILE=1000000
[Install]
WantedBy=default.target
```

注意： 上述 hubble.service 文件中的 `<new node>` 相关内容修改为当前节点内容

3.4.5 添加普通用户启停特权

在新节点上，使用 root 用户操作，修改后输入 `:wq!` 保存退出

```
## 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
```

HUBBLE

```
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
```

3.4.6 启动服务

使用 hubble 用户操作

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

3.4.7 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>
```

3.4.8 设置开机自启动

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

3.5 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。

3.5.1 删除单个活动节点 (安全模式)

3.5.1.1 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。

HUBBLE

- 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。

3.5.1.2 步骤 1. 停用之前检查节点

展示所有节点状态：

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

```
  id | address   | ... | is_available | is_live
+----+-----------+ ... +--------------+---------+
   1 | h3:26257  | ... | true         | true
   3 | h2:26257  | ... | true         | true
   5 | h4:26257  | ... | true         | true
   6 | h1:26257  | ... | true         | true
(4 rows)
```

3.5.1.3 步骤 2. 退役并删除节点

SSH 到运行该节点的计算机，并使用--decommission标志和其他必需标志执行 hubble quit 命令：

该命令区别于多节点退役的命令

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

然后，您将在退役状态更改时将其打印到 stderr：

```
 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
```

3.5.1.4 步骤 3. 停用后检查节点和群集

检查集群状态：

```
hubble node status --decommission --certs-dir certs --host <live node>
```

HUBBLE

```
id | address  |...| is_available|is_live|gossiped_replicas|is_decommissioning|
   ↪ is_draining
---+----------+...+-------------+-------+-----------------+------------------+-----------
   ↪
 1 | h3:26257 |...| true        |true   |               76|false             |
    ↪ false
 3 | h2:26257 |...| true        |true   |               76|false             |
    ↪ false
 5 | h4:26257 |...| false       |false  |                0|true              |
    ↪ true
 6 | h1:26257 |...| true        |true   |               76|false             |
    ↪ false
```

注意 id=5 的节点 is_available，is_live，is_decommissioning 和 is_draining 值其他节点的不同

同样，如果节点重新启动，它将不接受任何客户端连接，并且群集也不会重新平衡任何数据。为了使群集再次利用该节点，您必须重新调试它。

3.5.2 安全删除单个已停止节点

节点关闭 5 分钟后，Hubble 会自动将节点上的分片副本和分片租约转移到可用的活动节点。但是，如果重新启动，群集将重新平衡副本与租约给它。

为了防止群集在数据恢复联机后将其重新平衡到死节点，请执行以下操作：

3.5.2.1 步骤 1. 识别已停止节点的 ID

检查集群decommission状态：

```
hubble node status --decommission --certs-dir certs --host <live node>
```

```
id | address |...|is_available|is_live|gossiped_replicas|is_decommissioning|
   ↪ is_draining
---+---------+...+------------+-------+-----------------+------------------+-----------
   ↪
 1 | h3:26257|...|true        |true   |               76|false             |
    ↪ false
 3 | h2:26257|...|true        |true   |               76|false             |
    ↪ false
 5 | h4:26257|...|true        |true   |               76|false             |
    ↪ false
 6 | h1:26257|...|false       |false  |                0|false             |true
(4 rows)
```

id=6 的节点为观察宕机或者坏掉的节点，其 is_available，is_live，is_decommissioning 为 false，is_draining 为 true；is_draining 表示枯竭的不可用的，但是没有执行 decommission

3.5.2.2 步骤 2. 将失效节点标记为已停用

SSH 到集群中的任何活动节点，然后运行具有该节点 ID 的`hubble node decommission`命令以正式停用，以使该节点的执行正式退役的操作：

```
hubble node decommission <DeadNodeID> --certs-dir=certs --host=<address of live
    ↪ node>
```

```
例如 DeadNodeID=6
id | is_live | replicas | is_decommissioning | is_draining
+---+---------+----------+--------------------+-------------+
  6 |  false  |        0 |        true        |    true
(1 row)

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

同样，如果节点曾经重新启动，它将不接受任何客户端连接，并且群集也不会重新平衡任何数据。为了使群集再次利用该节点，您必须重新调试它。

3.5.3 删除多个节点

3.5.3.1 开始之前

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。

3.5.3.2 步骤 1. 确定要停用的节点的 ID

检查集群状态

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

```
  id | address   |  ... | is_available | is_live
+----+-----------+- ... +--------------+---------+
   1 | h3:26257  |  ... | true         | true
   2 | h2:26257  |  ... | true         | true
   3 | h1:26257  |  ... | true         | true
   4 | h4:26257  |  ... | true         | true
   5 | h5:26257  |  ... | true         | true
```

3.5.3.3 步骤 2. 停用节点

SSH 到集群中的任何活动节点，并运行带有节点 ID 的hubble node decommission命令以正式停用

```
hubble node decommission 4 5 --certs-dir=certs --host=<address of live node>
```

然后，您将在退役状态更改时将其打印到 stderr：

```
 id | is_live | replicas | is_decommissioning | is_draining
+---+---------+----------+--------------------+-------------+
  4 |  true   |       18 |        true        |    false
  5 |  true   |       16 |        true        |    false
(2 rows)
```

节点完全停用后，您将看到一个确认：

```
 id | is_live | replicas | is_decommissioning | is_draining
+---+---------+----------+--------------------+-------------+
  4 |  true   |        0 |        true        |    false
  5 |  true   |        0 |        true        |    false
(2 rows)

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

3.5.3.4 步骤 4. 停用后检查节点和群集

检查集群状态

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

```
id|address |...|is_available|is_live|gossiped_replicas|is_decommissioning|
    ↪ is_draining
--+--------+...+------------+-------+-----------------+------------------+----------
    ↪
 1|h3:26257|...| true       | true  |               76|false             |false
 2|h2:26257|...| true       | true  |               76|false             |false
 3|h1:26257|...| true       | true  |               76|false             |false
 4|h4:26257|...| true       | true  |               74|true              |false
 5|h5:26257|...| true       | true  |               75|true              |false
```

id=4,5 的节点 is_decommissioning 为 true

3.5.3.5 步骤 5. 删除退役的节点

在这一点上，尽管退役的节点处于活动状态，但是群集不会向其重新平衡任何数据，并且节点将不接受任何客户端连接。但是，要从群集中正式删除节点，您仍然需要停止它们。

对于每个停用的节点，通过 SSH 连接到运行该节点的计算机，然后执行hubble quit命令:

```
hubble quit --certs-dir=certs --host=<address of decommissioned node>
```

```
ok
```

检查集群状态

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

```
id|address |...|is_available|is_live|gossiped_replicas|is_decommissioning|
    ↪ is_draining
--+--------+...+------------+-------+-----------------+------------------+----------
    ↪
 1|h3:26257|...| true       | true  |               76|false             |false
```

HUBBLE

```
 2|h2:26257|...| true       | true  |               76|false             |false
 3|h1:26257|...| true       | true  |               76|false             |false
 4|h4:26257|...| false      | false |               74|true              |true
 5|h5:26257|...| false      | false |               75|true              |true
```

3.5.4 重新调试节点（安全模式）

如果您不小心使任何节点退役，拥有导致进程挂起的退役节点，或者以其他方式希望退役节点作为活动成员重新加入群集，请执行以下操作:

3.5.4.1 步骤 1. 识别退役节点的 ID

```
hubble node status --decommission --certs-dir certs --host <address of live node
    ↪ >
```

```
id|address |...|is_available|is_live|gossiped_replicas|is_decommissioning|
    ↪ is_draining
--+--------+...+------------+-------+-----------------+------------------+----------
    ↪
 1|h3:26257|...|true        |true   |               61|false             |false
 3|h2:26257|...|true        |true   |               61|false             |false
 5|h4:26257|...|true        |true   |                0|true              |false
 6|h1:26257|...|true        |true   |               61|false             |false
```

id=5 的节点为退役节点

3.5.4.2 步骤 2. 重新调试节点

SSH 到活动节点之一，并使用要重新调试的节点的 ID 执行hubble node recommission 命令:

```
hubble node recommission 5 --certs-dir=certs --host=<address of live node>
```

```
 id | is_live | replicas | is_decommissioning | is_draining
+---+---------+----------+--------------------+-------------+
  5 |  false  |        0 |       false        |    true
(1 row)
The affected nodes must be restarted for the change to take effect.
```

3.5.4.3 步骤 3. 重新启动重新使用的节点

SSH 到具有重新调试的节点的每台计算机，并运行与最初启动该节点时相同的 hubble 启动命令，例如:

```
hubble start --certs-dir=certs --advertise-addr=<address of node to restart> --
    ↪ join=<address of node 1> --background
```

3.5.5 总结

停用节点要考虑表的表的分片副本数，只有节点数 < 所有表的最大分片副本数，才可以停用节点，且可停用的节点数 = 节点数-所有表的最大分片副本数；所有表的最大分片副本数 = 分片 + 分片的副本，SHOW ZONE ↪ CONFIGURATION FOR TABLE users; 会体现该 users 表的副本情况；即所有表中某个表的分片副本数 =5，节

点数 =7, 那么可停用 2 个节点，并且数据库会自动平衡停止的那 2 个节点的副本到其他节点上；当分片副本数 = 节点数时，无法停用节点，进行停用操作后，会标记被停用节点为“可疑”且无限期延迟，直至添加新的节点才可停用旧节点，关键要满足副本数的要求，这是分布式数据库架构的基本要求。

3.6 常见问题

主要包含一些使用数据库时经常需要注意到的问题，需要持续更新。

3.6.1 导入问题

1. 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，例如`SET CLUSTER SETTING kv.bulk_io_write.max_rate = '10MB';`

2. 导入格式问题，原则上需要导入标准 csv，一些常用的 csv 导入参数：

   `delimiter`分隔符，

   `strict_quotes`是否严格 csv（如果在一个字段里存在"", 默认关闭）

   `nullif` 转化 null

   `rows_terminated_by` 行分隔

   `fields_enclosed_by` 字段关闭规则，默认"

   `fields_escaped_by` 特殊字符处理。

3.6.2 副本数

默认 3 副本，容忍同时一个节点宕机，可通过`ALTER RANGE ... CONFIGURE ZONE USING num_replicas=...;`来设置这个区域内的副本数（默认配置在一个区域，对于大型的集群按机架、位置等设置区域）

3.6.3 错误日志

如果在某些节点已经被停用的状态下，在其他节点上遇到`grpc: addrConn.createTransport failed to connect to`错误日志，目前的解决办法是滚动重启。

> 依赖于 grpc，grpc 存在这个问题，目前无法避免

3.7 日常巡检

3.7.1 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。

3.7.2 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。

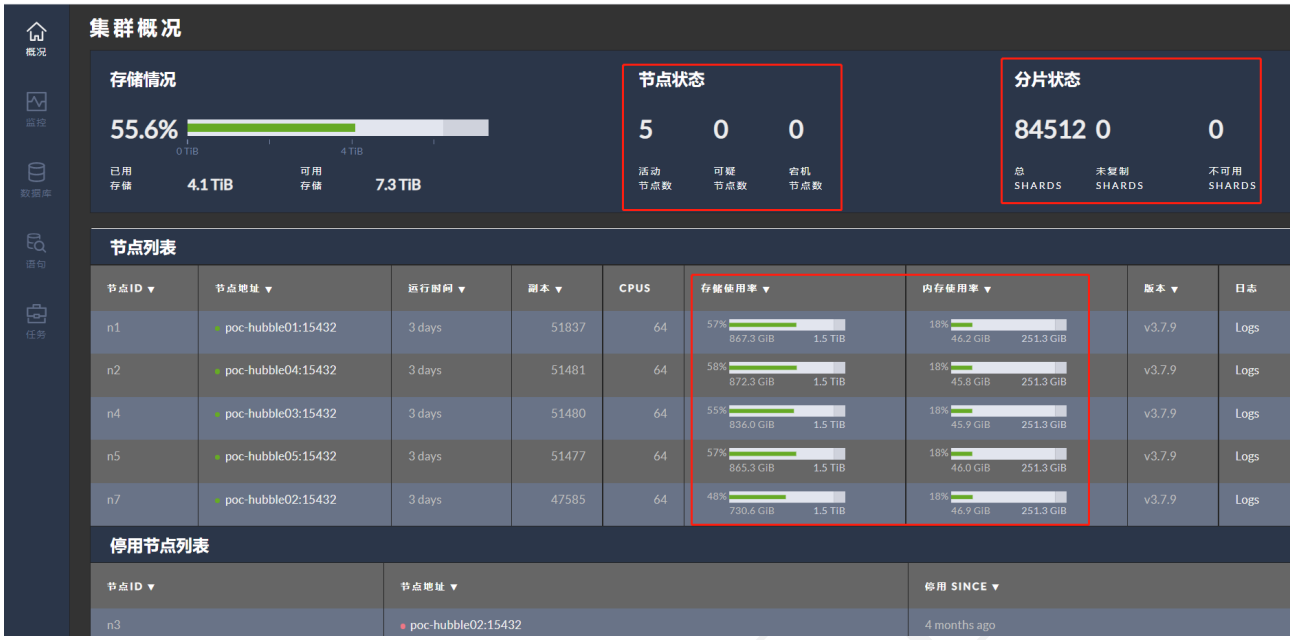
3.7.3 登录的网址

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

HUBBLE

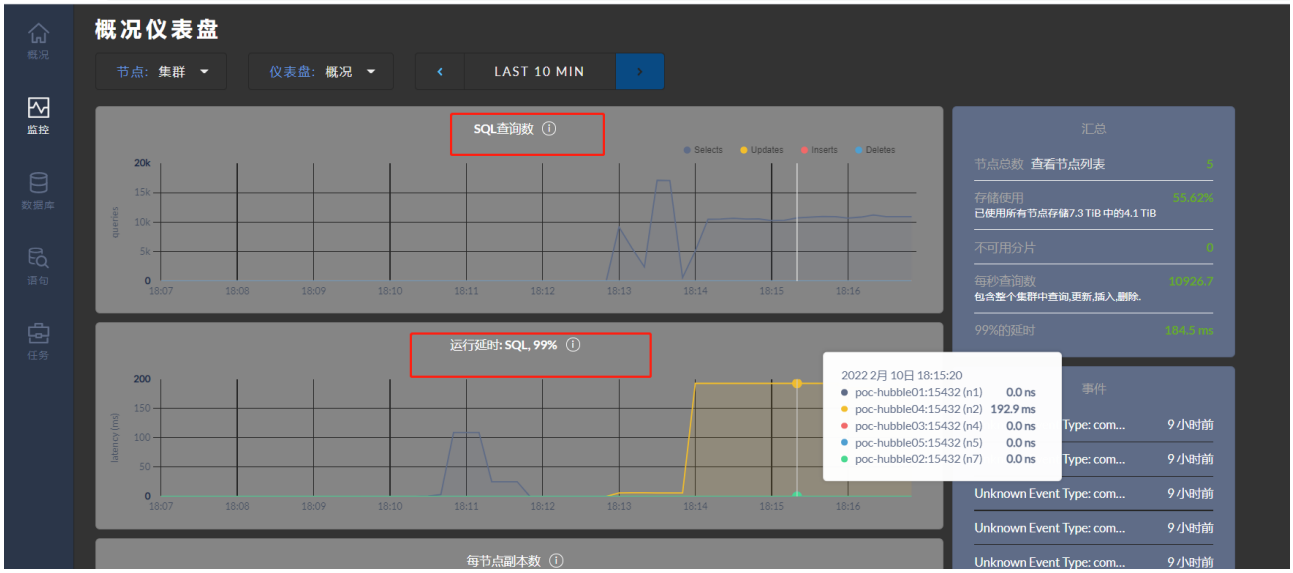
3.7.4 页面服务

3.7.4.1 概况



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

3.7.4.2 监控



重点关注 SQL 查询数是否有任务

SQL 运行延时问题，如果延时时间是 >3 秒的需要进行关注

3.7.4.3 语句

关注语句平均执行时间是否大于 3 秒，如果是要引起关注

3.7.4.4 任务

关注是否有较多的进行中的任务

3.7.5 异常处理检测

3.7.5.1 节点实例查看

```
hubble node status --certs-dir=/var/lib/hubbletp/certs --host=poc-hubble01:15432
```

```
[hubble@poc-hubble01 ~]$ hubble node status --certs-dir=/var/lib/hubbletp/certs --host=poc-hubble01:15432
  id |      address       |     sql_address     | build  |            started_at            |            updated_at            |                          locality
| is_available | is_live
+----+--------------------+--------------------+--------+----------------------------------+----------------------------------+-------------------------------------------------------------
+-------------+---------+
   1 | poc-hubble01:15432 | poc-hubble01:15432 | v3.7.9 | 2022-02-07 01:18:23.956347+00:00 | 2022-02-10 10:33:29.046086+00:00 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=1
| true         | true
   2 | poc-hubble04:15432 | poc-hubble04:15432 | v3.7.9 | 2022-02-07 01:36:00.203055+00:00 | 2022-02-10 10:33:26.722142+00:00 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=4
| true         | true
   4 | poc-hubble03:15432 | poc-hubble03:15432 | v3.7.9 | 2022-02-07 01:35:53.259936+00:00 | 2022-02-10 10:33:27.669689+00:00 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=3
| true         | true
   5 | poc-hubble05:15432 | poc-hubble05:15432 | v3.7.9 | 2022-02-07 01:36:38.21816+00:00  | 2022-02-10 10:33:29.154096+00:00 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=5
| true         | true
   7 | poc-hubble02:15432 | poc-hubble02:15432 | v3.7.9 | 2022-02-07 03:13:56.515928+00:00 | 2022-02-10 10:33:25.083643+00:00 | country=cn,region=ch-beijin,datacenter=tianyun,rack=1,node=2
| true         | true
(5 rows)
```

3.7.5.2 运行任务处理

```
show jobs;
```

```
root@poc-hubble01:15432/zl> show jobs;
        job_id       |   job_type    |       description       | statement | user_name | status  |   running_status   |             created             |             started             |
finished |             modified             | fraction_completed | error | coordinator_id
+--------------------+---------------+-------------------------+-----------+-----------+---------+--------------------+---------------------------------+---------------------------------+-
---------+---------------------------------+--------------------+-------+---------------+
  735376389088280577 | SCHEMA CHANGE | DROP TABLE zl.public.tf |           | root      | running | waiting for GC TTL | 2022-02-10 10:37:44.05017+00:00 | 2022-02-10 10:37:44.063655+00:00 |
NULL     | 2022-02-10 10:37:44.126309+00:00 |                  0 |       |           NULL
(1 row)
```

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 735376389088280577;
```

```
root@poc-hubble01:15432/zl> cancel job 735376389088280577;
CANCEL JOBS 1

Time: 10.910796ms

root@poc-hubble01:15432/zl> show jobs;
```

3.7.5.3 运行 sql 语句处理

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

例如，假设您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';
```

3.7.6 服务启停

注意使用 hubble 账户

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
```

3.7.7 服务器端

主要检查 cpu、内存、网络、硬盘占用情况

3.8 紧急预案

3.8.1 节点重启 (注意使用 hubble 账户操作)

如果需要对服务器进行启动或者关机操作，执行以下命令

数据库状态

```
systemctl status hubble
```

关闭数据库

```
sudo systemctl stop hubble
```

启动数据库

```
sudo systemctl start hubble
```

3.8.2 节点可疑/宕机

在偶发情况下，大量的并发任务操作 (增删改查、导出、导入、drop、数据副本移动等) 情况下，导致集群硬件负载过高，有可能会出现节点状态变成可疑或者宕机状态, 同时会伴随分片状态有些变成未复制

图 60: 图 1

出现此现象时，需要排查执行语句和执行任务

3.8.2.1 语句

选择语句

延时时间 > 30 秒以上的语句，需要进行执行计划查看，里面涉及 scan 全表扫描等操作的需要应用层进行 sql 优化。可以通过命令查询正在执行的 sql

```
SELECT * FROM [ SHOW QUERIES ] order by start limit 22;
```

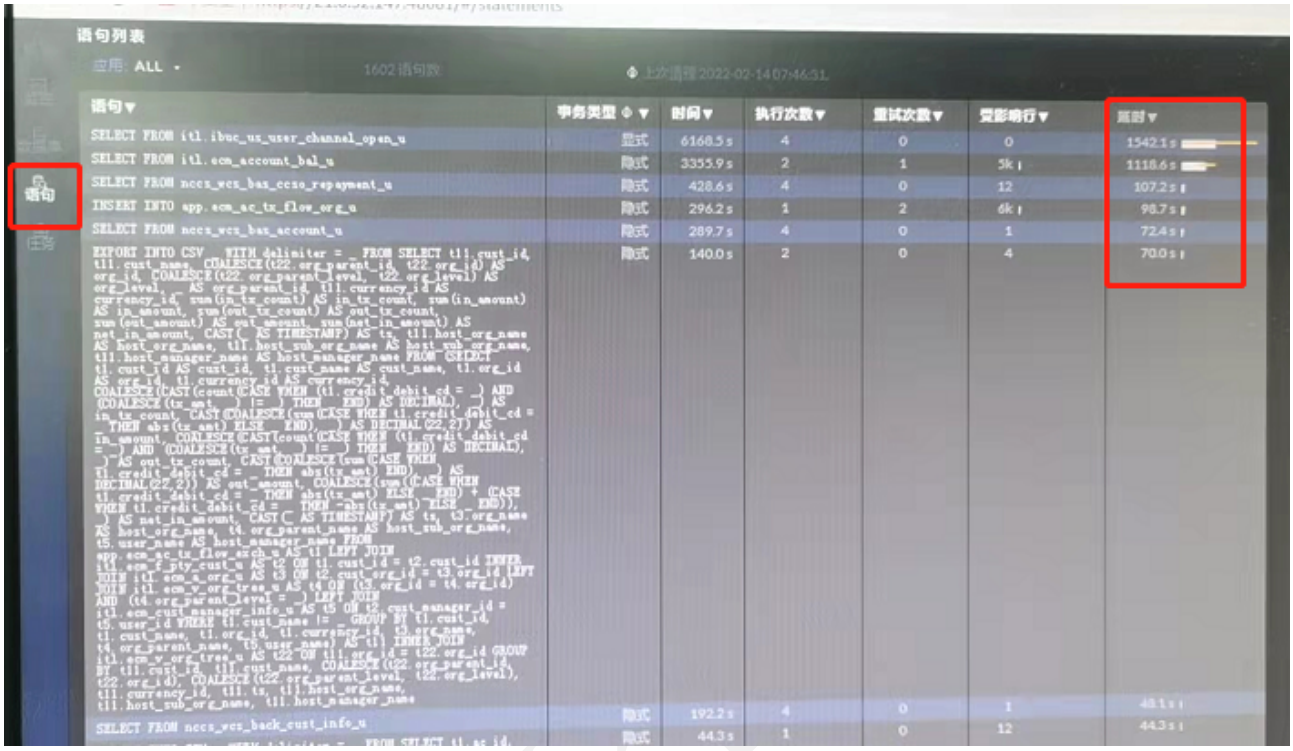

语句列表
应用: ALL
1602 语句数
上次请求 2022-02-14 07:46:31
语句
语句▼
事务类型▼
时间▼
执行次数▼
重试次数▼
受影响行▼
耗时▼
SELECT FROM itl.ibuc_us_user_channel_open_u
显式
6168.5 s
4
0
0
1542.1 s
SELECT FROM itl.ecm_account_bal_u
隐式
3355.9 s
2
1
5k
1118.6 s
SELECT FROM nccs_wcs_bas_ccss_repayment_u
隐式
428.6 s
4
0
12
107.2 s
INSERT INTO app.ecm_ac_tx_flow_org_u
隐式
296.2 s
1
2
6k
98.7 s
SELECT FROM nccs_wcs_bas_account_u
隐式
289.7 s
4
0
1
72.4 s
隐式
140.0 s
2
0
4
70.0 s
SELECT FROM nccs_wcs_back_cust_info_u
隐式
192.2 s
4
0
1
48.1 s
隐式
44.3 s
1
0
12
44.3 s

图 61: 图 2

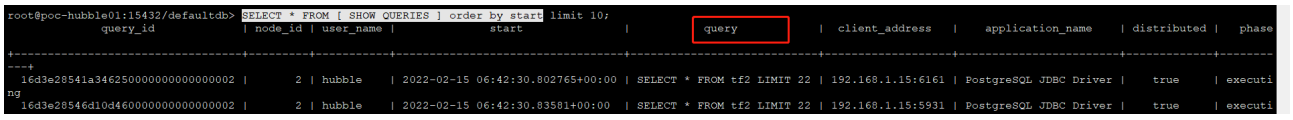

root@poc-hubble01:15432/defaultdb> SELECT * FROM [SHOW QUERIES] order by start limit 10;
query_id | node_id | user_name | start | query | client_address | application_name | distributed | phase
16d3e28541a34625000000000000002 | 2 | hubble | 2022-02-15 06:42:30.802765+00:00 | SELECT * FROM tf2 LIMIT 22 | 192.168.1.15:6161 | PostgreSQL JDBC Driver | true | executing
16d3e28546d10d460000000000000002 | 2 | hubble | 2022-02-15 06:42:30.83581+00:00 | SELECT * FROM tf2 LIMIT 22 | 192.168.1.15:5931 | PostgreSQL JDBC Driver | true | executi

图 62: 图 3

可以手动结束这些运行慢的 sql，保障集群正常运行，同时涉及这些 sql 的系统需要进行 sql 优化。

可以结束运行的 sql 命令为: CANCEL QUERY '15d611a6c5a6307f0000000000000001';

详见 hubble 官网 http://hubbledb.cn/docs/3.7/sql/sql-statements/#show-queries

3.8.2.2 任务

选择任务 (运行中的任务大量的资源占用，也有可能导致资源紧张，出现节点可疑)

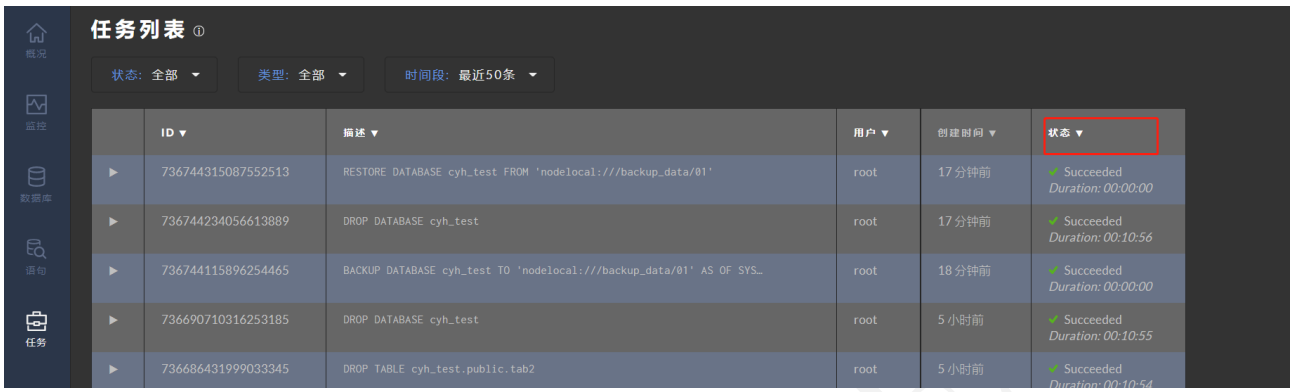


图 63: 图 4

查看状态是否有一直运行且不停顿的任务, 可以通过命令查询正在执行的任务

```
SELECT * FROM [ SHOW JOBS ];
```

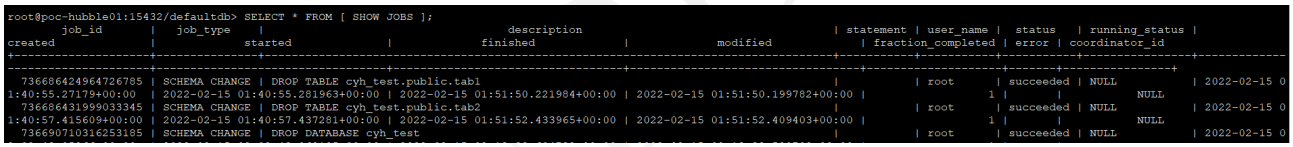


图 64: 图 5

找到运行时间超长的任务，基于现场情况处理，如果执行任务结束

```
CANCEL JOB 736686424964726785;
```

详见 hubble 官网 http://hubbledb.cn/docs/3.10.1/sql/sql-statements/#show-jobs

3.8.2.3 终结办法对可疑/宕机的节点进行重启操作

> 一个实例一个实例轮询执行操作

数据库状态

```
systemctl status hubble
```

关闭数据库

```
sudo systemctl stop hubble
```

启动数据库

```
sudo systemctl start hubble
```

3.8.3 服务器节点故障

当服务器出现宕机或者磁盘等硬件或者软件故障时，服务器无法提供服务时，需要下线此节点上的数据库实例

```
hubble node status --certs-dir=/var/lib/hubbletp/certs --host=poc-hubble01:15432
```

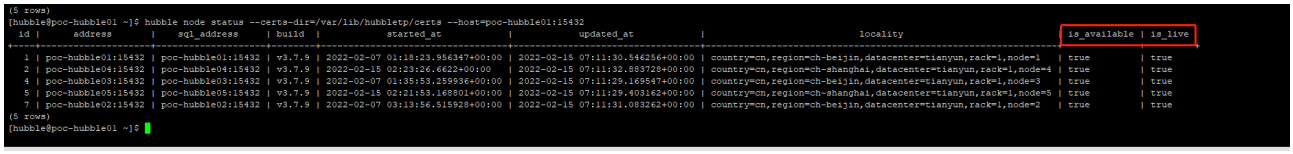


图 65: 图 6

退役节点

```
hubble node decommission 1 --certs-dir=/var/lib/hubbletp/certs --host=poc-
    ↪ hubble01:15432
```

3.9 时区调整

hubble 时区设置为客户端 session 级别设置

- 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)
```

3.9.1 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
```

3.9.2 在 Dbeaver 设置时区

- 在配置之中添加以下语句

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

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

3.9.3 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)
```

将当前时间返回前台

> 调整时区后，截取当前的时间，返回给前台，若不进行时区调整将会有时间差用 now() 函数返回给当前时间的时候，要引用当前时区 (比如:shanghai)

- 调整前 (设置时区前，错误的形式)

```
root@poc-hubble01:15432/defaultdb> select  to_char((now() :: timestamp without
    ↪ time zone),'%Y-%m-%d %H:%M');
      to_char
+------------------+
  2022-04-26 07:34
(1 row)
```

HUBBLE

- 调整后 (设置时区后，正确的形式)

```
root@poc-hubble01:15432/defaultdb> select  to_char((now() :: timestamp without
    ↪ time zone),'%Y-%m-%d %H:%M');
      to_char
+------------------+
  2022-04-26 15:35
(1 row)

Time: 449.134µs
```

4 安全

4.1 审计日志

4.1.1 所需权限

只有admin角色的成员才能在表上启用审计日志。默认情况下，root用户属于admin角色。

4.1.2 参数

参数	描述
table_name	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计日志文件中。
WRITE	将所有表写入记录到审计日志文件。
OFF	关闭审计日志。

4.1.3 审计日志文件存储位置

在 hubble.service 文件中，通过配置 --sql-audit-dir=/data/hubbledir/auditdir 来指定审计日志目录

4.1.4 开启审计日志

默认关闭，要开启审计日志，请执行以下命令

```
> ALTER TABLE accounts EXPERIMENTAL_AUDIT SET READ WRITE;
```

添加测试数据

```
> insert into accounts values(123,'xuejs',123.45);
```

在配置的审计日志目录中，查看 hubble-sql-audit.log 日志内容

> 开启审计日志后读写后，向表中写入数据和查询数据的操作都会记录在审计日志中，插入错误的操作也会记录。

```
I210805 08:54:40.205150 245942380 sql/exec_log.go:168  [n1,client
    ↪ =192.168.1.11:13566,user=root] 5 exec "$ hubble sql" {"accounts"[319]:
    ↪ READWRITE} "ALTER TABLE accounts EXPERIMENTAL_AUDIT SET READ WRITE" {}
    ↪ 6.712 0 OK 0
```

HUBBLE

```
I210805 08:54:43.563678 245942380 sql/exec_log.go:168  [n1,client
    ↪ =192.168.1.11:13566,user=root] 6 exec "$ hubble sql" {"accounts"[319]:
    ↪ READWRITE} "INSERT INTO accounts VALUES (123, 'xuejs', 123.45)" {} 1.720 0
    ↪  ERROR 0
I210805 08:54:58.675654 245942380 sql/exec_log.go:168  [n1,client
    ↪ =192.168.1.11:13566,user=root] 7 exec "$ hubble sql" {"accounts"[319]:
    ↪ READWRITE} "INSERT INTO accounts VALUES (123456, 'xuejs', 523.45)" {}
    ↪ 1.270 1 OK 0
I210805 08:55:06.900166 245942380 sql/exec_log.go:168  [n1,client
    ↪ =192.168.1.11:13566,user=root] 8 exec "$ hubble sql" {"accounts"[319]:READ
    ↪ } "SELECT * FROM accounts" {} 1.142 4 OK 0
```

4.1.5 关闭审计日志

要关闭审计日志，请使用以下命令

```
> ALTER TABLE accounts EXPERIMENTAL_AUDIT SET OFF;
```

在配置的审计日志目录中，查看 hubble-sql-audit.log 日志内容

```
I210805 08:53:08.544619 245942380 sql/exec_log.go:168  [n1,client
    ↪ =192.168.1.11:13566,user=root] 4 exec "$ hubble sql" {"accounts"[319]:READ
    ↪ , "accounts"[319]:READWRITE} "ALTER TABLE accounts EXPERIMENTAL_AUDIT SET
    ↪ OFF" {} 6.126 0 OK 0
```

4.2 更新安全证书

4.2.1 说明

hubble 集群在一定时期内需要进行证书的替换，以保证不影响后续的正常使用，允许您轮换安全证书而无需重新启动节点。

4.2.2 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。

4.2.3 操作步骤

4.2.3.1 查看证书信息

- 查看证书时效性，使用cert list语句 (使用 root 用户)

```
/usr/local/bin/hubble cert list --certs-dir=/var/lib/hubble/certs
```

HUBBLE

```
  Usage  | Certificate File |    Key File     |  Expires   |              Notes                 | Error
+--------+------------------+-----------------+------------+------------------------------------+-------+
  CA     | ca.crt           |                 | 2032/06/07 | num certs: 1                       |
  Node   | node.crt         | node.key        | 2027/06/04 | addresses: hubble01,192.168.100.11 |
  Client | client.root.crt  | client.root.key | 2027/06/04 | user: root                         |
(3 rows)
```

图 66: 图 1

4.2.3.2 备份证书文件

- 进入对应证书目录，进行备份 (使用 root 用户)

```
cd /var/lib/hubble
```

```
cp -r certs/ certs_bak
```

```
cp -r mysafedirectory/ mysafedirectory_bak
```

4.2.3.3 关闭服务

- 关闭服务命令 (使用 hubble 用户)

```
systemctl stop hubble
```

> 针对所有需要换证书的节点，都要关闭服务

4.2.3.4 数据库 license 注册激活

> 本步骤通过 root 用户操作

4.2.3.4.1 生成 license 机器码

- 选取任意一个节点，使用`license machinecode`语句

```
cd /var/lib/hubble
```

```
/usr/local/bin/hubble  license  machinecode --certs-dir=/var/lib/hubble/certs
```

> 执行后会在/var/lib/hubble/certs 下生成 BDPMACHINE.txt 文件

4.2.3.4.2 license 注册激活

- 将/var/lib/hubble/certs/BDPMACHINE.txt 文件机器码发送到公司申请 license 激活码
- 将 license 激活码写入到/var/lib/hubble/certs/BDPLIC.txt 文件 (**必须保证 BDPLIC.txt 文件正确命名**)
- 执行 license 注册激活，使用`license register`语句

```
/usr/local/bin/hubble license register     --certs-dir=/var/lib/hubble/certs
```

```
 AppNum | ModuleCode | AppName | EffectTime | Validate
--------+------------+---------+------------+----------
      4 |       9999 | HUBBLE  | 2022-6-2   |       90
(1 row)
```

图 67: 图 2

4.2.3.5 证书更新

- 在第一台服务器生成证书，使用cert create-ca语句 (使用 hubble 用户)

```
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cert create-node语句 (使用 hubble 用户)

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

- 分发至其他节点 (使用 hubble 用户)

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

> 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cert create-node语句 (使用 hubble 用户)

```
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cert create-client语句 (使用 hubble 用户)

```
cd /var/lib/hubble/
hubble cert create-client root --certs-dir=certs --ca-key=mysafedirectory/ca.key
    ↪  --overwrite  #root的，也可以生成其他用户的
```

> 如果在执行 create-client 客户端授权时，有的 license 申请期限时常较少，会出现”Error: failed to generate client certificate and key: xxx key: CA lifetime is 2364.193159h, shorter than the requested 43920.000000h. Renew CA certificate, or rerun with `--lifetime=2364h` for a shorter duration” 需要后面添加`--lifetime=2364h`参数，执行命令 hubble cert create-client root --certs-dir=certs --ca-key=mysafedirectory/ca.key --overwrite `--lifetime=2364h`

> 所有用到的节点都需要执行一次

- 再一次查看证书信息 (root 用户)

```
/usr/local/bin/hubble cert list --certs-dir=/var/lib/hubble/certs
```

说明: 查看失效日期的变化

4.2.3.6 重启服务

- 启动服务 (使用 hubble 用户)

```
systemctl start  hubble
```

> 所有用到的节点都重启

4.2.3.7 连接测试

- 连接数据库

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

- 查询数据，进行测试

```
show databases
```

5 优化

5.1 索引

索引通过帮助 SQL 无需遍历表快速定位数据，从而提高数据库的性能。

5.1.1 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，然后对它们的值进行排序（而不对表本身的值进行排序）。

列索引建立好后，SQL 可以便捷得使用索引来过滤值，而不需要对表进行一行行扫描。在一张大表中，这极大减少了 SQL 语句执行需要用到的行，执行查询的速度会指数级增长。

例如，如果您为INT列建立索引，SQL 通过WHERE <indexed column> = 10子句进行过滤，就可以使用索引查找从 10 开始但小于 11 的值。如果没有索引，SQL 将必须对表中的每一行等于 10 的值进行统计。这也称为全表扫描，这可能会降低查询性能。

5.1.1.1 索引创建

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性能。

primary索引有助于过滤表的主键，但对于查找其他列并没有帮助。不过可以使用二级索引来提高查询的性能，您可以创建它们:

- 在CREATE TABLE建表时的INDEX子句中使用。除了明确定义的索引之外，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。

HUBBLE

- 使用ALTER TABLE添加UNIQUE约束，它会为约束列自动创建索引。

如何创建更有用的二级索引，你可以查看我们的最佳实践。

5.1.1.2 索引选择

由于一个表中只能使用一个索引，因此 Hubble 选择它计算时需要扫描最少行的索引（即最快的行）。

你还可以使一个查询强制使用一个指定的索引，来覆盖 Hubble 选择的索引。SELECT，DELETE和UPDATE语句支持指定的索引。

5.1.1.3 索引存储

Hubble 将索引直接存储在键值对中。

5.1.1.4 性能

索引是一种权衡的选择：它极大提高了查询速度，但会稍微降低写入的速度（因为必须复制和对新值排序）。

5.1.2 最佳实践

我们建议为所有常用查询创建索引。通过对每个查询的 WHERE 和SELECT子句来设计最有用的索引，可以创建以下索引：

- 索引WHERE子句中的所有列。
- 存储仅在SELECT子句中的列。
- 出现回表查询时，优化相关索引

5.1.2.1 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（称为primary）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。

此功能还可以避免使用单列索引，使用该列作为多列索引中的第一列，这对于更多查询很有用。

- 在 WHERE 子句中用等号运算符（=或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。
- 对于顺序不同，列相同的索引可以为每个查询成生不同的结果。

5.1.2.2 存储字段

STORING子句不属于索引，但存储在索引中。这优化了查询这些列而不对它们进行过滤，它避免了读取主索引。

5.1.2.3 示例

TPC-C 标准测试自动生成的表order的建表如下：

```
CREATE TABLE order (
    o_id integer NOT NULL,
    o_d_id integer NOT NULL,
    o_w_id integer NOT NULL,
```

```
    o_c_id integer NULL,
    o_entry_d TIMESTAMP NULL,
    o_carrier_id integer NULL,
    o_ol_cnt integer NULL,
    o_all_local integer NULL,
    CONSTRAINT "primary" PRIMARY KEY (o_w_id ASC, o_d_id ASC, o_id DESC),
    CONSTRAINT fk_o_w_id_ref_customer FOREIGN KEY (o_w_id, o_d_id, o_c_id)
       ↪ REFERENCES customer(c_w_id, c_d_id, c_id),
    UNIQUE INDEX order_idx (o_w_id ASC, o_d_id ASC, o_c_id ASC, o_id DESC),
    FAMILY "primary" (o_id, o_d_id, o_w_id, o_c_id, o_entry_d, o_carrier_id,
       ↪ o_ol_cnt, o_all_local)
)
```

例如，SQL 对索引列进行筛选但检索未定义索引的列o_ol_cnt，可以通过使用索引order_idx筛选出o_d_id> 5 且 o_w_id> 5 且 o_c_id>2000 的值，通过index join从主索引读取o_ol_cnt。

```
explain select o_ol_cnt from order 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@primary
            scan   |             |
                   | table       | order@order_idx
                   | spans       | /6/6/2001-
                   | filter      | (o_d_id > 5) AND (o_c_id > 2000)
(9 rows)
```

但是，如果没有索引，SQL 将必须对表中的每一行进行统计，这也称为全表扫描，这会降低查询性能。

```
CREATE TABLE order (
    o_id integer NULL,
    o_d_id integer NULL,
    o_w_id integer NULL,
    o_c_id integer NULL,
    o_entry_d TIMESTAMP NULL,
    o_carrier_id integer NULL,
    o_ol_cnt integer NULL,
    o_all_local integer NULL,
    FAMILY "primary" (o_id, o_d_id, o_w_id, o_c_id, o_entry_d, o_carrier_id,
       ↪ o_ol_cnt, o_all_local, rowid)
)
```

HUBBLE

```
explain select o_ol_cnt from order 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@primary
             | spans       | ALL
             | filter      | ((o_d_id > 5) AND (o_w_id > 5)) AND (o_c_id > 2000)
(7 rows)
```

但是如果我们将o_ol_cnt通过STORING存储在索引中，则不再需要索引联接index join。这意味着我们的查询仅需要从二级索引中读取，因此效率更高。

```
CREATE TABLE order (
    o_id integer NOT NULL,
    o_d_id integer NOT NULL,
    o_w_id integer NOT NULL,
    o_c_id integer NULL,
    o_entry_d TIMESTAMP NULL,
    o_carrier_id integer NULL,
    o_ol_cnt integer NULL,
    o_all_local integer NULL,
    CONSTRAINT "primary" 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),
    FAMILY "primary" (o_id, o_d_id, o_w_id, o_c_id, o_entry_d, o_carrier_id,
       ↪ o_ol_cnt, o_all_local)
)
```

```
explain select o_ol_cnt from order 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@order_idx
             | spans       | /6/6/2001-
             | filter      | (o_d_id > 5) AND (o_c_id > 2000)
(7 rows)
```

HUBBLE

5.2 向量引擎

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。

5.2.1 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。
experimental_on	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。
off	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

5.2.2 支持的数据类型

- STRING
- UUID
- BYTES
- DATE
- FLOAT
- INT
- BOOL
- TIMESTAMP

5.2.3 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：

HUBBLE

```
show vectorize_row_count_threshold;

vectorize_row_count_threshold
+-----------------------------+
  1000
(1 row)
```

```
set vectorize_row_count_threshold=10000;
```

5.3 开发使用规范

5.3.1 开发设计

5.3.1.1 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。

5.3.1.2 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。

5.3.1.3 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。

5.3.1.4 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。

5.3.1.5 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。

5.3.1.6 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。

5.3.1.7 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。

5.3.1.8 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。

5.3.1.9 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。强烈推荐手动执行建表语句然后执行`insert into T2 select * from T1`方式导入数据。

5.3.2 SQL 规范建议

5.3.2.1 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。

5.3.2.2 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;
```

5.3.2.3 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。

5.3.2.4 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。

5.3.2.5 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

5.3.2.6 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

5.3.2.7 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。

HUBBLE

```
CREATE TABLE t_t2 (
    fundid INT8 NOT NULL,
    trans_amount decimal(18,2) NULL,
    data BYTES NULL,
    CONSTRAINT "primary" PRIMARY KEY (id ASC),
    FAMILY f1 (fundid, trans_amount),
    FAMILY f2 (data)
)
```

5.3.2.8 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。

5.3.2.9 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。

5.3.2.10 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。

5.3.2.11 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。

5.3.2.12 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。

5.3.3 部署规范

5.3.3.1 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。

HUBBLE

6 导入

6.1 csv 导入

6.1.1 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 option 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（,默认情况下）
双引号（"）
换行符（\n）
回车（\r）
```

4. 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”
5. 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。

6.1.2 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
http	http	主机地址	N/A	http://localhost:8080/mydatest.csv
NFS/Local	nodelocal	节点 ID 或为空	N/A	nodelocal:///path/mydatest,nodelocal:///path/mydatest2

6.1.3 步骤三：将数据迁移到 Hubble 数据库中

6.1.3.1 hubble 库中不存在的表的数据迁移

```
IMPORT TABLE employees (
    emp_no INT PRIMARY KEY,
    birth_date DATE NOT NULL,
    first_name STRING NOT NULL,
    last_name STRING NOT NULL,
    gender STRING NOT NULL,
    hire_date DATE NOT NULL
  ) CSV DATA ('http://localhost:3000/employees.csv');
```

```
root@tytest16:26257/hubble_demo> IMPORT TABLE employees (
                              ->     emp_no INT PRIMARY KEY,
```

HUBBLE

```
                                ->     birth_date DATE NOT NULL,
                                ->     first_name STRING NOT NULL,
                                ->     last_name STRING NOT NULL,
                                ->     gender STRING NOT NULL,
                                ->     hire_date DATE NOT NULL
                                ->   ) CSV DATA ('http://localhost:3000/employees.
                                   ↪ csv');
        job_id        |  status   | fraction_completed |  rows  | index_entries |
           ↪  system_records |  bytes
+--------------------+-----------+--------------------+--------+---------------+----------
   ↪
  535422568764833793 | succeeded |                  1 | 300024 |             0 |
     ↪                0 | 12134341
(1 row)

Time: 3.078573146s
```

6.1.3.2 hubble 库中已存在表的数据迁移

```
IMPORT INTO employees (emp_no, birth_date,first_name,last_name,gender,hire_date)
   ↪  CSV DATA ('http://localhost:3000/employees.csv')
```

6.1.3.3 多文件导入

```
IMPORT TABLE employees (
    emp_no INT PRIMARY KEY,
    birth_date DATE NOT NULL,
    first_name STRING NOT NULL,
    last_name STRING NOT NULL,
    gender STRING NOT NULL,
    hire_date DATE NOT NULL
    )csv data(
'http://192.168.100.101/tpdata1106/t05_on_board_order_evt_1y/02/xaa',
'http://192.168.100.101/tpdata1106/t05_on_board_order_evt_1y/02/xab',
'http://192.168.100.101/tpdata1106/t05_on_board_order_evt_1y/02/xac',
'http://192.168.100.101/tpdata1106/t05_on_board_order_evt_1y/02/xad',
'http://192.168.100.101/tpdata1106/t05_on_board_order_evt_1y/02/xae',
'http://192.168.100.101/tpdata1106/t05_on_board_order_evt_1y/02/xaf'
) with nullif = '\N',delimiter = e'|';
```

6.1.4 配置选项

以下选项可用于 IMPORT ... CSV：

6.1.4.1 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：

HUBBLE

```
IMPORT TABLE employees (
    emp_no INT PRIMARY KEY,
    birth_date DATE NOT NULL,
    first_name STRING NOT NULL,
    last_name STRING NOT NULL,
    gender STRING NOT NULL,
    hire_date DATE NOT NULL
  )
  CSV DATA ('http://localhost:3000/employees.csv') WITH delimiter = e'\t';
```

```
root@tytest16:26257/hubble_demo> IMPORT INTO employees (emp_no, birth_date,
   ↪ first_name,last_name,gender,hire_date) CSV DATA ('http://localhost:3000/
   ↪ employees.csv');
        job_id       |  status   | fraction_completed |  rows  | index_entries |
           ↪  system_records |  bytes
+--------------------+-----------+--------------------+--------+---------------+----------
   ↪
  535428382456872961 | succeeded |                  1 | 300024 |             0 |
     ↪                0 | 12134341
(1 row)

Time: 5.768633441s
```

6.1.4.2 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 TABLE employees (
    emp_no INT PRIMARY KEY,
    birth_date DATE NOT NULL,
    first_name STRING NOT NULL,
    last_name STRING NOT NULL,
    gender STRING NOT NULL,
    hire_date DATE NOT NULL
  )
  CSV DATA ('http://localhost:3000/employees.csv') WITH  comment = '#';
```

6.1.4.3 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 TABLE employees (
    emp_no INT PRIMARY KEY,
    birth_date DATE NOT NULL,
    first_name STRING NOT NULL,
```

HUBBLE

```
    last_name STRING NOT NULL,
    gender STRING NOT NULL,
    hire_date DATE NOT NULL
  )
  CSV DATA    ('http://localhost:3000/employees.csv') WITH skip = '2';
```

6.1.4.4 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 TABLE employees (
    emp_no INT PRIMARY KEY,
    birth_date DATE NOT NULL,
    first_name STRING NOT NULL,
    last_name STRING NOT NULL,
    gender STRING NOT NULL,
    hire_date DATE NOT NULL
  )
  CSV DATA ('http://localhost:3000/employees.csv') WITH nullif = '';
```

6.1.4.5 文件压缩格式

compress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 TABLE employees (
    emp_no INT PRIMARY KEY,
    birth_date DATE NOT NULL,
    first_name STRING NOT NULL,
    last_name STRING NOT NULL,
    gender STRING NOT NULL,
    hire_date DATE NOT NULL
  )
  CSV DATA    ('http://localhost:3000/employees.csv.gz') WITH compress = 'gzip';
```

6.2 MySQL 导入

6.2.1 MySQL 单表迁移至 Hubble

6.2.1.1 步骤一：导出需要迁移的表

```
mysqldump -uroot -p123456  sysbench sbtest1_500 > /opt/willynew/newhubble/mysql.
    ↪ sql
```

6.2.1.2 步骤二：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。可使用 httpd 服务。

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
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:///path/mydatest,nodelocal://2/path/mydatest

6.2.1.3 步骤三：导入 MySQL 导出的文件

6.2.1.3.1 mysql 导出单个表

```
mysqldump -uhadoop -p  hilbert_3  vertexlable >  /data/path/vertexlable_test.sql ;
```

6.2.1.3.2 把数据导入 hubble 中

```
create database if not exists hubble_demo;

use hubble_demo;

IMPORT MYSQLDUMP 'http://localhost:3000/vertexlable_test.sql';
```

```
IMPORT MYSQLDUMP 'http://localhost:3000/vertexlable_test.sql';

        job_id        | status    | fraction_completed | rows | index_entries | system_records | bytes
+--------------------+-----------+--------------------+------+---------------+----------------+-------
  535405586715115521 | succeeded |                  1 |    5 |            10 |              0 |  1152
(1 row)
```

6.2.2 MySQL 整库迁移至 Hubble

6.2.2.1 步骤一：导出需要迁移的整库

```
mysqldump -uhadoop -p  hilbert_3  > /data/path/hilbert_v3.sql;
```

6.2.2.2 步骤二：将导出的数据文件放置于集群可访问到的位置

如同 mysql 单表导入的步骤二操作一致

HUBBLE

6.2.2.3 步骤三：导入 MySQL 导出的整库数据

用户导出了整个数据库，可以通过以下的 IMPORT 语句从导出文件中读取全部数据和 DDL 语句

```
create database if not exists hubble_demo;

use  hubble_demo;

import mysqldump 'http://localhosts:3000/hilbert_v3.sql';
```

```
import mysqldump 'http://localhost:3000/hilbert_v3.sql';

        job_id       |  status   | fraction_completed | rows | index_entries |
           ↪ system_records | bytes
+--------------------+-----------+--------------------+------+---------------+------------
    ↪
  535411573412855809 | succeeded |                  1 |   45 |             1 |
     ↪              0 |  2101
(1 row)

Time: 19.386075748s
```

6.3 Oracle 导入

6.3.1 步骤 1: 导出 Oracle 模式

```
expdp user/password directory=datapump dumpfile=oracle_example.dmp content=
   ↪ metadata_only logfile=example.log
```

6.3.2 步骤 2: 将 Oracle 模式转换为 SQL

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

6.3.3 步骤 3: 导出表数据

您需要将每个表的数据提取到数据列表文件（.lst）中。我们编写了一个简单的 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
```

HUBBLE

```
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
```

要提取数据，我们为 SQL * Plus 中的每个表运行了脚本：

```
$ sqlplus user/password
```

```
@spool CUSTOMERS
  @spool ADDRESSES
  @spool CARD_DETAILS
  @spool WAREHOUSES
  @spool ORDER_ITEMS
  @spool ORDERS
  @spool INVENTORIES
  @spool PRODUCT_INFORMATION
  @spool LOGON
  @spool PRODUCT_DESCRIPTIONS
  @spool ORDERENTRY_METADATA
```

.lst 为每个表创建一个带有前导和尾随空格的数据列表文件 ()。

退出 SQL * Plus：

```
>EXIT
```

6.3.4 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:
```

HUBBLE

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

```
python fix-example.py CUSTOMERS.lst ADDRESSES.lst CARD_DETAILS.lst WAREHOUSES.
    ↪ lst ORDER_ITEMS.lst ORDERS.lst INVENTORIES.lst PRODUCT_INFORMATION.lst
    ↪ LOGON.lst PRODUCT_DESCRIPTIONS.lst ORDERENTRY_METADATA.lst
```

6.3.5 步骤 5:Oracle 映射到 hubble 数据类型

使用之前创建的 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 语句以包括主键。

6.3.5.1 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 1 个
CHAR(n)，CHARACTER(n)n <256	CHAR(n)，CHARACTER(n)
CLOB	STRING 1 个
DATE	DATE
FLOAT(n)	DECIMAL(n)
INTERVAL YEAR(p) TO MONTH	VARCHAR，INTERVAL
INTERVAL DAY(p) TO SECOND(s)	VARCHAR，INTERVAL
JSON	JSON 2
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 3
NUMBER(p,0)，NUMBER(p)5 <= p <9	INT4 3
NUMBER(p,0)，NUMBER(p)9 <= p <19	INT8 3
NUMBER(p,0)，NUMBER(p)19 <= p <= 38	DECIMAL(p)
NUMBER(p,s) s> 0	DECIMAL(p,s)
NUMBER，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)
XML	JSON 2

1. 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 1 MB 以

HUBBLE

下，以确保性能。1 MB 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。

2. 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。
3. 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。（可选）NUMBERS 可以转换为 DECIMAL。

6.3.6 步骤 6: 数据导入

如同 CSV 数据迁移一致

6.4 Postgesql 导入

6.4.1 单表迁移

6.4.1.1 步骤一：导出需要迁移的表

```
pg_dump -t sysbench sbtest1_500 > /opt/willynew/newhubble/pg.sql
```

6.4.1.2 步骤二：将导出的数据文件放置于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
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:///path/mydatest,nodelocal://2/path/mydatest`

6.4.1.3 步骤三：导入 PostgreSQL 导出的整库数据

```
create database if not exists sysbench;

use sysbench;

import pgdump 'http://localhost:3000/pg.sql';
```

6.4.2 PostgreSQL 整库迁移至 Hubble

6.4.2.1 步骤一：导出需要迁移的整库

```
pg_dump sysbench> /opt/willynew/newhubble/sysbenchdb.sql
```

6.4.2.2 步骤二：将导出的数据文件放置于集群可访问到的位置

6.4.2.3 步骤三：导入 PostgreSQL 导出的整库数据

```
create database if not exists sysbench;
use sysbench;
```

HUBBLE

```
import pgdump 'http://localhost:3000/sysbenchdb.sql';
```

6.5 DB2 导入

6.5.1 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
```

6.5.2 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
```

HUBBLE

6.5.3 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

HUBBLE

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
- 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));
```

6.5.4 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
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:///path/mydatest,nodelocal://n/path/mydatest`

`nodelocal://n/path/mydatest`，n 是节点数，不能省略，地址这块与 3.7 版本是有区别的。

HUBBLE

6.5.5 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 = ','
;
```

```
root@poc-hubble01:35432/oracle> 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

```
root@poc-hubble01:35432/oracle> 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

7 开发文档

7.1 go 编码指南

本编码指南的灵感来源于Uber-Go 风格指南，并对 hubble 进行了专门的修改。

7.1.1 介绍

这些准则用来规范我们的代码的惯例，这些约定已经超出了 gofmt 中规定的内容。

本指南的目标是通过详细描述在 HUBBLE 中编写 Go 代码的注意事项。这些规则的存在是为了保持代码库的可管理性，同时仍然允许工程师有效地使用 Go 语言特性。

HUBBLE

这记录了我们在 CRL 中遵循的 Go 代码中的惯用约定。其中许多是 Go 的一般指导原则，而其他的则依赖于外部资源：

1. effective go
2. The Go common mistakes guide

当运行 golint 和 go vet 时，所有代码都应该是无错误的。我们建议您将编辑器设置为：

- 在保存时运行 crlfmt
- 在检查错误时运行 make lintshort

您可以在“Go 工具编辑器支持”中找到相关信息：golang/go/wiki/IDEsAndTextEditorPlugins

7.1.2 准则

7.1.2.1 指向接口的指针

几乎不需要指向接口的指针。您应该将接口作为值传递，底层数据仍然可以是指针。

一个接口由两部分组成：

1. 指向特定类型信息的指针。你可以认为这是”type”
2. 数据指针。如果存储的数据是指针，则直接存储。如果存储的数据是值，则存储指向该值的指针。

如果希望接口方法修改基础数据，则必须使用指针。

7.1.2.2 接收器和接口

一个接收值的方法被指针调用可以和值调用一样。

例如，

```
type S struct {
  data string
}

func (s S) Read() string {
  return s.data
}

func (s *S) Write(str string) {
  s.data = str
}

sVals := map[int]S{1: {"A"}}

// You can only call Read using a value
sVals[1].Read()

// This will not compile:
//  sVals[0].Write("test")

sPtrs := map[int]*S{1: {"A"}}
```

HUBBLE

```
// You can call both Read and Write using a pointer
sPtrs[1].Read()
sPtrs[1].Write("test")
Similarly, an interface can be satisfied by a pointer, even if the method has a
    ↪ value receiver.

type F interface {
  f()
}

type S1 struct{}

func (s S1) f() {}

type S2 struct{}

func (s *S2) f() {}

s1Val := S1{}
s1Ptr := &S1{}
s2Val := S2{}
s2Ptr := &S2{}

var i F
i = s1Val
i = s1Ptr
i = s2Ptr

// The following doesn't compile, since s2Val is a value, and there is no value
    ↪ receiver for f.
//   i = s2Val
```

HUBBLE

请参考: pointers_vs_values

7.1.2.3 零值互斥锁是有效的

sync.Mutex 和 sync.RWMutex 的零值是有效的，因此不需要指向互斥锁对象的指针。

错误	正确
mu := new(sync.Mutex)mu.Lock()	var mu sync.Mutexmu.Lock()

如果结构体使用指针，那么互斥锁可以是非指针字段，或者最好是直接嵌入到结构中。

- 嵌入为私有类型或需要实现互斥锁接口的类型。

```
type smap struct {
  sync.Mutex
```

```
  data map[string]string
}

func newSMap() *smap {
  return &smap{
    data: make(map[string]string),
  }
}

func (m *smap) Get(k string) string {
  m.Lock()
  defer m.Unlock()

  return m.data[k]
}
```

- 为导出类型使用私有锁

```
type SMap struct {
  mu sync.Mutex

  data map[string]string
}

func NewSMap() *SMap {
  return &SMap{
    data: make(map[string]string),
  }
}

func (m *SMap) Get(k string) string {
  m.mu.Lock()
  defer m.mu.Unlock()

  return m.data[k]
}
```

HUBBLE

7.1.2.4 当需要时拷贝 Map 和 Slices

Slices 和 Map 包含指向底层数据的指针，因此在将它们作为参数传递或作为值返回时，请注意所有权。

另一方面，不要做不必要拷贝 slices, 堆分配是非常昂贵的，尽量避免做这个操作。

接收 Slices 和 Maps

用户可以修改一个作为参数接收的 map 或 slice, 如果存储了它的引用。

如果 API 通过引用捕获一个 slice，请将其记录在函数头中。

错误的代码：

```
func (d *Driver) SetTrips(trips []Trip) {
  d.trips = trips
}

trips := ...
d1.SetTrips(trips)

// 修改 d1.trips?
trips[0] = ...
```

正确的:

```
// SetTrips sets the driver's trips.
// Note that the slice is captured by reference, the
// caller should take care of prevening unwanted aliasing.
func (d *Driver) SetTrips(trips []Trip) { d.trips = trips }

// or:

func (d *Driver) SetTrips(trips []Trip) {
  d.trips = make([]Trip, len(trips))
  copy(d.trips, trips)
}

trips := ...
d1.SetTrips(trips)

// 这时候我们可以修改trips[0]而不会对d1.trips造成影响。
trips[0] = ...
```

HUBBLE

返回 Slices 和 Maps

同样，要小心用户对暴露内部状态的映射或切片的修改。

导出的 API 应该避免返回具有其他访问器的内部状态，例如当它隐藏在互斥锁后面时。

对于“内部”函数有更大的灵活性，尽管函数名应该反映这一点（参见 pkg/util/log 中的 log.closeFileLocked）。

错误的代码:

```
type Stats struct {
  sync.Mutex

  counters map[string]int
}

// Snapshot returns the current stats.
func (s *Stats) Snapshot() map[string]int {
  s.Lock()
  defer s.Unlock()
```

```
  return s.counters
}

// snapshot is no longer protected by the lock!
snapshot := stats.Snapshot()
```

正确的代码:

```
type Stats struct {
  sync.Mutex

  counters map[string]int
}

func (s *Stats) Snapshot() map[string]int {
  s.Lock()
  defer s.Unlock()

  result := make(map[string]int, len(s.counters))
  for k, v := range s.counters {
    result[k] = v
  }
  return result
}

// Snapshot is now a copy.
snapshot := stats.Snapshot()
```

7.1.2.5 延迟清理

使用 defer 清理文件和锁资源

错误的:

```
p.Lock()
if p.count < 10 {
  p.Unlock()
  return p.count
}

p.count++
newCount := p.count
p.Unlock()

return newCount

// easy to miss unlocks due to multiple returns
```

HUBBLE

正确的：

```
p.Lock()
defer p.Unlock()

if p.count < 10 {
  return p.count
}

p.count++
return p.count

// more readable
```

Defer 的开销非常小，只有在能够证明函数执行时间是以纳秒为顺序的情况下才应该避免。使用 defer 可读性比使用 defer 的微小成本要代价要小。这尤其适用于具有比简单内存访问更多的大型方法，其中其他计算比 Defer 成本更高。

7.1.2.6 Channel 的大小为 1 或者 None

通道的大小通常应为 1 或无缓冲。默认情况下，通道是无缓冲的，大小为零。任何其他大小都必须经过高度审查。考虑大小是如何确定的，什么可以防止通道在负载和阻塞写入程序时填满，以及发生这种情况时会发生什么。

错误的：

```
// Ought to be enough for anybody!
c := make(chan int, 64)
```

正确的：

```
// Size of one
c := make(chan int, 1) // or
// Unbuffered channel, size of zero
c := make(chan int)
```

7.1.2.7 枚举类型从 1 开始

在 Go 中引入枚举的标准方法是使用 iota 声明自定义类型和常量组。由于变量的默认值为 0，所以通常应该在非零值上开始枚举。

错误的：

```
type Operation int

const (
  Add Operation = iota
  Subtract
  Multiply
)

// Add=0, Subtract=1, Multiply=2
```

HUBBLE

正确的：

```
type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

// Add=1, Subtract=2, Multiply=3
```

在某些情况下，使用零值是有意义的，例如零值情况是理想的默认行为。

```
type LogOutput int

const (
  LogToStdout LogOutput = iota
  LogToFile
  LogToRemote
)

// LogToStdout=0, LogToFile=1, LogToRemote=2
```

7.1.2.8 Error Types

- errors.New() for errors with simple static strings
- errors.Newf() for formatted error strings
- errors.AssertionFailedf() for assertion/internal errors that should be louder than usual
- Add context on return paths using errors.Wrap()/errors.Wrapf()
- Add user-directed hints with errors.WithHint()

更多请参考 errors 包的 README。

返回错误时，请考虑以下因素以确定最佳选择：

- 这是一个简单的错误，你不需要额外的信息，这是使用 errors.New()
- 当客户端需要检测和处理这个错误，你应该使用 Custom type，并实现 Error() 方法。
- 如果需要由下游方法返回时，传递错误请参考 error 的设计文档。

如果客户端需要检测错误，并且您已经使用 errors.New 创建了一个简单的错误。请对错误使用 var。

错误的：

```
// package foo

func Open() error {
  return errors.New("could not open")
}

// package bar
```

HUBBLE

```
if err := foo.Open(); err != nil {
  if err.Error() == "could not open" {
    // handle
  } else {
    panic("unknown error")
  }
}
```

正确的:

```
// package foo

var ErrCouldNotOpen = errors.New("could not open")

func Open() error {
  return ErrCouldNotOpen
}

// package bar

if err := foo.Open(); err != nil {
  if errors.Is(err, foo.ErrCouldNotOpen) {
    // handle
  } else {
    return errors.Wrap(err, "unknown error")
  }
}
```

如果您有一个客户端可能需要检测的错误，并且希望向该错误添加更多信息（例如，它不是静态字符串），那么您应该使用自定义类型。

错误的:

```
func open(file string) error {
  return fmt.Errorf("file %q not found", file)
}

func use() error {
  if err := open(); err != nil {
    if strings.Contains(err.Error(), "not found") {
      // handle
    } else {
      panic("unknown error")
    }
  }
}
```

正确的:

HUBBLE

```
type errNotFound struct {
  file string
}

func (e errNotFound) Error() string {
  return fmt.Sprintf("file %q not found", e.file)
}

func open(file string) error {
  return errNotFound{file: file}
}

func use() error {
  if err := open(); err != nil {
    var nfErr errNotFound
    if errors.As(err, &nfErr); ok {
      // handle
    } else {
      return errors.Wrap(err, "opening file")
    }
  }
}
```

7.1.2.8.1 Error Wrapping

如果调用失败，有四个主要选项用于传递错误：

- 如果没有要添加的其他上下文，并且希望保留原始错误类型，则返回原始错误。
- 使用 errors.Wrap 添加上下文，以便错误消息提供更多上下文和错误。可以使用 Unwrap 提取原始错误。
- 如果调用方不需要检测或处理特定的错误情况，请使用 errors.Newf。
- 使用 errors.Handle 或 errors.NewAssertionErrorWithWrappedErrf 隐藏原始原因。

建议在可能的情况下添加上下文，这样就不会出现诸如“connection rejected”之类的模糊错误，而是出现诸如“call service foo:connection rejected”之类的更有用的错误。

将上下文添加到返回的错误时，请避免使用“failed to”这样的短语，以保持上下文的简洁性，这些短语表示明显的错误并随着错误在堆栈中的渗透而堆积：

错误的：

返回信息 failed to x: failed to y: failed to create new store: the error

```
s, err := store.New()
if err != nil {
    return errors.Newf(
        "failed to create new store: %s", err)
}
```

正确的：

HUBBLE

返回信息 x: y: new store: the error

```
s, err := store.New()
if err != nil {
    return errors.Wrap(
        "new store", err)
}
```

但是，一旦错误被发送到另一个系统，就应该清楚消息是一个错误（例如日志中的错误标签或“失败”前缀）。

Go 已经有所发展，并在适用的情况下使用 errors.Is() 和 errors.As()。

7.1.2.8.2 Handle Type Assertion Failures

类型断言的单一返回值形式将在不正确的类型上将产生 panic。所以，一定要用“comma ok”这个短语。

错误的:

```
t := i.(string)
```

正确的:

```
t, ok := i.(string)
if !ok {
  // handle the error gracefully
}
```

7.1.2.8.3 避免 Panic

代码在生产模式下必须避免产生 panics。panics 是连锁故障的主要来源，如果发生错误，函数必须返回错误并允许调用方决定如何处理它。

错误的:

```
func foo(bar string) {
  if len(bar) == 0 {
    panic("bar must not be empty")
  }
  // ...
}

func main() {
  if len(os.Args) != 2 {
    fmt.Println("USAGE: foo <bar>")
    os.Exit(1)
  }
  foo(os.Args[1])
}
```

正确的:

```
func foo(bar string) error {
  if len(bar) == 0
```

HUBBLE

```
    return errors.New("bar must not be empty")
  }
  // ...
  return nil
}

func main() {
  if len(os.Args) != 2 {
    fmt.Println("USAGE: foo <bar>")
    os.Exit(1)
  }
  if err := foo(os.Args[1]); err != nil {
    panic(err)
  }
}
```

在大多数情况下，panic/recover 不是一种错误处理策略。只有当发生不可恢复的事情（如 nil 解引用）时，程序才必须死机。

一个例外是程序初始化：程序启动时的错误会导致程序中止，这可能会导致 panic。如`var _statusTemplate = template.Must(template.New("name").Parse("_statusHTML"))`

另一个例外是当包是纯功能的并且没有副作用时。例如，在这种情况下，Hubble 中的部分 SQL 规划使用 panic 传递进行错误处理。

即使在测试中，也要选择 t.Fatal 或者 t.FailNow 而不是 panics，以确保测试被标记为失败。

错误的：

```
// func TestFoo(t *testing.T)

f, err := ioutil.TempFile("", "test")
if err != nil {
  panic("failed to set up test")
}
```

正确的：

```
// func TestFoo(t *testing.T)

f, err := ioutil.TempFile("", "test")
if err != nil {
  t.Fatal("failed to set up test")
}
```

7.1.2.9 性能

特定于性能的指导原则仅适用于 hot path。

7.1.2.9.1 注意 strconv 比 fmt 快

当原语与 string 进行转换时，strconv 比 fmt 快。

错误的：

```
for i := 0; i < b.N; i++ {
  s := fmt.Sprint(rand.Int())
}
```

BenchmarkFmtSprint-4 143 ns/op 2 allocs/op

正确的：

```
for i := 0; i < b.N; i++ {
  s := strconv.Itoa(rand.Int())
}
```

BenchmarkStrconv-4 64.2 ns/op 1 allocs/op

7.1.2.9.2 避免字符串到字节的转换

不要重复地从固定字符串创建 byte slices：它会导致堆分配和副本。相反，执行一次转换并捕获结果。

错误的：

```
for i := 0; i < b.N; i++ {
  w.Write([]byte("Hello world"))
}
```

BenchmarkBad-4 50000000 22.2 ns/op

正确的：

```
data := []byte("Hello world")
for i := 0; i < b.N; i++ {
  w.Write(data)
}
```

BenchmarkGood-4 500000000 3.25 ns/op

7.1.2.10 代码风格

7.1.2.10.1 Line Length

格式化代码，假设它将在 100 列宽的窗口中读取。将代码包装为 100 个字符，将注释包装为 80 个字符，除非这样做会降低代码的可读性。这些值假定 tab width 为 2 个字符。

7.1.2.10.2 Wrapping Function Signatures

当包装不适合一行的函数声明时，将名称、参数和返回类型放在单独的行上，并在与“func”相同的缩进处加上结束符（这有助于直观地将缩进的参数与缩进的函数体分开）。例子：

```
func (s *someType) myFunctionName(
arg1 somepackage.SomeArgType, arg2 int, arg3 somepackage.SomeOtherType,
) (somepackage.SomeReturnType, error) {
...
```

HUBBLE

```
}
```

如果参数列表过长，一个参数一行：

```
func (s *someType) myFunctionName(
    arg1 somepackage.SomeArgType,
    arg2 int,
    arg3 somepackage.SomeOtherType,
) (somepackage.SomeReturnType, error) {
...
}
```

当返回类型需要换行时，遵循相同的规则：

```
func (s *someType) myFunctionName(
  arg1 somepackage.SomeArgType, arg2 somepackage.SomeOtherType,
) (
  somepackage.SomeReturnType,
  somepackage.SomeOtherType,
  error,
) {
...
}
```

一个例外是为连续参数省略重复类型时：短参数和相关参数（例如 “start，end int64”）应位于同一行，或者该类型应在每一行上重复--任何参数都不应单独出现在没有类型的行上（编辑时易混淆且易碎）。

7.1.2.10.3 相似声明分组

Go supports grouping similar declarations.

错误的：

```
import "a"
import "b"
```

正确的：

```
import (
  "a"
  "b"
)
```

这也适用于常量、变量和类型声明。

错误的：

```
const a = 1
const b = 2

var a = 1
var b = 2
```

HUBBLE

```
type Area float64
type Volume float64
```

正确的:

```
const (
  a = 1
  b = 2
)

var (
  a = 1
  b = 2
)

type (
  Area float64
  Volume float64
)
```

仅分组相关的声明。不要对不相关的声明进行分组。

错误的:

```
type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
  ENV_VAR = "MY_ENV"
)
```

正确的:

```
type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

const ENV_VAR = "MY_ENV"
```

分组并不限制在什么地方使用，例如，可以在函数内使用

错误的:

HUBBLE

```
func f() string {
  var red = color.New(0xff0000)
  var green = color.New(0x00ff00)
  var blue = color.New(0x0000ff)

  ...
}
```

正确的:

```
func f() string {
  var (
    red   = color.New(0xff0000)
    green = color.New(0x00ff00)
    blue  = color.New(0x0000ff)
  )

  ...
}
```

7.1.2.10.4 导入的分组排序

应该存在两个分组:

- 基础包
- 其他

错误的:

```
import (
  "fmt"
  "os"
  "go.uber.org/atomic"
  "golang.org/x/sync/errgroup"
)
```

正确的:

```
import (
  "fmt"
  "os"

  "go.uber.org/atomic"
  "golang.org/x/sync/errgroup"
)
```

7.1.2.10.5 Package Names

命名包时，请选择一个名称，即，

HUBBLE

- 全部小写。没有大写或下划线。
- 大多数调用站点不需要使用命名导入重命名。
- 简明扼要。请记住，每个呼叫站点都会完整地标识该名称。
- 不是复数。例如，net/url，而不是 net/url。
- 不是“common”、“util”、“shared”或“lib”。这些都是不好的，没有信息的名字。

参考Package Names和Style guideline for Go packages

7.1.2.10.6 Function Names

我们遵循 Go 社区使用MixedCaps 作为函数名的惯例。测试函数有一个例外，它可能包含下划线，以便对相关的测试用例进行分组，例如 TestMyFunction_WhatIsBeingTested。

7.1.2.10.7 导入别名

如果包名称与导入路径的最后一个元素不匹配，则必须使用导入别名。

```
import (
  "net/http"

  client "example.com/client-go"
  trace "example.com/trace/v2"
)
```

在所有其他情况下，应避免导入别名，除非导入之间存在直接冲突。

错误的：

```
import (
  "fmt"
  "os"

  nettrace "golang.net/x/trace"
)
```

正确的：

```
import (
  "fmt"
  "os"
  "runtime/trace"

  nettrace "golang.net/x/trace"
)
```

7.1.2.10.8 Function Grouping and Ordering

- 函数应按粗略的调用顺序排序。
- 文件中的函数应按接收器分组。

因此，导出的函数应该首先出现在文件中，在 struct、const、var 定义之后。

newXYZ()/NewXYZ() 可能出现在定义类型之后，但在接收器上的其他方法之前。

HUBBLE

因为函数是按接收器分组的，所以纯实用函数应该出现在文件的末尾。

错误的：

```
func (s *something) Cost() {
  return calcCost(s.weights)
}

type something struct{ ... }

func calcCost(n int[]) int {...}

func (s *something) Stop() {...}

func newSomething() *something {
    return &something{}
}
```

正确的：

```
type something struct{ ... }

func newSomething() *something {
    return &something{}
}

func (s *something) Cost() {
  return calcCost(s.weights)
}

func (s *something) Stop() {...}

func calcCost(n int[]) int {...}
```

7.1.2.10.9 减少嵌套

在可能的情况下，代码应该通过首先处理错误情况/特殊条件并提前返回或继续循环来减少嵌套。减少多层嵌套的代码量。

错误的：

```
for _, v := range data {
  if v.F1 == 1 {
    v = process(v)
    if err := v.Call(); err == nil {
      v.Send()
    } else {
      return err
    }
  } else {
```

HUBBLE

```
    log.Printf("Invalid v: %v", v)
  }
}
```

正确的:

```
for _, v := range data {
  if v.F1 != 1 {
    log.Printf("Invalid v: %v", v)
    continue
  }

  v = process(v)
  if err := v.Call(); err != nil {
    return err
  }
  v.Send()
}
```

7.1.2.10.10 非必要的 else

如果在 If 的两个分支中都设置了变量，则可以用单个 If 替换它。

错误的:

```
var a int
if b {
  a = 100
} else {
  a = 10
}
```

正确的:

```
a := 10
if b {
  a = 100
}
```

7.1.2.10.11 Top-level Variable Declarations

即顶层，使用标准的 var 关键字。不要指定类型，除非它与表达式的类型不同。

错误的:

```
var _s string = F()

func F() string { return "A" }
```

正确的:

HUBBLE

```
var _s = F()
// Since F already states that it returns a string, we don't need to specify
// the type again.

func F() string { return "A" }
```

如果表达式的类型与所需类型不完全匹配，请指定类型。

```
type myError struct{}

func (myError) Error() string { return "error" }

func F() myError { return myError{} }

var _e error = F()
// F returns an object of type myError but we want error.
```

7.1.2.10.12 使用字段名初始化结构

在初始化结构时，几乎总是应该指定字段名。现在这是由 go vet 强制执行的。

错误的:

```
k := User{"John", "Doe", true}
```

正确的:

```
k := User{
    FirstName: "John",
    LastName: "Doe",
    Admin: true,
}
```

异常：当有 3 个或更少的字段时，可以在测试表中省略字段名。

```
tests := []struct{
}{
  op Operation
  want string
}{
  {Add, "add"},
  {Subtract, "subtract"},
}
```

7.1.2.10.13 局部变量声明

如果变量被显式设置为某个值，则应使用短变量声明（:=）。

错误的:

```
var s = "foo"
```

HUBBLE

正确的：

```
s := "foo"
```

但是，在某些情况下，当使用 var 关键字时，默认值更清晰。例如，声明空切片。

错误的：

```
func f(list []int) {
  filtered := []int{}
  for _, v := range list {
    if v > 10 {
      filtered = append(filtered, v)
    }
  }
}
```

正确的：

```
func f(list []int) {
  var filtered []int
  for _, v := range list {
    if v > 10 {
      filtered = append(filtered, v)
    }
  }
}
```

HUBBLE

7.1.2.10.14 nil 是一个有效的 slice

nil 是一个长度 0 的有效切片，这意味着，

- 不应显式返回长度为零的片段。相反，返回零。

错误的	正确的
if x == "" { return []int{} }	if x == "" { return nil }

- 检查切片是否空，使用 len(s) == 0，不要检查 nil

错误的	正确的
func isEmpty(s []string) bool { return s == nil}	func isEmpty(s []string) bool { return len(s) == 0 }

- 零值（用 var 声明的 slice）在不使用 make () 的情况下立即可用。

错误的：

```
nums := []int{}
// or, nums := make([]int)

if add1 {
```

```
  nums = append(nums, 1)
}

if add2 {
  nums = append(nums, 2)
}
```

正确的:

```
var nums []int

if add1 {
  nums = append(nums, 1)
}

if add2 {
  nums = append(nums, 2)
}
```

7.1.2.10.15 缩小变量范围

尽可能缩小变量的范围。如果与reduce 嵌套冲突，请不要缩小范围。

错误的:

```
err := f.Close()
if err != nil {
 return err
}
```

正确的:

```
if err := f.Close(); err != nil {
 return err
}
```

如果需要 If 之外的函数调用的结果，则不应尝试缩小范围。

错误的:

```
if f, err := os.Open("f"); err == nil {
  _, err = io.WriteString(f, "data")
  if err != nil {
    return err
  }
  return f.Close()
} else {
  return err
}
```

正确的:

HUBBLE

```
f, err := os.Open("f")
if err != nil {
    return err
}

if _, err := io.WriteString(f, "data"); err != nil {
  return err
}

return f.Close()
```

7.1.2.10.16 避免裸参数

函数调用中的裸参数可能会损害可读性。添加 C 样式注释（/。。。/）当参数名的含义不明显时。

错误的：

```
// func printInfo(name string, isLocal, done bool)

printInfo("foo", true, true)
```

正确的：

```
// func printInfo(name string, isLocal, done bool)

printInfo("foo", true /* isLocal */, true /* done */)
```

更好的是，用自定义类型替换裸 bool 类型，以获得更可读和类型安全的代码。这允许以后该参数的状态不只是两种（真/假）。

```
type Region int

const (
  UnknownRegion Region = iota
  Local
)

type Status int

const (
  StatusReady = iota + 1
  StatusDone
  // Maybe we will have a StatusInProgress in the future.
)

func printInfo(name string, region Region, status Status)
```

注意：对于“bool”常量，与所有文本一样，注释应该指示参数的名称，而不依赖于参数值。在注释“false”常量时，不要在注释中插入叹号。另外，不要修改注释来否定参数的名称。例如：

```
func endTxn(commit bool){}
OK:     endTxn(false /* commit */)
NOT OK: endTxn(false /* !commit */)
NOT OK: endTxn(false /* abort */)
// If you want to add an explanation to an argument, a suggested style is to
// include both the param name and the explanation with a dash between them:
OK: endTxn(false /* commit - we abort as we concluded above that we can't commit
    ↪  */)
```

7.1.2.10.17 尽量避免“bool”参数

函数的bool参数通常是不可靠的，因为它们暗示代码的基本内容如下：

```
func doSomething(shouldDoX bool) {
  if shouldDoX {
    doX()
  } else {
    doY()
  }
}
```

情况并非总是如此。然而，如果这是对情况的公平评估，则应考虑是否应存在“做其他事”功能。

如果所讨论的“bool”与其他参数一起充当函数的“旋钮”，请考虑将其替换为某种类型的“配置”结构。如果只有一个“bool”参数或情况不太明确，可以考虑用枚举来替换有问题的“bool”。例如：

```
type EndTxnAction bool

const (
Commit EndTxnAction = false
Abort = true
)

func endTxn(action EndTxnAction) {}
is better than:

func endTxn(commit bool) {}
```

7.1.2.10.18 fmt Verbs

最好用最具体的动词。换言之，尽可能避免%v。但是，当格式化可能为 nil 且尚未处理 nil 格式的绑定时，将使用%v。值得注意的是，格式为%s 的 nil 错误将呈现为`%!s(<nil>)`，而格式为%v 的 nil 错误将呈现为`<nil>`。因此，在格式化未知为非零的错误时，首选%v。

7.1.2.10.19 使用原始字符串文字避免转义

Go 支持raw string literals，它可以跨越多行并包含引号。使用这些来避免手动转义的字符串，这些字符串很难读取。

错误的：

```
wantError := "unknown name:\"test\""
```

正确的:

```
wantError := `unknown error:"test"`
```

7.1.2.10.20 初始化结构体引用

初始化结构引用时使用&T{}而不是new(T)，以使其与结构体初始化一致。

错误的:

```
sval := T{Name: "foo"}

// inconsistent
sptr := new(T)
sptr.Name = "bar"
```

正确的:

```
sval := T{Name: "foo"}

sptr := &T{Name: "bar"}
```

7.1.2.10.21 在 Printf 外设置字符串格式

如果在字符串文本之外声明 Printf 样式函数的格式字符串，请使它们成为常量值。

这有助于go vet对格式字符串执行静态分析。

错误的:

```
msg := "unexpected values %v, %v\n"
fmt.Printf(msg, 1, 2)
```

正确的:

```
const msg = "unexpected values %v, %v\n"
fmt.Printf(msg, 1, 2)
```

7.1.2.10.22 命名 Printf 样式函数

声明 Printf 样式函数时，请确保go vet可以检测到它并检查格式字符串。

这意味着您应该尽可能使用预定义的 Printf 样式函数名。go vet会检查这些默认。有关详细信息，请参见Printf family。

如果使用预定义的名称不是一个选项，请用 f:Wrapf 结束您选择的名称，而不是 Wrap。可以要求 go vet 检查特定的 Printf 样式名称，但它们必须以 f 结尾。

```
go vet -printfuncs=wrapf,statusf
```

请参考go vet: Printf family check

7.1.2.10.23 go 实现 SQL

在定义语句或虚拟表的结果列标签时，应用以下一般原则：

- column labels 的大小写应一致
- 不同语句中的同一概念应使用同一个词命名。
- 例如，“variable” 和 “value” 对应于 “SHOW ALL CLUSTER SETTINGS” 和 “SHOW SESSION ALL” 是一致的。
- 这些标签必须可由周围的查询使用，而不必指定引号。
- 例如，`start_key`而不是 “`Start Key`” 。另外，避免使用还需要引用的 SQL 关键字：`table_name`而不是 “`table`” ，`index_name`而不是 “`index`” ，等等。永远不要将列命名为`user`，语法特殊形式`user`的错误风险太高。改用`user_name`。
- 使用下划线分隔单词，以与`information_schema`保持一致。例如，`start_key`而不是`startkey`
- 避免使用缩写，除非英语口语中的大多数单词已经使用缩写。例如， `id`是`identifier`可以， 但`loc` ↪ 是`location`不行.
- 对于类似于 “主键” 的列（标识其余列所描述的对象），必须在标签中消除其含义高度重载的单词的歧义。例如，`zone_id`，`job_id`，`table_id`而不是到处都是'id'。`table_name`、`column_name`等，而不是`name`。
- 对于可以由两种或两种以上类型的句柄（如按 ID 或按名称）引用的数据库对象，列标签必须消除正在报告的句柄类型的歧义。这使得以后可以在其他列中报告其他句柄。例如，`table_name`而不是`table`，以便以后可以添加`table_id`。
- 复制 “information_schema” 表或先前的 “SHOW” 语句中已存在的数据的标签应与 “information_schema” 表或 “SHOW” 语句使用相同的标签，前提是它已符合上述原则。

7.1.2.11 Patterns

7.1.2.11.1 Data-driven tests

我们通过数据驱动测试包为 hubble 中的表驱动测试（见下一节）提供了一种替代方法。

```
package datadriven

// RunTest invokes a data-driven test. The test cases are contained in a
// separate test file and are dynamically loaded, parsed, and executed by this
// testing framework. By convention, test files are typically located in a
// sub-directory called "testdata". Each test file has the following format:
//
//   <command>[,<command>...] [arg | arg=val | arg=(val1, val2, ...)]...
//   <input to the command>
//   ----
//   <expected results>
//
// The command input can contain blank lines. However, by default, the expected
// results cannot contain blank lines. This alternate syntax allows the use of
// blank lines:
//
//   <command>[,<command>...] [arg | arg=val | arg=(val1, val2, ...)]...
//   <input to the command>
//   ----
//   ----
//   <expected results>
```

```
//
//   <more expected results>
//   ----
//   ----
//
// To execute data-driven tests, pass the path of the test file as well as a
// function which can interpret and execute whatever commands are present in
// the test file. The framework invokes the function, passing it information
// about the test case in a TestData struct. The function then returns the
// actual results of the case, which this function compares with the expected
// results, and either succeeds or fails the test.
func RunTest(t *testing.T, path string, f func(d *TestData) string)
```

7.1.2.11.2 Test Tables

对子测试使用表驱动测试，以避免在核心测试逻辑重复时重复代码。

错误的:

```
// func TestSplitHostPort(t *testing.T)

host, port, err := net.SplitHostPort("192.0.2.0:8000")
require.NoError(t, err)
assert.Equal(t, "192.0.2.0", host)
assert.Equal(t, "8000", port)

host, port, err = net.SplitHostPort("192.0.2.0:http")
require.NoError(t, err)
assert.Equal(t, "192.0.2.0", host)
assert.Equal(t, "http", port)

host, port, err = net.SplitHostPort(":8000")
require.NoError(t, err)
assert.Equal(t, "", host)
assert.Equal(t, "8000", port)

host, port, err = net.SplitHostPort("1:8")
require.NoError(t, err)
assert.Equal(t, "1", host)
assert.Equal(t, "8", port)
```

正确的:

```
// func TestSplitHostPort(t *testing.T)

tests := []struct{
  give     string
  wantHost string
  wantPort string
```

```
}{
  {
    give:     "192.0.2.0:8000",
    wantHost: "192.0.2.0",
    wantPort: "8000",
  },
  {
    give:     "192.0.2.0:http",
    wantHost: "192.0.2.0",
    wantPort: "http",
  },
  {
    give:     ":8000",
    wantHost: "",
    wantPort: "8000",
  },
  {
    give:     "1:8",
    wantHost: "1",
    wantPort: "8",
  },
}

for _, tt := range tests {
  t.Run(tt.give, func(t *testing.T) {
    host, port, err := net.SplitHostPort(tt.give)
    require.NoError(t, err)
    assert.Equal(t, tt.wantHost, host)
    assert.Equal(t, tt.wantPort, port)
  })
}
```

测试表使向错误消息添加上下文、减少重复逻辑和添加新测试用例变得更容易。

我们遵循这样的约定：结构的片段被称为测试和每个测试用例 tt。此外，我们鼓励使用 give 和 want 前缀解释每个测试用例的输入和输出值。

```
tests := []struct{
  give     string
  wantHost string
  wantPort string
}{
  // ...
}

for _, tt := range tests {
  // ...
}
```

HUBBLE

7.1.2.11.3 Functional Options

函数选项是一种模式，在这种模式中，您可以声明一个不透明的选项类型，该类型在某些内部结构中记录信息。您接受这些选项的可变数目，并根据选项在内部结构上记录的完整信息进行操作。

将此模式用于构造函数和其他需要扩展的公共 api 中的可选参数，特别是在这些函数上已经有三个或更多参数的情况下。

错误的:

```
// package db

func Connect(
  addr string,
  timeout time.Duration,
  caching bool,
) (*Connection, error) {
  // ...
}

// Timeout and caching must always be provided,
// even if the user wants to use the default.

db.Connect(addr, db.DefaultTimeout, db.DefaultCaching)
db.Connect(addr, newTimeout, db.DefaultCaching)
db.Connect(addr, db.DefaultTimeout, false /* caching */)
db.Connect(addr, newTimeout, false /* caching */)
```

正确的:

```
type options struct {
  timeout time.Duration
  caching bool
}

// Option overrides behavior of Connect.
type Option interface {
  apply(*options)
}

type optionFunc func(*options)

func (f optionFunc) apply(o *options) {
  f(o)
}

func WithTimeout(t time.Duration) Option {
  return optionFunc(func(o *options) {
    o.timeout = t
  })
```

HUBBLE

```
}

func WithCaching(cache bool) Option {
  return optionFunc(func(o *options) {
    o.caching = cache
  })
}

// Connect creates a connection.
func Connect(
  addr string,
  opts ...Option,
) (*Connection, error) {
  options := options{
    timeout: defaultTimeout,
    caching: defaultCaching,
  }

  for _, o := range opts {
    o.apply(&options)
  }

  // ...
}

// Options must be provided only if needed.

db.Connect(addr)
db.Connect(addr, db.WithTimeout(newTimeout))
db.Connect(addr, db.WithCaching(false))
db.Connect(
  addr,
  db.WithCaching(false),
  db.WithTimeout(newTimeout),
)
```

参考下面关于函数选项的文档Functional options for friendly APIs

7.2 go 的测试方案

7.2.1 单元测试

7.2.1.1 编写测试

编写测试和函数很类似，其中有一些规则

- 程序需要在一个名为 xxx_test.go 的文件中编写
- 测试函数的命名必须以单词 Test 开始
- 测试函数只接受一个参数`t *testing.T`

HUBBLE

类型为*testing.T的变量t是测试框架中的 hook（钩子），使用这个可以在测试用例中添加相关的操作，例如失败可以执行t.Fail()

举例说明，测试最简单的 helloworld 方法

hello.go

```
func Hello() string {
    return "Hello, world"
}
```

hello_test.go

```
func TestHello(t *testing.T) {
    got := Hello()
    want := "Hello, world"

    if got != want {
        t.Errorf("got '%q' want '%q'", got, want)
    }
}
```

子测试例子

```
func TestHello(t *testing.T) {

    t.Run("saying hello to people", func(t *testing.T) {
        got := Hello("Chris")
        want := "Hello, Chris"

        if got != want {
            t.Errorf("got '%q' want '%q'", got, want)
        }
    })

    t.Run("say hello world when an empty string is supplied", func(t *testing.T)
        ↪ {
        got := Hello("")
        want := "Hello, World"

        if got != want {
            t.Errorf("got '%q' want '%q'", got, want)
        }
    })
}
```

这个测试用例说明 Hello 函数需要有字符串参数，并且当空字符串时返回Hello, World，当字符串非空时返回Hello,加字符串。

可以对断言进行重构，变成函数提高可读性，在 Go 中，可以在其他函数中声明函数，并将它们分给变量。像调用普通函数一样调用它们，即是断言重构，断言重构例子：

HUBBLE

```
func TestHello(t *testing.T) {
    assertCorrectMessage := func(t *testing.T, got, want string) {
        t.Helper()
        if got != want {
            t.Errorf("got '%q' want '%q'", got, want)
        }
    }

    t.Run("saying hello to people", func(t *testing.T) {
        got := Hello("Chris")
        want := "Hello, Chris"
        assertCorrectMessage(t, got, want)
    })

    t.Run("empty string defaults to 'world'", func(t *testing.T) {
        got := Hello("")
        want := "Hello, World"
        assertCorrectMessage(t, got, want)
    })
}
```

将断言重构为函数可以减少重复，并提高测试的可读性。`t.Helper()`告诉测试这个方法是辅助函数，这样，当失败时，测试失败时所报告的行号将在函数调用中而不是在这个辅助函数内部。

7.2.1.2 编写示例

Go 示例代码执行起来就像测试一样，所以可以认为示例反映出的是代码的实际功能。

作为包的测试套件的一部分，示例会被编译并可以选择性的执行。

示例也存在于一个包的 _test.go 文件的函数中。例如

```
func ExampleAdd() {
    sum := Add(1, 5)
    fmt.Println(sum)
    // Output: 6
}
```

当删除注释 Output 行时，示例函数虽然被编译，但不会执行。

通过添加这段代码，示例将出现在godoc的文档中，将使代码更容易理解。

可以通过运行`godoc -http=:6060`来访问所有包的文档中找到示例。

7.2.1.3 Mocking

假设存在一个测试，耗时 4 秒钟，如果大量运行这类测试用例会破坏开发人员的生产力。

例如，有一个程序，从 3 开始依次递减向下，当到 0 时打印 GO! 并退出，要求每次打印从新的一行开始且打印间隔 1 秒钟。

```
3
2
1
GO!
```

代码如下

```
const finalWord = "Go!"
const countdownStart = 3

func Countdown(out io.Writer) {
    for i := countdownStart; i > 0; i-- {
        time.Sleep(1 * time.Second)
        fmt.Fprintln(out, i)
    }

    time.Sleep(1 * time.Second)
    fmt.Fprint(out, finalWord)
}
```

如果不使用 mock 的话，测试用例：

```
func TestCountdown(t *testing.T) {
    buffer := &bytes.Buffer{}

    Countdown(buffer)

    got := buffer.String()
    want := `3
2
1
Go!`

    if got != want {
        t.Errorf("got '%s' want '%s'", got, want)
    }
}
```

每次输出将间隔 3 秒钟。这是可以采用 Mock 技术，就是模拟 time.Sleep，通过依赖注入的方式来代替真正的 time.Sleep，然后通过断言来监视调用。

将依赖关系定义为一个接口，只有在实际使用 Countdown 时使用真实的 sleeper，测试时使用 spysleeper。同时考虑时间间隔，上边的测试并没有反应每次输出时间隔 3 秒，比如说实际程序并没有 sleep 动作，或者只有行有 sleep 动作，一样可以通过测试，如果要反应真实的执行顺序，需要打印出每一个动作的，例如：

1. Sleep
2. Print N
3. Sleep
4. Print N-1

HUBBLE

5. Sleep
6. ...

代码示例：

主程序中定义 Sleeper 接口，并定义实际使用的接口实现

```
type Sleeper interface {
    Sleep()
}

type ConfigurableSleeper struct {
    duration time.Duration
    sleep    func(time.Duration)
}

func (c *ConfigurableSleeper) Sleep() {
    c.sleep(c.duration)
}

const finalWord = "Go!"
const countdownStart = 3

func Countdown(out io.Writer, sleeper Sleeper) {
    for i := countdownStart; i > 0; i-- {
        sleeper.Sleep()
        fmt.Fprintln(out, i)
    }

    sleeper.Sleep()
    fmt.Fprint(out, finalWord)
}

func main() {
    sleeper := &ConfigurableSleeper{1 * time.Second, time.Sleep}
    Countdown(os.Stdout, sleeper)
}
```

测试用例，采用 mock 技术来实现 Sleeper 接口，同时每个调用都打印各自的调用动作。

```
type CountdownOperationsSpy struct {
    Calls []string
}

func (s *CountdownOperationsSpy) Sleep() {
    s.Calls = append(s.Calls, sleep)
}

func (s *CountdownOperationsSpy) Write(p []byte) (n int, err error) {
```

HUBBLE

```
    s.Calls = append(s.Calls, write)
    return
}

const write = "write"
const sleep = "sleep"

type SpyTime struct {
    durationSlept time.Duration
}

func (s *SpyTime) Sleep(duration time.Duration) {
    s.durationSlept = duration
}

func TestCountdown(t *testing.T) {

    t.Run("prints 3 to Go!", func(t *testing.T) {
        buffer := &bytes.Buffer{}
        Countdown(buffer, &CountdownOperationsSpy{})

        got := buffer.String()
        want := `3
2
1
Go!`

        if got != want {
            t.Errorf("got %q want %q", got, want)
        }
    })

    t.Run("sleep before every print", func(t *testing.T) {
        spySleepPrinter := &CountdownOperationsSpy{}
        Countdown(spySleepPrinter, spySleepPrinter)

        want := []string{
            sleep,
            write,
            sleep,
            write,
            sleep,
            write,
            sleep,
            write,
        }
```

HUBBLE

```
        if !reflect.DeepEqual(want, spySleepPrinter.Calls) {
            t.Errorf("wanted calls %v got %v", want, spySleepPrinter.Calls)
        }
    })
}

func TestConfigurableSleeper(t *testing.T) {
    sleepTime := 5 * time.Second

    spyTime := &SpyTime{}
    sleeper := ConfigurableSleeper{sleepTime, spyTime.Sleep}
    sleeper.Sleep()

    if spyTime.durationSlept != sleepTime {
        t.Errorf("should have slept for %v but slept for %v", sleepTime, spyTime
            ↪ .durationSlept)
    }
}
```

mock 会使的一个简单的程序变得繁复，对于简单的功能用处不大，但对于一些比较复杂的操作，有着必要性。

在使用时需要注意：

- 在测试时想要的是行为还是实现细节
- 如果要重构，需要做多少修改
- 一个测试需要经过多少次模拟动作，如果超过 3 个，就需要重新考虑设计

没有对代码中重要区域进行 mock 可能导致难以测试，例如，调用一个可能失败的服务，怎么在特定状态下测试系统。

*** 并发测试

代码如下：

```
import (
    "time"
)

type WebsiteChecker func(string) bool

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
    results := make(map[string]bool)

    for _, url := range urls {
        go func(u string) {
            results[u] = wc(u)
        }(url)
    }

    time.Sleep(2 * time.Second)
```

HUBBLE

```
    return results
}
```

对于存在并发的方法，可以使用 benchmark 来进行测试，既可以测试方法效率又可以查看是否存在并发问题：

```
func slowStubWebsiteChecker(_ string) bool {
    time.Sleep(20 * time.Millisecond)
    return true
}

func BenchmarkCheckWebsites(b *testing.B) {
    urls := make([]string, 100)
    for i := 0; i < len(urls); i++ {
        urls[i] = "a url"
    }

    for i := 0; i < b.N; i++ {
        CheckWebsites(slowStubWebsiteChecker, urls)
    }
}
```

基准测试使用一百个网址的 slice 来对 CheckWebsites 进行测试，并使用 WebsiteChecker 的模拟实现，故意放慢速度到 20 毫秒。

这时，运行`go test -bench=./..`将会出现并发问题。

这个问题可以使用 race 标志来运行测试，go test -race 来找到错误。

可以看到对应同一内存地址，进行了同时写入。

对于这个例子可以使用 chan 来解决。

race 同时也可以用在运行时的并发检测，如`go run -race`来运行。

对于上边的测试用例，比较的都是字符串，如果比较结构体等，需要使用反射进行深层比较，就是`reflect.DeepEqual`方法。同时针对单元测试应该保证有效代码的测试覆盖率。

7.2.2 go-fuzz

go-fuzz 是 Go 语言的随机测试工具，是覆盖驱动的 fuzzing 解决方案。Fuzzing 主要是应用在解析复杂输入的包（文本或者二进制）。模糊测试是一项使用随机数据加载程序的测试技术，是对常规测试的补充，并且使开发者可以发现那些在手工生成的输入下难以发现的 bug。模糊测试在 Go 程序中很容易设置，并且可以适应于几乎所有类型的代码。

数据库的 SQL 输入就是这样的一种复杂输入，使用 go-fuzz 可以比较全面的发现解析器可能存在的问题。

在 go 中由两个工具包适用于模糊测试：

- Google 的 gofuzz, 地址：https://github.com/google/gofuzz
- dvyukov 的 go-fuzz, 地址：https://github.com/dvyukov/go-fuzz

二者用途不同：

- gofuzz 提供了一个可以用随机值填充你的 Go 结构体的包。而你需要做的是编写测试代码，并且调用这个包来获取随机数据。当你想要模糊测试结构化数据的时候，这个包是完美的。这里是使用随机数据对一个结构体进行 50000 次模糊测试的例子，其中指针/切片/map 有 50% 的几率被设置为空
- go-fuzz 基于已经在大多数知名的软件或库中发现了上百个 bug 的 American Fuzzy Lop。Go-Fuzz 会连续运行，并且根据提供的样本生成随机的字符串。之后必须解析这些字符串，并且明确地将其标记为是否可用于测试。任何有趣的生成的数据都会被该工具所报告，这些数据增加了代码的覆盖率或者导致崩溃。该工具十分适合那些管理诸如 XML，JSON，图像等字符串信息的程序。这里是该工具运行以及发行问题的预览，被称为 crasher。

7.2.3 数据库测试体系

参考 sqlite 的testing

7.2.3.1 测试工具

TCL 脚本语言测试，可以应对大部分的交互式测试。

SQL 逻辑测试，可以参考 sqlite 的逻辑测试

sqlfuzz，参考上边的 go-fuzz 工具。

7.2.3.2 异常测试

异常测试目的在于出现问题时验证数据库正确行为的测试。一个数据库在正常情况下可以正常的运行，并输出格式正确的结果。但需要对无效输入做出理性响应并在系统故障时进行异常测试验证。

7.2.3.2.1 内存不足测试

数据库对于内存回收异常重要，引发的 OOM 将是灾难性的。需要通过模拟 OOM 错误来完成内存不足测试。如果内存分配失败时如何处理。

7.2.3.2.2 I/O 错误测试

验证数据库是否对失败的 I/O 操作做出合理响应。例如磁盘驱动器已满，磁盘故障，使用网络文件系统时网络中断，系统配置或权限更改等等一起的故障。是否都能做到正确的响应。

7.2.3.2.3 崩溃测试

崩溃测试目的在证明如果应用或操作系统崩溃，或者数据库在运行过程中出现了电源故障等，数据库不会损坏。

崩溃测试可以在仿真情况下完成，例如使用虚拟系统。

测试方法可以是在写操作的中间随机崩溃，重新打开并读取测试数据库，验证写操作是否成功完成或已完全回滚。

7.2.3.2.4 复合故障测试

堆叠多个故障，运行测试确保从崩溃恢复时发生 I/O 或 OOM 故障时，数据库没有损坏，并且行为正常。

7.2.3.3 模糊测试

HUBBLE

7.2.3.3.1 SQL 模糊

使用 American Fuzzy Lop Fuzzer 的 SQL Fuzz，就是 go-fuzz 进行模糊测试。

Google OSS 模糊测试

其他模拟器。

7.2.3.3.2 格式错误的数据库文件

例如把数据库文件添加一个或多个字节，数据库是否可使用，是否可修复。

7.2.3.3.3 边界值检查

各种类型的最大长度，表的最大列数等边缘测试，验证允许的值都能正确执行，超出限制时是否正确返回错误。

7.2.3.4 回归测试

每当有针对数据库的 bug 被报告后，需要将 bug 存在的问题添加到测试用例中，回归测试可以确保不会将以前已修复的错误重新引入到数据库中。

7.2.3.5 资源泄漏测试

当分配资源后并不释放时，就会发生资源泄漏。

7.3 交互式测试脚本开发

一个数据库中包含的功能及语法非常庞大，如果是按照语法手动输入测试会非常繁琐且易出错。TCL 脚本广泛应用于自动化测试，快速原型开发，脚本编程，GUI 等各个方面。例如，Spirent TestCenter 就是基于 TCL 进行网络设备测试的。

expect 是 Unix 系统中用来进行自动化控制和测试的软件工具，应用在交互式软件中如 telnet，ftp，Passwd，fsck，rlogin，tip，ssh 等等。在 deepin 下，远程登录就是利用 expect 实现的。在实际工作中，我们运行命令、脚本或程序时，这些命令、脚本或程序都需要从终端输入某些继续运行的指令，而这些输入都需要人为的手工进行。而利用 expect，则可以根据程序的提示，模拟标准输入提供给程序，从而实现自动化交互执行。这就是 expect。expect 是由 tcl 语言演变而来的，所以 expect 脚本的运行需要 tcl 的支持。安装 tcl 和 expect 的步骤略，请自行查找安装方式。

7.3.1 expect 基础

7.3.1.1 基本命令

expect 基本命令：

- `send` 用于向进程发送字符串

send 命令接收一个字符串参数，并将该参数发送到进程。

- `expect` 从进程接收字符串

expect 命令和 send 命令相反，expect 通常用来等待一个进程的反馈，我们根据进程的反馈，再发送对应的交互命令。

- `spawn` 启动新的进程

spawn 命令用来启动新的进程，spawn 后的 send 和 expect 命令都是和使用 spawn 打开的进程进行交互。

- `interact` 允许用户交互

interact 命令用的其实不是很多，一般情况下使用 spawn、send 和 expect 命令就可以很好的完成我们的任务；但在一些特殊场合下还是需要使用 interact 命令的，interact 命令主要用于退出自动化，进入人工交互。

7.3.1.2 简单示例

参考下面脚本：

```
#!/usr/bin/expect

set timeout 30
set host "192.168.1.215"
set username "root"
set password "123456"

spawn ssh $username@$host
expect "*password*" {send "$password\r"}
interact
```

`#!/usr/bin/expect` 使用 expect 来解释这个脚本。

`set timeout 30` 设置超时时间，单位是秒，默认是 10 秒，这里设置了 30 秒超时。

`set host "192.168.1.215"`、`set username "root"`、`set password "123456"` 设置 host、username、password 变量。

`spawn ssh $username@$host` spawn 是进入 expect 环境后才可以执行的 expect 内部命令，如果没有装 expect 或者直接在默认的 SHELL 下执行是找不到 spawn 命令的。它主要的功能是给 ssh 运行进程加个壳，用来传递交互指令；

`expect "*password*"` 这里的 expect 也是 expect 的一个内部命令，这个命令的意思是判断上次输出结果里是否包含 “password” 的字符串，如果有则立即返回；否则就等待一段时间后返回，这里等待时长就是前面设置的 30 秒；

`send "$password\r"` 当匹配到对应的输出结果时，就发送密码到打开的 ssh 进程，执行交互动作；

`interact` 执行完成后保持交互状态，把控制权交给控制台，这个时候就可以手工操作了。如果没有这一句登录完成后会退出，而不是留在远程终端上。

很简单的 expect 脚本，就是设定好的用户名、密码登录服务器，实现了类似于 xshell 记录用户名密码进行自动登录的功能。

但这里包含了 expect 一个重要的概念，模式-动作。

7.3.1.3 模式-动作

`expect "*password*" {send "$password\r"}`即是模式-动作的代表，简单来说就是匹配一个模式，匹配到就执行对应的动作。在这里，就是匹配命令行输出password字符串，如果有，就输入密码。

复杂一点的例子：

```
expect {
    "password" {
        send "$password\r"
        exp_continue
    }
```

```
    eof
    {
        send "eof"
    }
}
```

其中 exp_continue 表示循环式匹配，通常匹配之后都会退出语句，但如果有 exp_continue 则可以不断循环匹配，输入多条命令，简化写法。

7.3.1.4 传参

很多时候，我们需要传递参数到脚本中，现在通过下面这段代码来看看如何在 expect 中使用参数:

```
#!/usr/bin/expect

if {$argc < 3} {
    puts "Usage:cmd <host> <username> <password>"
    exit 1
}

set timeout -1
set host [lindex $argv 0]
set username [lindex $argv 1]
set password [lindex $argv 2]

spawn ssh $username@$host
expect "*password*" {send "$password\r"}
interact
```

在 expect 中，*argc* *count* argv 中，比如取第一个参数就是 [lindex $argv 0]，以此类推。

TCL 和 expect 详细的参考文档请参考Expect and TCL mini reference manual

7.3.2 hubble 中的实际测试例子

test.tcl:

```
#! /usr/bin/env expect -f

source [file join [file dirname $argv0] common.tcl]

start_test "Check that demo says hello properly"

spawn $argv demo

eexpect "Welcome"

eexpect "your changes to data stored in the demo session will not be saved!"

eexpect "Web UI: http:"
```

HUBBLE

```
eexpect "Server version"
eexpect "Cluster ID"
eexpect "brief introduction"

eexpect root@

eexpect "movr>"
end_test
```

这个脚本依赖于common.tcl，在 common.tcl 中定义了一些公用调用 proc，例如这个例子中的eexpect, 就是在 common.tcl 中定义的。

```
proc eexpect {text} {
    expect {
    $text {}
    timeout { handle_timeout $text }
    }
}
```

eexpect 用于匹配字符串，并处理 timeout。start_test，end_test也是 common 中定义的输出文本语句。

实际测试时使用 expect -f path/to/test.tcl path/to/hubble

所以在spawn $argv demo时，会运行hubble demo命令。

之后就是校验输出，确定进入的库等信息。

如果是校验具体的 SQL，可以添加

```
send "select count(*) from t.foo;\r"
eexpect "0"
```

即可校验 t.foo 表的 count 值是否符合预期。

7.3.3 计划

持续更新自动化测试部分代码，尽量使测试覆盖的范围更广。

7.4 generate 命令

go generate命令是从 Go1.4 开始才设计的，用于在编译前自动化生成某类代码。go generate和go build是完全不一样的命令，通过分析源码中特殊的注释，然后执行相应的命令。

这些命令都是很明确的，没有任何的依赖在里面。这个go generate是给源码开发人员用的，不是给使用这个包的人用的，是方便你来生成一些代码的。

有几点需要注意:

1. 该特殊注释必须在.go源码文件中。
2. 每个源码文件可以包含多个generate特殊注释时，显式运行go generate命令时，才会执行特殊注释后面的命令。
3. 命令串行执行的，如果出错，就终止后面的执行。

HUBBLE

4. 特殊注释必须以"//go:generate" 开头，双斜线后面没有空格。

例如，使用 yacc 来生成代码，可能如下命令：

```
go tool yacc -o gopher.go -p parser gopher.y
```

-o 指定了输出的文件名，-p指定了package的名称，这是一个单独的命令，如果我们想用go generate来触发这个命令，那么就可以在当前目录的任意一个xxx.go文件里面的任意位置增加一行如下的注释：

```
//go:generate go tool yacc -o gopher.go -p parser gopher.y
```

//go:generate 是没有任何空格的，这其实就是一个固定的格式，在扫描源码文件的时候就是根据这个来判断的。

这时可以通过如下命令来生成、编译、测试。如果gopher.y文件有修改，那么就重新执行go generate重新生成文件就好。

```
go generate
go build
go test
```

在下面这些场景下，我们会使用go generate命令（不完整列表）：

- goyacc Go 的 Yacc。从.y文件生成.go文件
- stringer 实现 fmt.Stringer 枚举的接口。
- gostringer fmt.GoStringer 为枚举实现接口。
- jsonenums 枚举的实现json.Marshaler和json.Unmarshaler接口。
- go-syncmap 使用软件包作为的通用模板生成 Go 代码sync.Map。
- go-syncpool 使用软件包作为的通用模板生成 Go 代码sync.Pool。
- go-atomicvalue 使用软件包作为的通用模板生成 Go 代码atomic.Value。
- go-nulljson 使用包作为实现database/sql.Scanner和的通用模板生成 Go 代码database/sql/driver ↪ .Valuer。
- go-enum 使用包作为实现接口的通用模板生成 Go 代码fmt.Stringer|binary|json|text|sql|yaml枚举。
- go-import 执行非 go 文件的自动导入。
- gojson 从示例 json 文档生成 go 结构定义。
- vfsgen 生成静态实现给定虚拟文件系统的vfsdata.go文件。
- goreuse 使用包作为通用模板通过替换定义来生成 Go 代码。
- embedfiles 将文件嵌入 Go 代码。
- ragel 状态机编译器
- peachpy 嵌入在 Python 中的 x86-64 汇编器，生成 Go 汇编
- bundle Bundle 创建适用于包含在特定目标软件包中的源软件包的单一源文件版本。
- msgp MessagePack 的 Go 代码生成器
- protobuf 从protocol buffer定义文件.proto生成 .pb.go文件
- thriftrw thrift
- gogen-avro avro
- swagger-gen-types 从 swagger 定义中去生成代码
- avo 使用 Go 生成汇编代码
- Wire Go 的编译时依赖注入
- sumgen 从 sum-type 声明生成接口方法实现
- interface-extractor 生成所需类型的接口，仅在包内使用方法。
- deep-copy 为给定类型创建深度复制方法。

- Unicode 从UnicodeData.txt生成Unicode表
- HTML 将HTML文件嵌入到 Go 源码
- binddata 将形如JPEG这样的文件转成 Go 代码中的字节数组。
- 宏为既定的包生成特定的实现，比如用于ints的sort.Ints

> 在 hubble 中存在大量的 probuf，yacc,stringer 自动生成，同时 makefile 文件中也有相关的构建脚本。

go generate命令格式如下所示：

```
go generate [-run regexp] [-n] [-v] [-x] [command] [build flags] [file.go... |
    ↪ packages]
```

参数说明如下：

- -run 正则表达式匹配命令行，仅执行匹配的命令；
- -v 输出被处理的包名和源文件名；
- -n 显示不执行命令；
- -x 显示并执行命令；
- command 可以是在环境变量 PATH 中的任何命令。

执行go generate命令时，也可以使用一些环境变量，如下所示：

- $GOARCH体系架构（arm、amd64 等）；
- $GOOS当前的 OS 环境（linux、windows 等）；
- $GOFILE当前处理中的文件名；
- $GOLINE 当前命令在文件中的行号；
- $GOPACKAGE 当前处理文件的包名；

7.4.1 用法

7.4.1.1 自动化

实现类似于 shell 脚本的功能。

示例

假设我们有一个main.go文件，内容如下：

```
package main

import "fmt"

//go:generate go run main.go
//go:generate go version
func main() {
  fmt.Println("Hello World!")
}
```

执行go generate -x命令，输出结果如下：

```
go run main.go
Hello World!
```

HUBBLE

```
go version
go version go1.13.4 linux/amd64
```

通过运行结果可以看出，相当于写了一个顺序执行go run main.go、go version的脚本。这里-x参数是为了将执行的具体命令打印出来。

7.4.2 枚举

参照官方博客，例子在代码。

在定义枚举时，如果增加一个新的枚举类型，可能需要修改大量的方法，而使用了go:generate stringer可以简化这个过程，再增加新枚举类型时可能只需要修改枚举定义即可。

这里介绍另一个例子，例如有一个错误定义：

```
const (
    ERR_CODE_OK = 0 // OK
    ERR_CODE_INVALID_PARAMS = 1 // invalid parameter
    ERR_CODE_TIMEOUT = 2 // time out
    // ...
)
```

如果需要将错误码和描述信息对应上，可能需要做：

```
var mapErrDesc = map[int]string {
    ERR_CODE_OK: "OK",
    ERR_CODE_INVALID_PARAMS: "invalid parameter",
    ERR_CODE_TIMEOUT: "time out",
    // ...
}

func GetDescription(errCode int) string {
    if desc, exist := mapErrDesc[errCode]; exist {
        return desc
    }

    return fmt.Sprintf("error code: %d", errCode)
}
```

这时每次添加新的错误码时除了要修改错误码时也需要修改 map。

使用 go generate 方式，如果本地没有，首先安装 stringer 包。

```
git clone https://github.com/golang/tools/ $GOPATH/src/golang.org/x/tools
go install golang.org/x/tools/cmd/stringer
```

代码修改为：

```
type ErrCode int

//go:generate stringer -type ErrCode -linecomment
const (
```

HUBBLE

```
    ERR_CODE_OK              ErrCode = 0 // OK
    ERR_CODE_INVALID_PARAMS ErrCode = 1 // invalid parameter
    ERR_CODE_TIMEOUT         ErrCode = 2 // time out
    // ...
)
```

执行 go generate 命令，这时会自动生成文件，包含一个 string() 方法，这时使用它就可以得到对应的错误描述了。再修改枚举时只需要修改枚举类型本身即可。

复杂例子可以参考 hubble 或者jsonenums。jsonenums 是一个用于为枚举类型自动生成 JSON 编组样板代码的类库。在 Go 标准类库里面已经有大量可以用于解析 AST 的接口，而 AST 使得编写元编程工具更简单，更容易。

7.5 cobra 包的使用

hubble 的命令行利用cobra包编写。

cobra 作为 go 语言开发命令行程序最流行的包，应用于多个知名项目，包括：docker,Kubernetes,Hugo,rkt,etcd,Moby,OpenShift 等。

Cobra 是一个库，提供了一个简单的接口来创建类似于 git&go 工具的强大的现代 CLI 接口。Cobra 也是一个应用程序，它将生成应用程序脚手架以快速开发基于 Cobra 的应用程序。

Cobra 提供：

- 基于子命令的简单 CLIs:app server、app fetch 等。
- 完全符合 POSIX 的标志（包括短版本和长版本）
- 嵌套子命令
- 全局、本地和级联标志
- 使用`cobra init appname`和`cobra add cmdname`轻松生成应用程序和命令
- 智能建议（`app srver...`拼写错误时会提示为`app server`）
- 命令和标志的自动帮助生成
- 自动识别`-h`、`-help`等帮助标志。
- 为应用程序生成 bash 自动完成
- 为应用程序自动生成的手册页
- 命令别名，以便您可以更改而不破坏它们
- 定义自己的帮助、用法等的灵活性。
- 可选的与`viper`结合，来开发 12-factor 程序。

如果参考代码学习 cobra 可以参考码云上的简单例子cobra-example

7.5.1 概念

Cobra 是建立在命令 (Commands)、参数 (Args) 和标志 (Flags) 之上的。命令表示动作，参数是事物，标志是这些动作的修饰符。最好的命令行应用程序在使用时读起来像句子。用户将知道如何使用该应用程序，因为他们将了解如何使用它。

模式遵从 APPNAME-动词-名词-形容词，或者 APPNAME-命令-ARG-标志。

例如，hugo 中这样使用，`server`是命令，`port`是标志：

```
hugo server --port=1313
```

git 中 bare 的方式进行克隆，形如“APPNAME-动词-名词-形容词”：

```
git clone URL --bare
```

7.5.1.1 Commands

命令是应用程序入口。应用程序支持的每个交互都将包含在命令中。命令可以有子命令，也可以运行操作。

例如，cobra-example下的root.go代码下的RootCmd。

7.5.1.2 Flags

标志是修改命令行为的一种方式。Cobra 支持完全符合 POSIX 的标志以及 Go 标志包。Cobra 命令可以定义持续到子命令的标志，以及仅对该命令可用的标志。

在上面的示例中，port就是标志。

标志功能是由pflag库提供的，pflag库是标志标准库的一个分支，在添加 POSIX 兼容性的同时维护相同的接口。

7.5.2 安装

如果需要支持 cobra 命令自动生成代码，可以使用

```
go get github.com/spf13/cobra/cobra
```

将cobra添加到$GOROOT/bin下作为命令使用，这时也可以在程序中引入使用

```
import "github.com/spf13/cobra"
```

如果不需要代码生成只需要在项目中使用，比如 hubble 使用的 cobra 库版本是 v0.0.5，可以在go.mod下增加

```
github.com/spf13/cobra v0.0.5
```

引入同上边描述。

命令的使用方式请参考官方文档，可以试着按官方文档使用命令生成脚手架程序。

7.5.3 使用

参考 cobra-example 程序，入口在 main.go 中。

```
func main() {
    Execute()
}
```

Execute()在 root.go 中定义，通过这个方法调用主命令RootCmd的Execute()来启动命令行。

RootCmd 的定义：

```
var RootCmd = &cobra.Command{
    Use:   "cobra-example",
    Short: "cobra例子",
    Long: `cobra-example是一个使用cobra工具包的例子程序。

Cobra是一个开发Go语言CLI端程序的库，现已应用于docker,Kubernetes,Hugo,etcd等多个
    ↪ 知名的大型Go语言项目中。`,
```

```
    // Uncomment the following line if your bare application
    // has an action associated with it:
    // Run: func(cmd *cobra.Command, args []string) { },
}
```

一般在 init() 中绑定 flags，flags 分为两类，一种 LocalFlag，一种 PersistentFlag。

PersistentFlag相当于全局 flag,LocalFlag一般只应用于本命令。

例如:

```
RootCmd.PersistentFlags().StringVar(&cfgFile, "config",
    "", "config file (default is $HOME/.cobra-example.yaml)")
RootCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
```

这样定义的 flag 在程序的所有命令中都可用。

在运行其他命令时会输出:

```
Global Flags:
      --config string   config file (default is $HOME/.cobra-example.yaml)
```

而toogle这时看不到。

标志的使用可以参考 start.go,

```
startCmd.Flags().BoolVarP(&backend, "backend", "b", false, "后端启动")
```

这样在命令行输入--backend=true时，定义的backend就能获取值了。

命令嵌套使用AddCommand，例如 hubble 中:

```
hubbleCmd.AddCommand(
        startCmd,
        startSingleNodeCmd,
        initCmd,
        certCmd,
        quitCmd,

        sqlShellCmd,
        userCmd,
        nodeCmd,
        dumpCmd,

        demoCmd,
        genCmd,
        versionCmd,
        DebugCmd,
        sqlfmtCmd,
        workloadcli.WorkloadCmd(true /* userFacing */),
        systemBenchCmd,
    )
```

HUBBLE

这段代码将start等子命令添加到hubble命令下。

每个命令都是先执行 init 进行 flag 绑定，run 进行实际操作的。其他详细使用方式请参考例子。

7.5.4 后续

例子中包含了一个标准的使用配置文件的方法。更详细的 viper 包使用请参考官方文档

由 init 中定义cobra.OnInitialize来加载配置文件，在使用时将其与 flags 进行绑定。

但实际开发中遇到了其中存在的问题。cobra.OnInitialize 虽然在代码之初定义，但实际运行是在所有 init 之后。看代码的话发现 AddCommand 和 OnInitialize 虽然都是调用append，但实际上

OnInitialize 源码定义

```
initializers = append(initializers, y...)
```

AddCommand 中定义

```
c.commands = append(c.commands, x)
```

可能规则是按字母顺序进行加载（存疑）。

为什么 OnInitialize 后加载呢，OnInitialize 相当于添加额外的初始化，而这部分应该在绑定 flag 之后，否则也无法在 OnInitialize 中读取到 flag 的参数。

如

```
./cobra-example start --config .cobra-example.yaml
```

先获取 config 的参数文件名，然后在读取文件中的配置。

逻辑上是正确的，但如果想在 init 中使用文件中的配置这种方式就会失败。而 hubble 中这种方式就存在问题，原因是所有参数都是要在 init 时获取的。

所以，目前 hubble 中的读取实现是采用的直接定义配置文件的方式，但这样做不够自定义化。在后续重构代码时可以考虑将此处作为一个待修改点。

7.6 go 中使用 web

web 是基于 http 协议的一个服务，Go 中提供了一个完善的 net/http 包，通过 http 包可以很方便的搭建一个可以运行的 web 服务。

http 包建立 web 服务器:

```
package main

import (
    "fmt"
    "net/http"
    "strings"
    "log"
)

func sayHello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
```

```
}

func main() {
    http.HandleFunc("/", sayHello)

    err := http.ListenAndServe(":9000", nil)

    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}
```

在go run main.go后，打开浏览器，输入http://localhost:9000将看到输出Hello World!。

7.6.1 web 工作方式的几个概念

以下均是服务器端的几个概念

Request： 用户请求的信息，用来解析用户的请求信息，包括 post、get、cookie、url 等信息

Response： 服务器需要反馈给客户端的信息

Conn： 用户的每次请求链接

Handler： 处理请求和生成返回信息的处理逻辑

http 有两个核心功能：Conn、ServeMux

7.6.1.1 Conn 的 goroutine

与一般编写的 http 服务器不同，Go 为了实现高并发和高性能，使用了 goroutines 来处理 Conn 的读写事件，这样每个请求都能保持独立，相互不会阻塞，可以高效的响应网络事件。这是 Go 高效的保证。

Go 在等待客户端请求里面是这样写的：

```
c, err := srv.newConn(rw)
if err != nil {
    continue
}
go c.serve()
```

可以看到客户端的每次请求都会创建一个 Conn，这个 Conn 里面保存了该次请求的信息，然后再传递到对应的 handler，该 handler 中便可以读取到相应的 header 信息，这样保证了每个请求的独立性。

7.6.1.2 ServeMux 的自定义

前面讲述conn.server的时候，其实内部是调用了http包默认的路由器，通过路由器把本次请求的信息传递到了后端的处理函数。那么这个路由器是怎么实现的呢?

它的结构如下：

```
type ServeMux struct {
    mu sync.RWMutex    //锁，由于请求涉及到并发处理，因此这里需要一个锁机制
    m  map[string]muxEntry  // 路由规则，一个string对应一个mux实体，这里的string
        ↪ 就是注册的路由表达式
```

```
    hosts bool // 是否在任意的规则中带有host信息
}
```

下面看一下 muxEntry

```
type muxEntry struct {
    explicit bool   // 是否精确匹配
    h        Handler // 这个路由表达式对应哪个handler
    pattern  string  //匹配字符串
}
```

接着看一下 Handler 的定义

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)  // 路由实现器
}
```

Handler 是一个接口，前边例子的sayHello并没有实现 ServeHTTP 接口，是因为在 http 包中还定义了一个类型HandlerFunc，sayHello实际上是HandlerFunc调用后的结果，这个类型默认实现了 ServeHTTP 接口，就是说，当我们调用了 HandlerFunc(f)，强制将 f 转换为了 HandlerFunc 类型，这样 f 就拥有了 ServeHTTP 方法。

```
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

路由器里面存储好了相应的路由规则之后，那么具体的请求又是怎么分发的呢？请看下面的代码，默认的路由器实现了ServeHTTP：

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) {
    if r.RequestURI == "*" {
        w.Header().Set("Connection", "close")
        w.WriteHeader(StatusBadRequest)
        return
    }
    h, _ := mux.Handler(r)
    h.ServeHTTP(w, r)
}
```

如上所示路由器接收到请求之后，如果是*那么关闭链接，不然调用mux.Handler(r)返回对应设置路由的处理Handler，然后执行h.ServeHTTP(w, r)

也就是调用对应路由的handler的ServerHTTP接口，那么mux.Handler(r)怎么处理的呢？

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string) {
    if r.Method != "CONNECT" {
        if p := cleanPath(r.URL.Path); p != r.URL.Path {
            _, pattern = mux.handler(r.Host, p)
            return RedirectHandler(p, StatusMovedPermanently), pattern
```

```
            }
        }
    return mux.handler(r.Host, r.URL.Path)
}

func (mux *ServeMux) handler(host, path string) (h Handler, pattern string) {
    mux.mu.RLock()
    defer mux.mu.RUnlock()

    // Host-specific pattern takes precedence over generic ones
    if mux.hosts {
        h, pattern = mux.match(host + path)
    }
    if h == nil {
        h, pattern = mux.match(path)
    }
    if h == nil {
        h, pattern = NotFoundHandler(), ""
    }
    return
}
```

根据用户请求的 URL 和路由器里面存储的map去匹配，当匹配到之后返回存储的handle，调用这个 handle 的 ServeHTTP 接口就可以执行到相应的函数了。

通过上面这个介绍，我们了解了整个路由过程，Go 其实支持外部实现的路由器 ListenAndServe 的第二个参数就是用以配置外部路由器的，它是一个 Handler 接口，即外部路由器只要实现了 Handler 接口就可以，我们可以在自己实现的路由器的 ServeHTTP 里面实现自定义路由功能。

如下代码所示，我们自己实现了一个简易的路由器：

```
package main

import (
    "fmt"
    "net/http"
)

type MyMux struct {
}

func (p *MyMux) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path == "/" {
        sayhelloName(w, r)
        return
    }
    http.NotFound(w, r)
    return
}
```

```
func sayhelloName(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello myroute!")
}

func main() {
    mux := &MyMux{}
    http.ListenAndServe(":9090", mux)
}
```

Go 代码的执行流程:

通过对 http 包的分析之后，现在让我们来梳理一下整个的代码执行过程。

首先调用Http.HandleFunc

按顺序做了几件事:

1 调用了DefaultServeMux的HandleFunc

2 调用了DefaultServeMux的Handle

3 往DefaultServeMux的map[string]muxEntry中增加对应的handler和路由规则

其次调用 http.ListenAndServe(":9090", nil)

按顺序做了几件事情:

1 实例化Server

2 调用Server的ListenAndServe()

3 调用net.Listen("tcp", addr)监听端口

4 启动一个for循环，在循环体中Accept请求

5 对每个请求实例化一个Conn，并且开启一个goroutine为这个请求进行服务go c.serve()

6 读取每个请求的内容w, err := c.readRequest()

7 判断handler是否为空，如果没有设置handler（这个例子就没有设置handler），handler就设置为DefaultServeMux

8 调用handler的ServeHttp

9 在这个例子中，下面就进入到DefaultServeMux.ServeHttp

10 根据request选择handler，并且进入到这个handler的ServeHTTP

11 选择 handler:

A 判断是否有路由能满足这个request（循环遍历ServerMux的muxEntry）

B 如果有路由满足，调用这个路由handler的ServeHttp

C 如果没有路由满足，调用NotFoundHandler的ServeHttp

在实例中，我们自定义了一个 handle。

web/server.go

HUBBLE

```
func Handler(cfg Config) http.Handler {
    fileServer := http.FileServer(&assetfs.AssetFS{
        Asset:     Asset,
        AssetDir:  AssetDir,
        AssetInfo: AssetInfo,
    })

    info := Info{
        buildTime: version.BuildTime,
        version:   version.Version,
    }

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if !haveUI() {
            http.ServeContent(w, r, "index.html", info.GoTime(), bytes.NewReader
                ↪ (bareIndexHTML))
            return
        }

        if r.URL.Path != "/" {
            fileServer.ServeHTTP(w, r)
            return
        }

        if err := indexHTMLTemplate.Execute(w, indexHTMLArgs{
            ExperimentalUseLogin: cfg.ExperimentalUseLogin,
            LoginEnabled:         cfg.LoginEnabled,
            }); err != nil {
            err = errors.Wrap(err, "templating index.html")
            http.Error(w, err.Error(), 500)
            log.Fatalln(err)
            }
        })
}
```

首先获取静态资源:

```
fileServer := http.FileServer(&assetfs.AssetFS{
    Asset:     Asset,
    AssetDir:  AssetDir,
    AssetInfo: AssetInfo,
    })
```

其次获取包的版本信息:

```
info := Info{
    buildTime: version.BuildTime,
    version:   version.Version,
```

HUBBLE

```
    }
```

最后返回http.HandlerFunc，在其中定义了以下逻辑：

1. 如果加载不到静态资源，返回一个无资源界面。
2. 如果不是"/" 返回对应的路由。
3. 通过模板加载index.html。

7.6.2 静态资源的使用

使用 Go 开发应用的时候，有时会遇到需要读取静态资源的情况。比如开发 Web 应用，程序需要加载模板文件生成输出的 HTML。在程序部署的时候，除了发布应用可执行文件外，还需要额外的发布依赖的静态资源文件。这给发布过程添加了一些麻烦。既然发布单独一个可执行文件是非常简单的操作，就有人会想办法把静态资源文件打包进 Go 的程序文件中。

在 hubble 中使用go-bindata方案。

go-bindata是一个可以把静态资源文件嵌入到 go 文件中，并提供一个操作方法的包。

然后结合go-bindata-assetfs，就可以很方便的使用静态资源了。

7.6.2.1 安装

```
go get -u github.com/kevinburke/go-bindata/go-bindata
```

或者下载https://github.com/kevinburke/go-bindata/releases/download/v*/go-bindata-linux-amd64对应的二进制文件，然后mv go-bindata-linux-amd64 go-bindata, 之后移动到$GOPATH/bin目录下。

7.6.2.2 使用

```
go-bindata -pkg dist$* -o dist/bindata.go -prefix dist$* dist$*/...
```

-pkg 包名 -o 输出文件 -prefix 去除路径

-prefix 去除路径的含义如下：

不加参数：

```
go-bindata /path/to/templates/

_bindata["/path/to/templates/foo.html"] = path_to_templates_foo_html
```

加参数：

```
go-bindata -prefix "/path/to/" /path/to/templates/

_bindata["templates/foo.html"] = templates_foo_html
```

bindata 时可以去掉不必要的路径。

在实例中，使用了前边说的 generate

```
//go:generate go-bindata -pkg dist$* -o dist/bindata.go -prefix \
    dist$* dist$*/... -ignore=\\.go
```

HUBBLE

这样在 build 时可以使用`go generate&&go build`

`http.FileSystem`是定义 HTTP 静态文件服务的接口。go-bindata-assetfs 实现了这个接口，支持 HTTP 访问静态文件目录的行为。详细使用请参考实例的`web/server.go`文件。

目前构建脚本未调整，还需要手动在 bindata.go 的 import 中增加 `"gitee.com/beagledata_1/cobra-example/web"`，在结尾增加 `func init(){ web.Asset = Asset; web.AssetDir = AssetDir; web.AssetInfo = AssetInfo }`。

目的是将`go-bindata.go`中的`Asset`、`AssetDir`、`AsserInfo`与 web 包下的三个函数绑定。

当然，如果是简单写法，直接在使用时用：

```
fs := assetfs.AssetFS{
        Asset:     bindata.Asset,
        AssetDir:  bindata.AssetDir,
        AssetInfo: bindata.AssetInfo,
    }
```

现在的写法的目的是不需要先生成 bindata.go 编译也不会报错。

在`start.go`的 import 中增加`_ "gitee.com/beagledata_1/cobra-example/web/dist"`的目的是调用`go-bindata.go`中的 init。把静态资源加载到 server.go 中。

构建后，`./cobra-example start --addr localhost --port 12345`即可访问到对应的页面了。

更高级的用法就是和 nodejs 结合，做到前后端的分离，前端只需要使用 nodejs 将前端打包成静态文件，然后再与 go 对接即可。

7.7 数据库工作原理

> 本文翻译自How does a relational database work

当提到关系型数据库时，我已经不认为其中有什么缺少的东西了。它们应用于各种场景。现在有着各种不同的数据库：从小型便于使用的 SQLite 到强大的 Teradata。但是，只有少量的文章解释了数据库是如何工作的。你可以搜索“关系型数据库是如何工作的”来看看有多少结果。此外，这些文章都很短。现在，如果你找最新的流行技术（Bigdata、NoSQL 或 JavaScript），你会找到更多深入解释它们是如何工作的文章。

关系型数据库是不是太老了，太无趣了，以至于无法在大学课程、研究论文或书籍外进行解释?

作为一个开发者，我痛恨用而不知，而且，数据库已经被使用了近 40 年了，这一定是有道理的。这些年里，我花费了上百个小时用来真正的理解这些每天我都在用的黑盒。关系型数据库是非常有趣的，因为它们基于高效与可重用的概念。如果你对理解数据库这事感兴趣并想要深入研究这个广泛的主题，但又没有时间，那这篇文章你一定会喜欢。

虽然本文的标题非常明确，但本文的目的并不是用于理解如何使用数据库的。因此，你需要具备 SQL 语句的基本查询知识（如 join，CRUD 语句）；否则你可能无法理解本文，这是你唯一需要具备能力，其他的我都会在本文中有解释。

我将会从计算机科学方面开始，例如时间复杂度。我知道有些人讨厌这个概念，但没有它，你无法理解数据库内在巧妙之处。由于这是一个很大的主题项，我将会专注于那些我觉得是最重要的内容：数据库处理 SQL 查询的方式。我将只介绍数据库背后的基本概念，这样在读完本文后，您就会对底层所发生的事情有一个很好的了解。

由于这是一篇涉及许多算法和数据结构的长篇技术文章，所以请慢慢阅读。有些概念比较晦涩难懂，即使你跳过部分对整体理解不会有影响。

图 68: 关系型数据库列表

为了便于读者理解，这边文章可以至少分为三部分。

- 数据库的底层与高层组件概述
- 查询优化过程概述
- 事务和缓冲池管理的概述

目录

- 数据库工作原理
  - 1. 基础知识
  - 2. 全局概述
  - 3. 客户端管理器
  - 4. 查询管理器
  - 5. 数据管理器
  - 6. 事务管理器
  - 7. 结论

7.7.1 基础知识

很久之前的开发者必须在编码时确切的知道代码中确切的操作次数，他们将算法和数据结构熟记于心，因为他们不能承担浪费 CPU 和内存的代价（硬件成本高昂的时代）。

在这一章节中，我将提及其中的一些概念，因为这些是理解数据库所必需的知识，同时我还会介绍数据库索引的概念。

7.7.1.1 O(1) 和 O(n2)

当下，大部分的开发人员不会关心时间复杂度，或许他们是对的。但当你处理庞大的数据量时（这可不是指的数千），或是为了争取毫秒级的效率时，理解这个概念就变得非常重要了。你得知道，数据库必须处理这两种情况。我不会让你厌烦很长时间，只是稍花点时间理解这个想法，这将有助于我们以后理解“基于成本优化”（CBO）的概念。

概念

例如，当说这个算法在 O(some_function()) 时，它意味着对于一定量的数据，算法需要 some_function(a_certain_amount_of_ 操作来完成这个工作。

重要的不是数据量，而是数据量增加时，操作数增加的方式。时间复杂度并不能给出操作的确切数量，但这是个好理念。

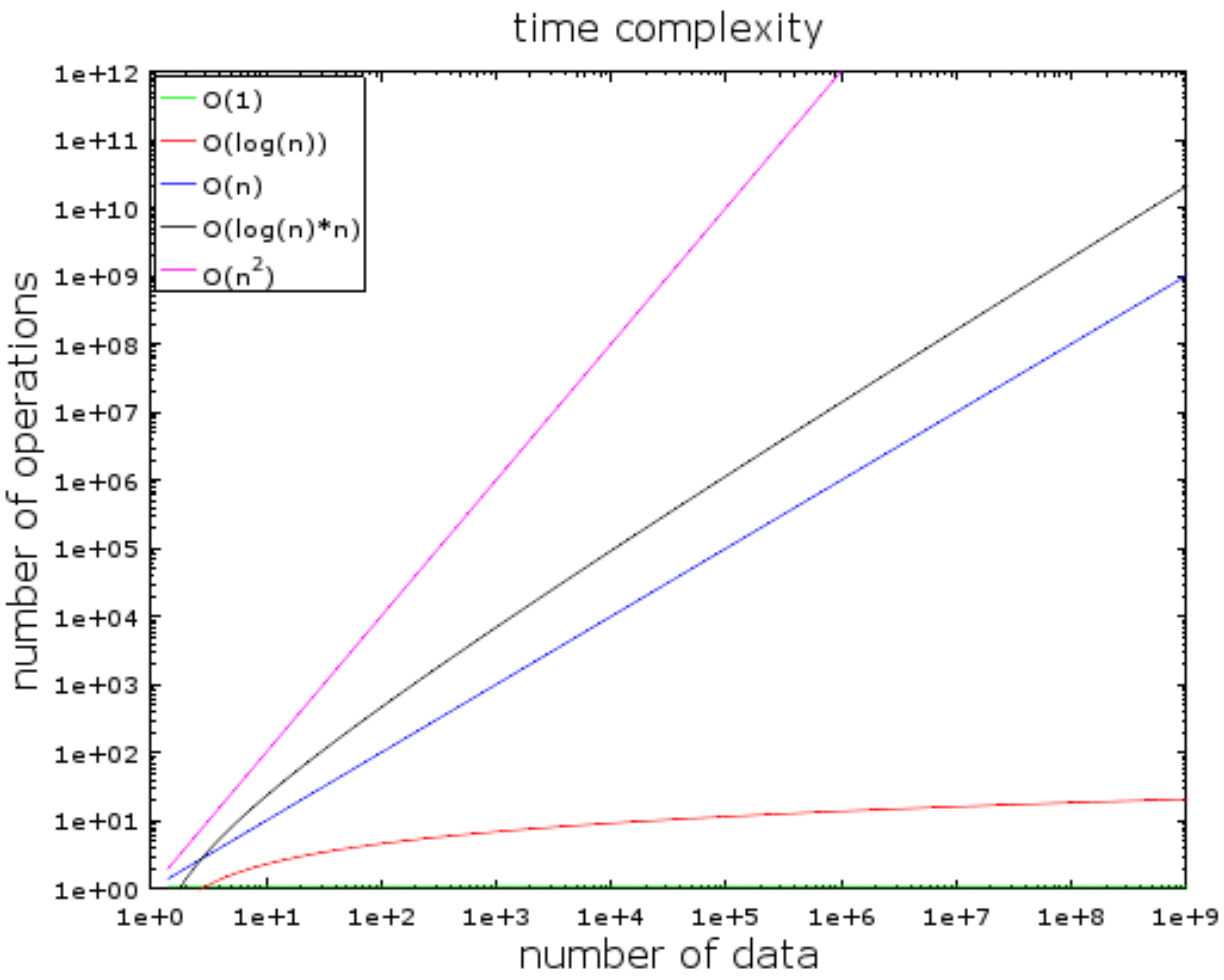


图 69: TimeComplexity.png

在这个图中，您可以看到不同类型复杂度的曲线。我用对数标度来表示。换句话说，数据的数量正在迅速从 10 亿增长到 10 亿。我们可以看到:

- O(1) 或常数复杂度保持不变 (否则就不叫常数复杂度)
- O(log(n)) 即使在数据量为十亿的情况下，也保持稳定。
- 最糟糕的复杂度是 O(n2)，其操作数量会迅速激增
- 另外两个复杂度也增长的很快

例子

在数据量少时，O(1) 和 O(n2) 之间的差异可以忽略不计。假设有一个需要处理 2000 个元素的算法。

- O(1) 算法需要一次操作
- O(log(n)) 算法需要进行 7 次操纵
- O(n) 算法需要进行 2000 次操作
- O(n*log(n)) 算法需要进行 14000 次操作
- O(n2) 算法需要进行 3000000 次操作

O(1) 和 O(n2) 之间的差异似乎很大 (400 万)，但你最多会损失 2 毫秒，只是眨眼的时间。实际上，当前的处理器每秒可以处理数亿次操作。这就是为什么性能优化在许多 IT 项目中不是一个问题。

正如我所说，在面对大数据量时，了解这个概念仍然很重要。如果这一次算法需要处理 1 000 000 个元素 (对于数据库来说不算大):

- O(1) 算法需要一次操作
- O(log(n)) 算法需要进行 14 次操纵
- O(n) 算法需要进行 1000000 次操作
- O(n*log(n)) 算法需要进行 14000000 次操作
- O(n2) 算法需要进行 1000000000000 次操作

常用数据结构的复杂度

- 在一个好的哈希表中进行搜索，获取一个元素的时间复杂度是 O(1)
- 在平衡良好的树中进行搜索，获取一个元素的时间复杂度是 O(log(n))
- 在数组中搜索的时间复杂度是 O(n)
- 最好的排序算法复杂度为 O(n*log(n))
- 一个糟糕的排序算法时间复杂读是 O(n2)

在下面内容中，我们一起来看看这些算法和数据结构。

时间复杂度有多种情况

- 正常情况
- 最好情况
- 最坏情况

时间复杂度通常是指最糟糕的情况

我只讲了时间复杂度，但复杂度也适用于

- 算法对于内存得消耗
- 算法对于磁盘 IO 的消耗

当然也有比 n 的平方更糟糕的复杂度，例如:

- n 的四次方：槽糕透了，一些算法具有这种复杂度，之后我会讲到
- n 的三次方：非常糟糕，下文中会说到具有这种复杂度的算法（事实上很多数据库中都使用了这类算法）
- n 的阶乘：即使是小数据量的情况下，也无法得到预期结果。
- n 的 n 次方：如果用上这种复杂度，那真该扪心自问，IT 行业是否真的适合了

注意：我没有给出大 O 符号的真正意义，只是给出了概念，可以在 wiki 百科上阅读关于这个符号的概念。

7.7.1.2 归并排序

当需要对集合排序时，应该怎么做？什么？你说可以调用 sort() 函数……好的，回答得很好……但是对于数据库，您必须了解 sort() 函数是如何工作的。

有几种很好的排序算法，而我会着重于最重要的：归并排序。你可能现在还不理解为什么排序算法是非常有用的，但在查询优化部分后，你就能理解其原因了。此外，理解归并排序有助于往后理解一个数据库的常规操作，

HUBBLE

那就是合并连接（merge join）

merge

与那些有用的算法一样，归并排序基于一个技巧：将大小为 N/2 的两个排序数组合并到一个 N 元素排序数组只需要 N 次操作。这个操作过程我们称之为归并。

让我们来通过这个简单的例子来理解：

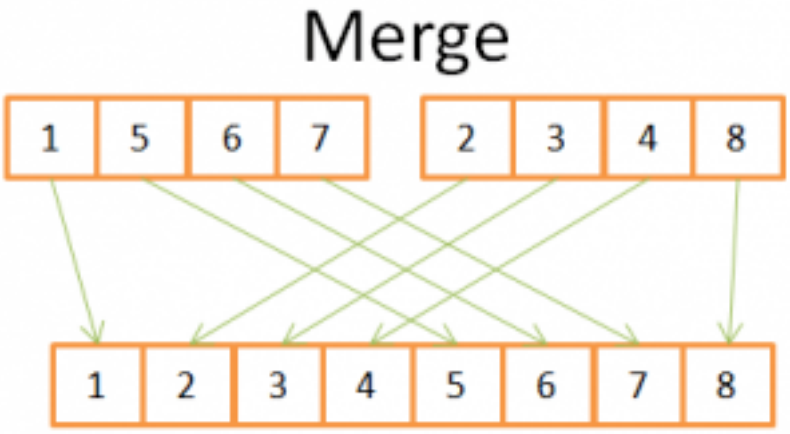


图 70: merge_sort

图中可见，构建一个最终排序的 8 个元素的数组，你只需要将 2 个 4 元素数组迭代一次。这是由于 2 个 4 元素数组都已经被排好序了。

- 1) 比较当前两个数组中的第一个元素
- 2) 然后将最小的元素放入到 8 个元素数组中
- 3) 然后移到你取了最小元素的数组的下一个元素
- 重复上述三步，直到其中一个数组的最后一个元素
- 然后取另一个数组的元素放入 8 个元素的数组中

这是可行的，因为两个 4 元素数组都是排序的，因此你不需要在数组中回移。

现在我们对这个有了解后就可以查看下面的归并排序伪码了：

```
array mergeSort(array a)
   if(length(a)==1)
      return a[0];
   end if

//recursive calls
[left_array right_array] := split_into_2_equally_sized_arrays(a); array
   ↪ new_left_array := mergeSort(left_array);
array new_right_array := mergeSort(right_array);

//merging the 2 small ordered arrays into a big one
array result := merge(new_left_array,new_right_array);
return result;
```

归并排序将问题拆分为较小的问题，然后找到小问题的结果，从而得到最初问题的结果。（注意，这种算法我们称之为分治算法）。如果你不理解这个算法，不要担心；我第一次看的时候还不明白。如果它可以帮助你，我认

为这个算法是一个两阶段的算法:

- 划分阶段，即是将一个数组拆分为更小的数组
- 排序阶段，即将小组数使用归并的方式合并成一个较大的数组

Division Phase

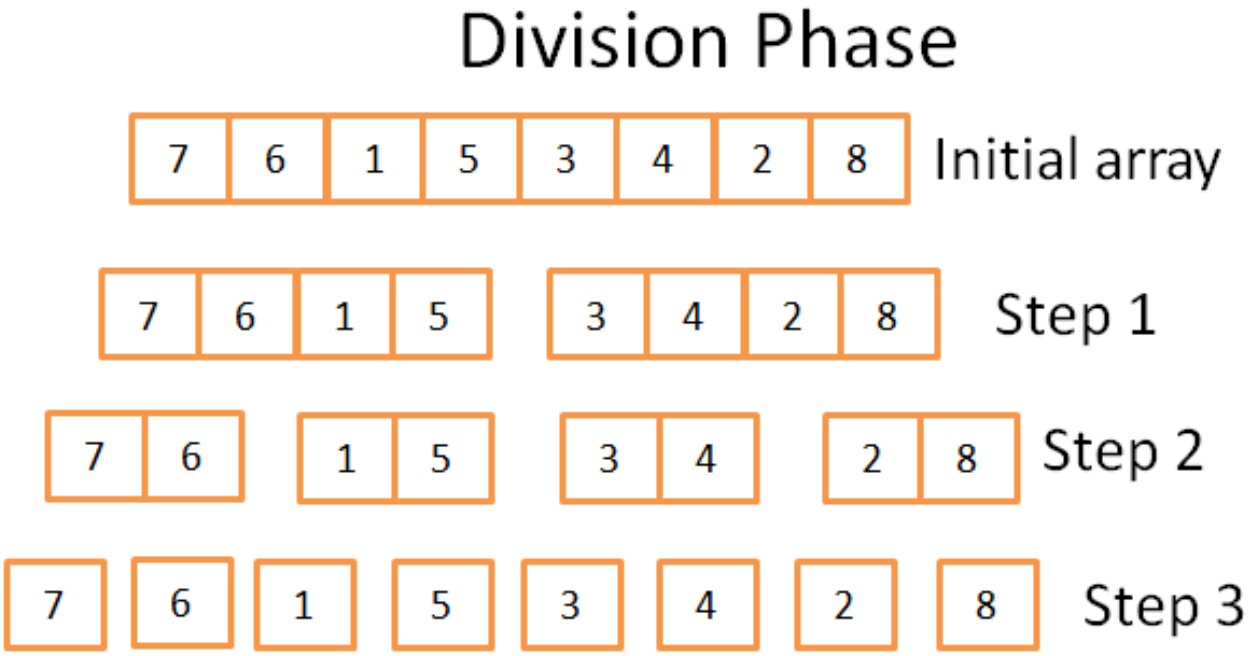


图 71: division_phase

在划分阶段，使用 3 步将数组划分为单元数组。正式的步骤数是 log(N)(因为 N=8，所以 log(N) = 3)。

这是怎么得出的结论？一个词：数学。其思想是每一步将初始数组的大小除以 2. 步骤的数量是将初始数组除以 2 的次数，这就是对数（以 2 为底）的精确定义。

Sorting Phase

在排序阶段，从单个组数开始，在每一步，你接受多个合并，其总成本是 N=8 操作:

- 在第一步中，你有 4 个合并，每个合并需要 2 个操作
- 在第二步中，你有 2 个合并，每个合并需要 4 个操作
- 在第三步，你有一个合并，这需要 8 个操作

归并排序效率

为何这个算法效率这么高?

原因:

- 可以修改它以减少内存占用，方法是不创建新数组，而是直接修改输入数组。(注意: 这种算法称为 in-place)
- 可以修改它，以便同时使用磁盘空间和少量内存，而不会造成巨大的磁盘 I/O 损失。其思想是在内存中只加载当前正在处理的部分。当您需要对一个只有 100 mb 内存缓冲区进行多个 GB 级的表进行排序时，这一点非常重要。(注意：这种算法我们称之为 external sorting)
- 您可以修改它以在多个进程/线程/服务器上运行。(例如，分布式归并排序是 [Hadoop] 的关键组件之一)
- 这个算法可以点石为金

大多数数据库都使用这种排序算法，但它不是惟一的。如果您想了解更多信息，可以阅读这篇讨论数据库中常见排序算法的优缺点的研究论文。

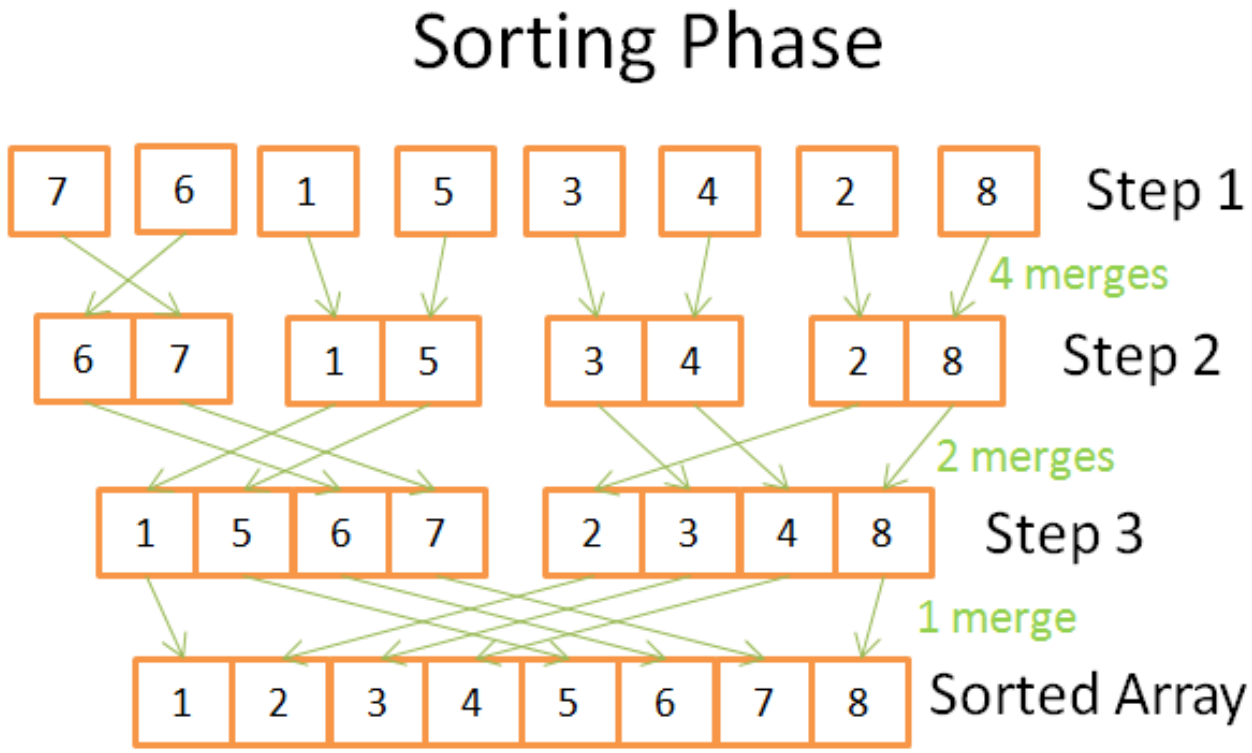


图 72: sorting_phase

7.7.1.3 Array, Tree and Hash table

既然我们已经理解了时间复杂度和排序背后的思想，我必须告诉你 3 种数据结构。这很重要，因为它们是现代数据库的基础。接下来我还将介绍数据库索引的概念。

7.7.1.3.1 Array

二维数组是最简单的数据结构。表可以看作一个数组。例如:

这个二维数组是一个包含行和列的表:

- 每一行代表一个主题
- 列描述主题的特征。
- 每个列存储特定类型的数据 (整数、字符串、日期…)。

虽然存储和可视化数据很好，但是当您需要查找特定的值时，它就很糟糕了。

例如，**如果你想找到所有在英国工作的人**，你必须查看每一行，看看这一行是否属于英国。这将消耗你 N 个操作 (N 是行数)，这并不坏，但有没有更快的方法? 这就是树结构发挥作用的地方。

注意: 大多数现代数据库提供高级数组来有效地存储表，如堆组织的表或索引组织的表。但这并没有改变在一组列上快速搜索特定条件的问题。

7.7.1.3.2 树和数据库索引

二叉搜索树是具有特殊属性的二叉树，每个节点的键必须是:

- 大于存储在左子树中的所有键
- 小于存储在右子树中的所有键

我们来直观地看看这是什么意思。

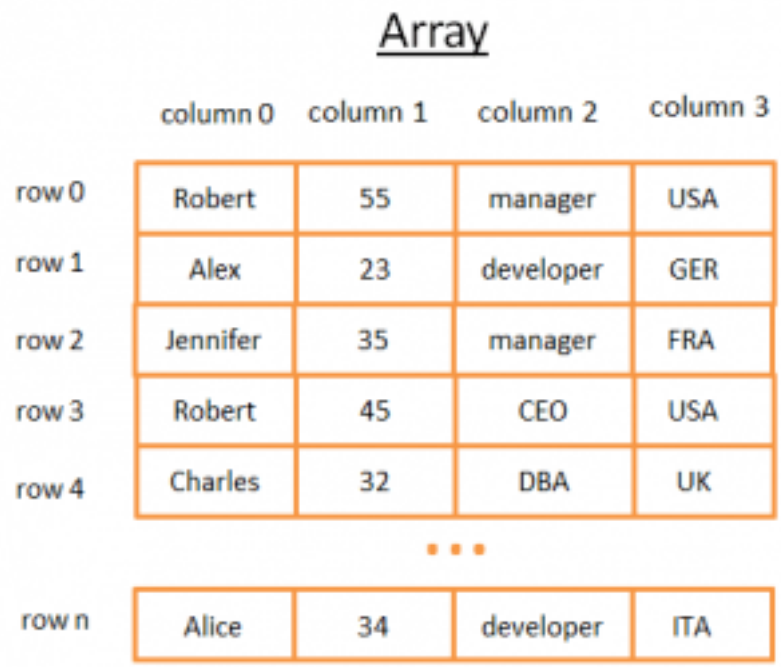


图 73: array

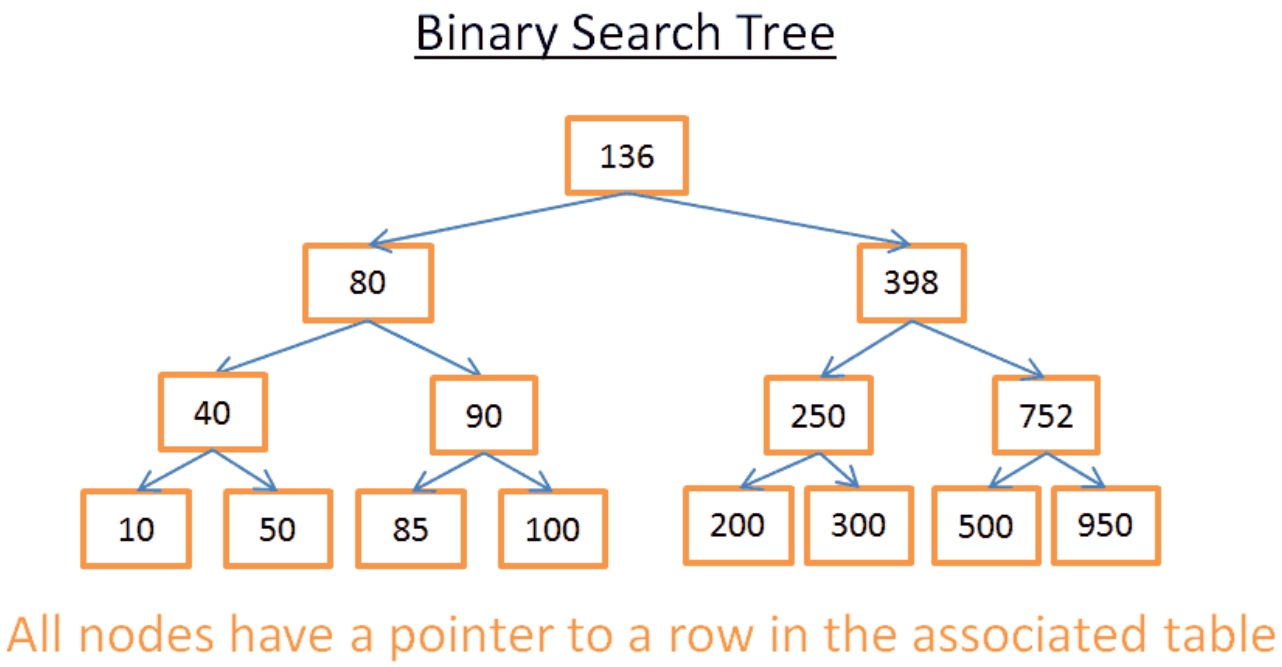


图 74: bst

这个数有 15 个元素，让我们来检索下 208 试试：

- 我从键值为 136 的根开始。因为 136<208，所以我查看节点 136 的右子树。
- 因为 398>208，我查看节点 398 的左子树
- 因为 250>208，我查看节点 250 的左子树
- 因为 200<208，我查看节点 200 的左子树。但是 200 没有右子树，这个值不存在 (因为如果它存在，它应该是 200 节点的子树)

现在我们再来检索下 40 试试：

- 我从键值为 136 的根开始。因为 i136>40 开始，我查看节点 136 的左子树。
- 因为 80>40，我查看节点 80 的左子树
- 40= 40，节点存在，我提取节点内的行 id(它不在图中)，并查看表中给定的行 id。
- 知道行 id 让我知道数据在表上的确切位置，因此我可以立即获得它。

最后，这两种搜索都消耗了树内的层数。如果您仔细阅读归并排序的部分，您应该会看到有 log(N) 层。所以二叉树搜索的代价是 log(N)，还不错!

但是这个东西很抽象，所以我们回到我们的问题。如果不是整数，而是表示上表中某个国家的字符串。假设你有一个包含表中“国家”列的树：

- 如果你想知道谁在英国工作
- 你如何查看树得到代表英国的节点
- 在“UK 节点”中，您将找到英国工人的行位置。

这个搜索仅需要 log(N) 操作，而不是 N 个操作（如果直接使用数组的话）。你刚才想象的就是一个**数据库索引**。

你可以建立一个树索引对于任何一组列，只要你有一个函数比较键，用于键的顺序。

B+ 树索引

虽然这个树在获取特定值时非常好用，但是当你需要在两个值之间获取多个元素时，就有大问题了。它将花费 O(N)，因为您必须查看树中的每个节点，并检查它是否位于这两个值之间 (例如，使用树的顺序遍历)。此外，由于你需要遍历整课树，这对于磁盘读写是不友好的。我们需要找到一个方法来有效的做范围查找。为了解决这个问题，现代数据库使用以前的树的一个修改版本，称为 B+ 树。

在 B + 树中:

- 只有最低节点 (叶节点) 存储信息 (相关表中行位置)
- 其他节点只是用于在搜索过程中，路由到正确的节点。

可以看到，节点更多 (多了两倍)。实际上，您还有其他节点，即“决策节点”，它将帮助您找到正确的节点 (在关联的表中存储行位置)。但是搜索复杂度仍然是 O(log(N))。最大的区别是，最低的节点链接到它们的后续的节点。

在这个 B+ 树中，如果你想找 40 到 100 之间的值:

- 你只需要寻找 40(或者 40 之后的最近的值，如果 40 不存在的话)，就像之前的树一样。
- 然后收集 40 的继任者使用直接链接到后面的节点，直到你达到 100。

假设有 M 个叶子结点，树有 N 个结点。对特定节点的搜索开销与前一棵树一样是 log(N)。但使用这种树时，一旦定位一个节点，你可以通过 M 次操作获取 M 个节点。这个搜索的开销是 M+log（N）次操作，相比 N 次操作（即二叉树）。此外，你不需要读取整课树，这意味着更少的磁盘操作。如果 M 很低 (比如 200 行) 而 N 很大 (1 000 000 行)，则会有很大的不同。

但新的问题又出现了。如果在数据库中添加或删除一行 (在相关的 B+ 树索引中):

- 你必须保持在 B+ 树的节点之间的顺序，否则你将无法通过无序的节点来查找。

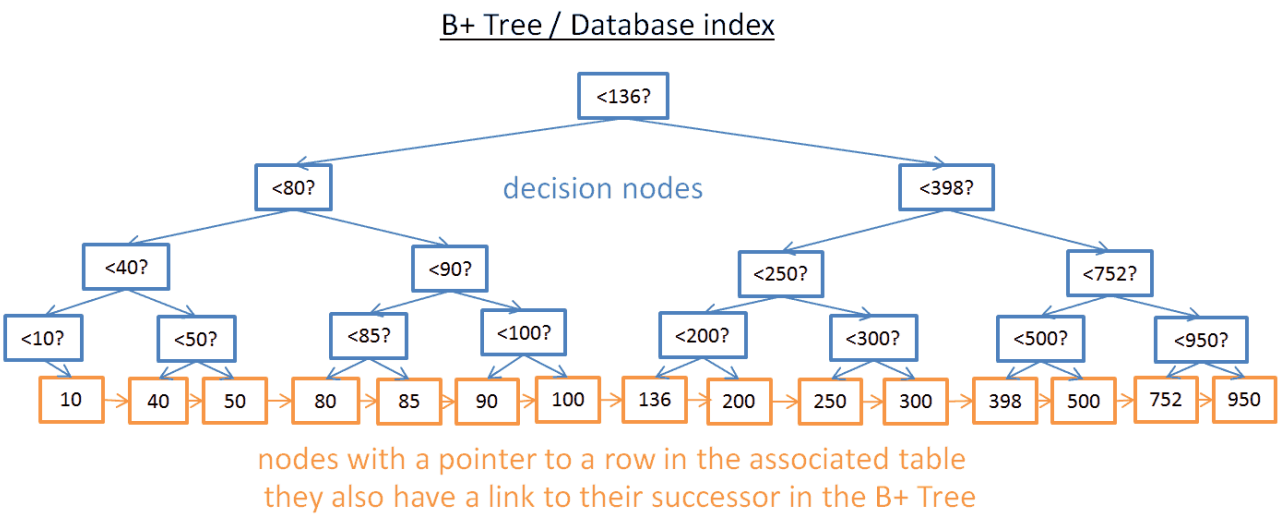


图 75: database index

- 你必须在 B+ 树中保持尽可能少的层级数，否则 O(log(N)) 中的时间复杂度将变成 O(N)。

换句话说，B+ 树需要是自有序和自平衡的。幸运的是，通过智能删除和插入操作，这是可能的。但是这是有代价的: 在 B+ 树中的插入和删除的开销在 O(log(N))。这就是为什么有些人听说使用太多索引不是一个好主意。因为这样减慢了表中行的快速插入/更新/删除，当数据库需要更新表的索引时，每个索引都需要执行代价高昂的 O(log(N)) 操作。此外，添加索引意味着事务管理器的工作量被增加了 (我们将在本文的最后介绍这个管理器)。

更多的细节，你可以看看维基百科关于 B+ 树的文章。如果你想要一个 B + 树在数据库方面的实现，看看这两篇文章 (http://blog.jcole.us/2013/01/07/the-physical-structure-of-innodb-index-pages/) 和 (http://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/) 这两篇文章来自 MySQL 的核心开发人员。它们都是关于 innoDB (MySQL 的引擎) 如何处理索引的文章。

7.7.1.3.3 Hash Table

最后一个重要的数据结构是哈希表。当您希望快速查找值时，它非常有用。此外，理解哈希表将帮助我们以后理解一个名为 hash join 的常规的数据库连接操作。数据库也使用此数据结构来存储一些内部内容 (如锁表或缓冲池，稍后我们将看到这两个概念)

哈希表是这种数据结构，可以快速使用键来找到的元素，要构建一个哈希表，你需要定义:

- 元素的键
- 键的哈希算法。用于计算键的散列值，从而得知元素的位置 (称为 bucket)
- 一个比较键的函数，一旦你找到正确含有你需要指定查找元素的 bucket，就需要使用比较方法在 bucket 中找到元素。

一个简单例子

我们来看一个可视化例子:

这个哈希表有 10 个 buckets。因为我很懒，我只画了 5 个桶但是我知道你很聪明，所以我让你想象其他 5 个。我使用的哈希函数是键的模 10。换句话说，我只保留一个元素的键的最后一位来找到它的 bucket:

- 如果元素的最后一位数字是 0，则该元素最终在 bucket 0 中
- 如果元素的最后一位数字是 1，则该元素最终在 bucket 1 中
- 如果元素的最后一位数字是 2，则该元素最终在 bucket 2 中
- ...

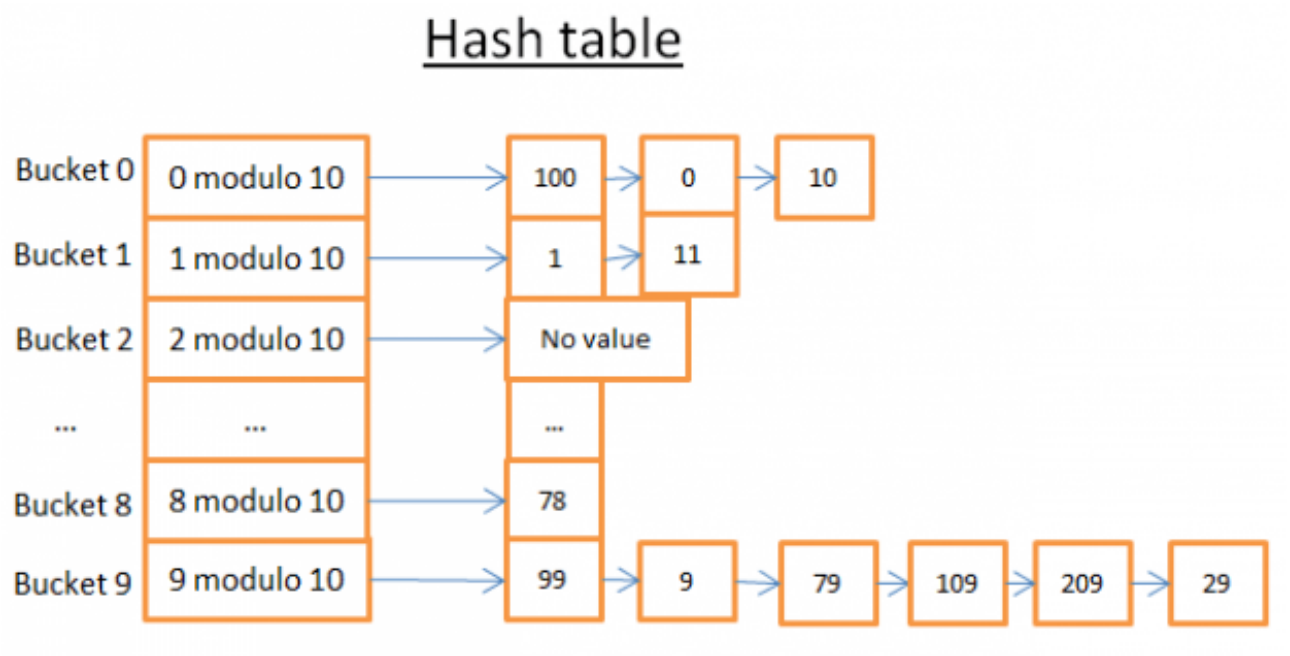


图 76: hash table

我使用的比较函数就是两个整数之间的等式。

假设你想要得到元素 78:

- 哈希表计算的哈希码为 78，即 8。
- 它在桶 8 中查找，它找到的第一个元素是 78。
- 它会返回 78 元素
- 搜索仅花费 2 个操作 (1 个用于计算哈希值，另一个用于查找桶内的元素)。

现在，假设你想要得到元素 59:

- 哈希表计算的哈希码为 59，即 9
- 在 bucket 9 中查找，它找到的第一个元素是 99。自 99 年以来!=59，元素 99 不是正确的元素。
- 使用相同的逻辑，它查看第二个元素 (9)、第三个元素 (79)、…和最后一个元素 (29)。
- 元素不存在。
- 搜索使用了 7 个操作

好的 hash 函数

正如您所看到的，根据您所寻找的值，代价是不一样的!

如果我现在用键的 1 000 000 的模 (即取最后 6 位数字) 来做为哈希函数，那么第二次搜索只需要一个操作，因为在 bucket 000059 中没有元素。真正的挑战是找到一个好的哈希函数，它将创建包含非常少量元素的 bucket。

在我的例子中，找到一个好的哈希函数是很容易的。但这是一个简单的例子，当键是如下情况时，找到一个好的哈希函数是有难度的:

- 一个字符串 (例如一个人的姓)
- 2 个字符串 (例如一个人的姓和名)
- 2 个字符串和一个日期 (例如一个人的姓、名和出生日期)

使用一个高效的 hash 函数，在 hash 表中查找元素的复杂度是 O(1)

数组和 hash 表

HUBBLE

为什么不使用数组呢?

嗯，你问了一个好问题。

- 哈希表可以在内存中加载一半，其他桶可以留在磁盘上。
- 当使用数组时，你必须使用一个连续的空间在内存。如果您正在加载一个大的表，那么拥有足够的连续的内存空间是非常困难的。
- 使用哈希表，您可以选择您想要的键 (例如国家和一个人的名字)。

要了解更多信息，您可以阅读文章Java HashMap ，这是一个高效的散列表实现; 您不需要了解 Java 就可以理解本文中的概念。

7.7.2 全局概述

我们已经看到了数据库中的基础组件。我们现在需要退一步来看看全局。

数据库是一组可以方便地访问和修改的信息。但是一堆简单的文件也可以做到这一点。事实上，像 SQLite 这样最简单的数据库只不过是一堆文件。但是 SQLite 是一组精心制作的文件，因为它允许您:

- 使用事务以确保数据安全和一致
- 快速处理数据，即使你正在处理数百万的数据

一般而言，数据库可以有如下部分:

在写这一部分之前，我已经阅读了许多书籍/论文，以及各种描述讨论数据库相关的资料。因此，不要过多地关注我如何组织这个数据库或如何命名这些过程，因为我做了一些选择来适应本文的规划。重要的是不同的组成部分，其中心思想是数据库如何划分为多个相互交互的组件。

核心组件:

- 进程管理器: 许多数据库都有一个用于管理的进程/线程池。此外，为了获得纳秒级的速度，一些现代数据库使用自己的线程而不是操作系统线程。
- 网络管理器: 网络 I/O 是一个大问题，特别是对于分布式数据库。这就是为什么有些数据库有自己的管理器。
- 文件系统管理器: 磁盘 I/O 是数据库的第一个瓶颈。拥有一个能够完美地处理操作系统文件系统甚至替换它的管理器是很重要的。
- 内存管理器: 为了避免磁盘 I/O 的损失，需要大量的随机存储内存。但如果处理大量内存，则需要高效的内存管理器。特别是当您同时有许多使用内存的查询时。
- 安全管理器: 用于管理用户的身份验证和授权
- 客户端管理器: 用于管理客户端连接

工具:

- 备份管理器: 用于保存和恢复数据库。
- 恢复管理器: 用于在崩溃后，将数据库重新启动恢复到一致性的状态
- 监控管理器: 用于记录数据库的活动并提供监视数据库的工具
- 管理员管理器: 用于存储元数据 (如表的名称和结构)，并提供工具来管理数据库、模式、表空间……

查询管理:

- 查询解析器: 检查查询是否有效
- 查询重写器: 预先优化一个查询
- 查询优化器: 优化一个查询
- 查询优化器: 编译和执行一个查询

数据管理:

- 事务管理器: 处理事务
- 缓存管理器: 在使用数据之前将数据放入内存，在将数据写入磁盘之前将数据放入内存
- 数据访问管理器: 访问磁盘上的数据

在本文的其余部分，我将重点介绍数据库如何通过如下过程管理 SQL 查询:

- 客户端管理器
- 查询管理器
- 数据管理器 (在本部分中我还将包括恢复管理器)

7.7.3 客户端管理器

客户端管理器是处理与客户端通信的部分。客户端可以是 (web) 服务器，也可以是终端用户/终端应用程序。客户端管理器提供了通过一组通用的 api 做为访问数据库的不同方方式:JDBC、ODBC、OLE-DB…

它还可以提供专用的数据库访问 api。

当你连接到一个数据库:

- 管理员首先检查您的身份验证 (您的登录名和密码)，然后检查您是否有授权使用数据库。这些访问权限是由 DBA 设置的。
- 然后，它检查是否有一个进程 (或线程) 可用来管理您的查询。
- 它还检查数据库是否处在在重负载状态。
- 它可以等待片刻，以获得所需的资源。如果此等待超时，则关闭连接并给出可读的错误消息。
- 然后它发送您的查询到查询管理器，并且此时您的查询将被处理
- 由于查询处理不是一个“全部或没有”的事情，一旦它从查询管理器获取数据，它将部分结果存储在一个缓冲区，并开始发送给你。
- 如果出现问题，它会停止连接，给你一个可读的解释，并释放资源。

7.7.4 查询管理器

这部分是数据库的强大之处。在这一部分中，一个写得不好的查询被转换成一个快速可执行代码。然后代码将被执行，并且结果会被返回给客户端管理器。这是一个多步骤的操作:

- 首先对查询进行解析，以查看它是否有效
- 然后将会重写，用于删除无用的操作，并添加一些预优化
- 然后查询将被优化以提高性能，并转换为执行和数据访问计划。
- 然后这个计划被编译
- 最后，它被执行

读完这部分后，如果你想更好的理解，我推荐你阅读:

- 关于 CBO 的最初研究论文 (1979): 关系数据库管理系统中的访问路径选择。这篇文章只有 12 页，并且只有有计算机科学的平均水平就可以理解。

  https://www.cs.berkeley.edu/$_{\text{brewer}}$/cs262/3-selinger79.pdf

- 一篇非常好的和深入的，关于 DB2 9.x 的查询优化的的介绍。

  http://infolab.stanford.edu/$_{\text{hyunjung}}$/cs346/db2-talk.pdf

- 一篇关于 PostgreSQL 如何优化查询的很好的介绍。这是最易访问的文档，因为它更多的是关于“让我们看看 PostgreSQL 在这些情况下提供了什么查询计划”，而不是“让我们看看 PostgreSQL 使用的算法”。

  http://momjian.us/main/writings/pgsql/optimizer.pdf

HUBBLE

- 关于优化的官方 SQLite 文档。它很容易阅读，因为 SQLite 使用简单的规则。此外，这是唯一的官方文件，真正解释了它是如何工作的。

  https://www.sqlite.org/optoverview.html

- 关于 SQL Server 2005 如何优化查询的一个很好的介绍

  https://blogs.msdn.com/cfs-filesystemfile.ashx/___key/communityserver-components-postattachments/00-08-50-84-93/QPTalk.pdf

- 关于 Oracle 12c 优化器的白皮书

  http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-optimizer-with-oracledb-12c-1963236.pdf

- 两位作者关于查询优化的理论课程。很好的着重描述磁盘 I/O 卡西奥，是计算机科学所必需的。

  http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/PPT-dir/ch12.ppt

  http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/PPT-dir/ch13.ppt

- 另一个理论课程，我发现它更容易阅读，但它只关注连接操作符和磁盘 I/O。

  https://www.informatik.hu-berlin.de/de/forschung/gebiete/wbi/teaching/archive/sose05/dbs2/slides/09_joins.pdf

7.7.4.1 查询解析

每个 SQL 语句都被发送到解析器，在那里检查语法是否正确。如果您在查询语句有问题，解析器将拒绝该查询。例如，如果你写的是“SLECT…”而不是“SELECT…”，故事到此结束。

但除此以外。它还检查关键字是否按正确的顺序使用。例如，where 在 select 之间的查询语句被拒绝。

然后，分析查询中的表和字段。解析器使用数据库的元数据来检查:

- 表是否存在
- 字段是否存在
- 字段类型的操作是否正确 (例如，您不能将整数与字符串进行比较，您不能在整数上使用 substring() 函数)

然后，它将检查您是否具有读取 (或写入) 查询中的表的授权。同样，这些表上的访问权限是由 DBA 设置的。

在此解析过程中，SQL 查询被转换为内部表达式 (通常是树)

如果一切正常，则将内部表达式发送给查询改写器。

7.7.4.2 查询改写器

在这一步，我们有一个查询的内部表示式。改写的目的是:

- 预先优化查询
- 避免不必要的操作
- 帮助优化找到最好的可能的解决方案

重写器对查询执行一个已知规则列表。如果查询符合规则的模式，那规则将被应用于重写查询。以下是一个非详尽的 (可选) 规则列表:

- 视图合并: 如果您在查询中使用视图，则视图将使用视图的 SQL 代码进行转换。
- 子查询扁平化: 优化子查询非常困难，所以重写器将尝试使用子查询修改查询以删除子查询。

例如:

```
SELECT` `PERSON.*``FROM` `PERSON``WHERE` `PERSON.person_key ``IN``(``SELECT` `
    ↪ MAILS.person_key``FROM` `MAILS``WHERE` `MAILS.mail ``LIKE` `'christophe
    ↪ %'``);
```

将会被替换为:

```
SELECT` `PERSON.*``FROM` `PERSON, MAILS``WHERE` `PERSON.person_key = MAILS.
    ↪ person_key``and` `MAILS.mail ``LIKE` `'christophe%'``;
```

- 删除不必要的操作符: 例如，如果您使用 DISTINCT 在一个有唯一的约束的数据，那么 DISTINCT 关键字将被删除。
- 冗余连接消除: 如果你有两个相同的连接条件，因为一个连接条件隐藏在一个视图中，或者根据传递性有一个无用的连接，它被删除。
- 常量算术计算: 如果你写的东西需要微积分，那么它在重写过程中计算一次。例如，将 age >10+2 转换为 age>12 和 TODATE("some date") 转换为日期时间格式的日期
- (高级) 分区剪裁: 如果您使用的是分区表，则重写器能够找到要使用的分区。
- (高级) 物化视图重写: 如果您有一个物化视图，它匹配查询中谓词的一个子集，则重写器检查该视图是否是最新的，并修改查询以使用物化视图而不是原始表。
- (高级) 自定义规则: 如果您有自定义规则来修改查询 (如 Oracle 策略)，则重写器将执行这些规则
- (高级)Olap 转换: 分析/窗口函数，星型连接，rollup…也将被转换 (但我不确定这是由重写器或优化器完成的，因为这两个过程非常接近，它取决于数据库)。

然后，将这个重写的查询发送给查询优化器，这才是有趣开始!

7.7.4.3 统计

在描述一个数据库如何优化一个查询之前，我们需要谈谈统计数据，因为没有的话，一个数据库是愚蠢的。如果你不告诉数据库去分析它自己的数据，它就不会去做，而且会做出（非常）糟糕的假设。

但是数据库需要什么样的信息呢?

我必须 (简要地) 谈一下数据库和操作系统如何存储数据。他们使用的最小单位，我们称之为一页或一个块 (默认为 4 或 8KB)。这意味着，如果你只需要 1KB，它将无论如何都消耗一页。如果一页占用 8KB，那么将浪费 7KB。

回到统计数据! 当您要求数据库收集统计数据时，它计算的值如下:

- 一个表包含的行/页数
- 对于表中的每一列:
  - 不同的数据值
  - 数据值的长度 (min、max、average)
  - 数据范围信息 (最小、最大、平均)
- 表的索引信息。这些统计数据将帮助优化器估计查询的磁盘 I/O、CPU 和内存使用量

每一列的统计数据非常重要。例如，如果需要在表 PERSON 上 JOIN 两个列: LAST_NAME、FIRST_NAME。通过统计数据，数据库知道FIRST_NAME上只有 1,000 个不同的值，LAST_NAME上有 1,000,000 个不同的值。因此，数据库将以 LAST_NAME，FIRST_NAME 的顺序关联，而不是 FIRST_NAME，LAST_NAME 顺序的方式，因为它产生的比较要少得多，因为LAST_NAME不太可能是相同的，所以大多数情况下，对 LAST_NAME 的前 2(或 3) 个字符进行比较就足够了。

但这些都是基本的统计数据。您可以要求数据库计算称为柱状图的高级统计数据。柱状图是关于列内值分布的统计信息。例如

- 最频繁出现值
- 分位数
- …

这些额外的统计信息将帮助数据库找到更好的查询计划。特别是对于相等谓词 (例如 where AGE = 18) 或范围谓词 (例如 where AGE > 10 and AGE <40)，因为数据库可以更好地了解这些谓词所涉及的行数 (注意: 这个概念的技术术语是 selectivity)。

统计数据存储在数据库的元数据中。例如，你可以看到 (非分区) 表的统计数据:

- 在 Oracle 中的 `USER/ALL/DBA_TABLES`和`USER/ALL/DBA_TAB_COLUMNS`
- 在 DB2 中的 SYSCAT.TABLES 和 SYSCAT.COLUMNS。

统计数字必须是最新的。没有什么比数据库认为一个表只有 500 行而它却有 100 万行更糟糕的了。统计数据的唯一缺点是计算它们需要时间。这就是大多数数据库默认情况下不会自动计算它们的原因。数百万的数据使得计算它们变得很困难。在这种情况下，可以选择只计算基本统计信息，也可以选择计算数据库样本的统计信息。

例如，当我在一个项目中处理每个表中的数亿行时，我选择只计算 10% 的统计数据，这带来了巨大的时间收益。就本文来说，这是一个糟糕的决定，因为 Oracle 10G 为特定表的特定列选择的 10% 有时与 100% 非常不同 (对于具有 100 万行的表，这种情况不太可能发生)。这个错误的统计数据导致查询有时花费 8 小时而不是 30 秒; 找到根本原因是一场噩梦。这个例子显示了统计数据的重要性。

注意: 当然，每个数据库都会有更高级的统计信息。如果您想了解更多，请阅读数据库的文档。话虽如此，我还是试图了解统计数据是如何使用的，我找到的最好的官方文档是来自 PostgreSQL。

7.7.4.4 查询优化器

所有现代数据库都使用基于**成本的优化** (或 **CBO**) 来优化查询。其思想是为每个操作设置一个成本，并通过使用最便捷的操作链来获得结果，从而找到降低查询成本的最佳方法。

为了理解 CBO 是如何工作的，我认为最好有一个例子来“感受”这个任务背后的复杂性。在这一部分中，我将介绍连接两个表的 3 种常见方法，我们很快就会发现，即使是一个简单的连接查询也很难优化。之后，我们将看到真正的优化器是如何完成这项工作的。

对于这些连接，我将关注它们的时间复杂度，但是数据库优化器计算它们的 CPU 开销、磁盘 I/O 开销和内存占用。时间复杂度和 CPU 成本之间的区别是，时间成本非常接近 (对于像我这样的懒家伙来说)。对于 CPU 开销，我应该计算每一个操作，比如加法、“if 语句”、乘法、迭代……

- 每个高级代码操作都有特定数量的底层的 CPU 操作。
- CPU 操作的开销是不一样的 (就 CPU 周期而言)，无论你使用的是英特尔酷睿 i7，英特尔奔腾 4,AMD Opteron…换句话说，它取决于 CPU 架构。

利用时间复杂性更容易 (至少对我来说是这样)，使用它，我们仍然可以得到 CBO 的概念。我有时会讨论磁盘 I/O，因为它是一个重要的概念。请记住，瓶颈主要是磁盘 I/O，而不是 CPU 使用量。

7.7.4.4.1 indexes

我们在 B+ 树章节中，已经讨论了索引。只需要记住这些索引已经排好序了。

仅供参考，还有其他类型的索引，如**位图索引**。它们在 CPU、磁盘 I/O 和内存方面提供的成本与 B+ 树索引不同。

此外，如果能够改善执行计划的成本，许多现代数据库可以**动态地为当前查询创建临时索引**。

Access Path

在应用连接操作符之前，首先需要获取数据。以下是获取数据的方法。

注意: 由于所有访问路径的真正问题是磁盘 I/O，所以我不会过多地讨论时间复杂性。

Full scan

如果你曾经读过一个执行计划，你一定见过一个词，全扫描 (或者只是扫描) 这个词。全表扫描就是数据库完全读取一个表或一个索引。在磁盘 I/O 方面，表的全扫描显然比索引的全扫描开销更为大。

Range scan

还有其他类型的扫描，如索引范围扫描。例如，当您使用诸如 “WHERE AGE > 20 AND AGE <40” 之类的谓词时，可以使用它。

当然，您需要在字段 AGE 上有一个索引来使用这个索引范围扫描。

我们在第一部分中已经看到，范围查询的时间成本类似于 log(N) +M，其中 N 是该索引中的数据数量，M 是对该范围内的行数的估计。得益与统计数据，N 和 M 值都是已知的 (注意:M 是谓词 AGE >20 和 AGE<40 的选择性)。此外，对于范围扫描，您不需要读取完整的索引，因此就磁盘 I/O 而言，它的成本比全表扫描要低。

Unique scan

如果你只需要索引中的一个值，你可以使用唯一扫描。

Access by row id

大多数情况下，如果数据库使用索引，它将不得不查找与索引关联的行。为此，它将使用 row id 进行访问。

例如，做了如下操作

```
SELECT LASTNAME, FIRSTNAME from PERSON WHERE AGE = 28
```

有的 person 表有 age 列索引, 优化器将使用索引来查找所有 age 为 28 的元素，因为索引只有 age 的信息，而想知道 lastname 和 firstname，就需要它关联到表中的行。

但如果你进行如下操作

```
SELECT TYPE_PERSON.CATEGORY from PERSON ,TYPE_PERSON WHERE PERSON.AGE =
    ↪ TYPE_PERSON.AGE
```

PERSON 上的索引将用于与 TYPE_PERSON 连接，但是 PERSON 表不会通过 row id 访问，因为您没有询问这个表上的信息。

尽管它对于一些访问非常有效，但是这种操作的真正问题是磁盘 I/O。如果您需要太多的 row id 访问，数据库可能会选择全扫描。

Others paths

我没有给出所有的访问路径。如果您想了解更多信息，可以阅读Oracle 文档。其他数据库所使用的名称可能不相同，但其背后的概念是相同的。

7.7.4.4.2 Join 操作

前边讲了如何获取数据，下面来看看如果进行 Join。

我将介绍 3 种常见的连接操作:merge join、hash join 和 nested loop join。但是在这之前，我需要介绍一些新的词汇: 内部关系和外部关系。一个关系可以是:

- 一个表
- 一个索引
- 前一个操作的中间结果 (例如前一个连接的结果)

当连接两个关系时，join 算法以不同的方式管理这两个关系。在本文的其余部分，我将假设:

- 外部关系是左数据集
- 内部关系是右数据集

例如，A join B 是 A 和 B 之间的连接，其中 A 是外部关系，B 是内部关系。

大多数情况下，A JOIN B 的成本与 B JOIN A 的成本是不一样的

在这一部分中，我还假设外部关系有 N 个元素，内部关系有 M 个元素。请记住，真正的优化器通过统计信息知道 N 和 M 的值。

注意:N 和 M 是关联的基数。

Nested loop join

Nested loop join 是最简单的。

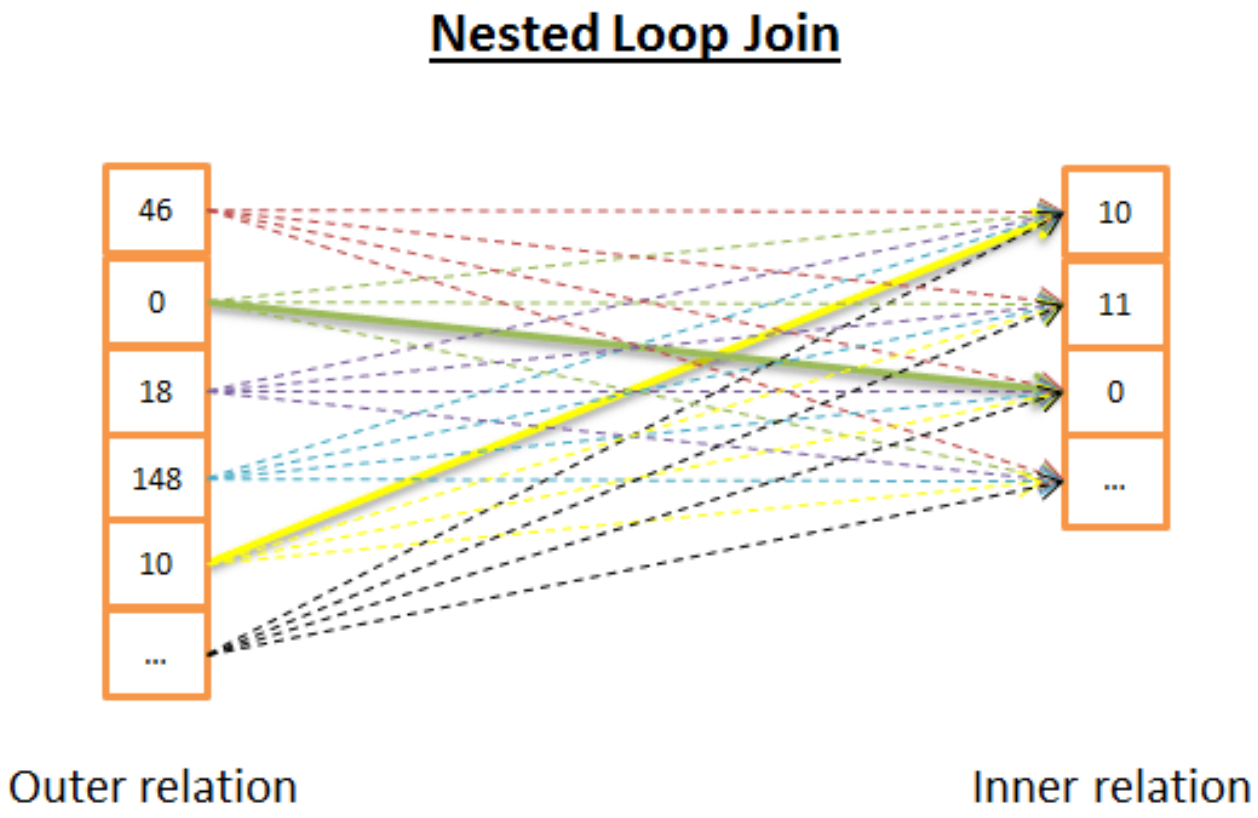


图 77: nestedloop

概念如下:

- 遍历外关联的每一行
- 查看内关联中的所有行，查看是否有匹配的行

伪码如下:

```
nested_loop_join(array outer, array inner)
  for each row a in outer
    for each row b in inner
      if (match_join_condition(a,b))
        write_result_in_output(a,b)
      end if
```

```
    end for
  end for
```

由于这是嵌套迭代，时间复杂度是 O(N*M)

对于磁盘 I/O，在循环读取外部关联中的 N 行时，内部循环需要从内部关联中读取 M 行。这个算法需要从磁盘中读取 N + N*M 行。但是，如果内部关系足够小，你可以把这个关系放到内存中，只需要读取 M +N 次。通过这种调整，内部关系必须是最小的关系，因为它有更多的机会来适应内存。

在时间复杂度方面没有区别，但在磁盘 I/O 方面，最好是一次读取这两个关系。

当然，内部关系可以用索引代替，这对磁盘 I/O 更有益。

由于这个算法非常简单，所以如果 inner relation 太大而无法装入内存，这里有另一个版本的磁盘 I/O 更友好。这个想法是这样的:

- 不是逐行读取两个关系，
- 以小批量的方式地读，并在内存中保存小批量数据，
- 比较两个批量数据里面的行，并保持匹配的行，
- 然后从磁盘加载新的批量数据，并进行比较
- 如此类推，直到没有批量数据可以加载。

这里是一个可行的算法:

```
// improved version to reduce the disk I/O.
nested_loop_join_v2(file outer, file inner)
  for each bunch ba in outer
  // ba is now in memory
    for each bunch bb in inner
    // bb is now in memory
      for each row a in ba
        for each row b in bb
          if (match_join_condition(a,b))
            write_result_in_output(a,b)
          end if
        end for
      end for
    end for
  end for
```

在这个版本中，时间复杂度保持不变，但是磁盘访问的数量减少了:

- 在前一个版本中，算法需要 N + N*M 次访问 (每次访问一个行)。
- 有了这个新版本,访问磁盘的次数变成了 number_of_bunches_for(outer)+ number_of_bunches_for(outer)* number_of_bunches_for(inner)。
- 如果你增加批量读取的大小，就能减少了磁盘访问的数量。

注意: 与以前的算法相比，每个磁盘访问会收集更多的数据，但这并不重要，因为它们是顺序访问 (机械磁盘的真正问题是获取第一个数据的时间)。

Hash Join

hash join 相比 nested loop jon 要复杂，但在多数的使用场景下代价更小

HUBBLE

hash join 的概念是:

- 1) 从 inner relation 中获取所有元素
- 2) 建立内存中的哈希表
- 3) 将 outer relation 的所有元素逐一获取
- 4) 计算每个元素的哈希值 (使用哈希表的哈希函数)，找到 inner relation 的关联 bucket
- 5) 查找 bucket 中的元素与外部表中的元素是否匹配

在时间复杂度方面，我需要做一些假设来简化问题:

- inner relation 被分成 X 个桶
- 哈希函数几乎均匀分布哈希值的两个关系。换句话说，桶的大小是相等的。
- outer relation 的一个元素与桶内所有元素的匹配的代价，与桶内元素的数量有关。

时间复杂度为 (M/X) N + $cost\_to\_create\_hash\_table(M)$ + $cost_of_hash_function$N

如果哈希函数创建了足够的小桶，那么时间复杂度为 O(M+N)

这是另一个版本的散列连接，它对内存友好，但对磁盘 I/O 不友好。流程:

- 1) 计算 inner relations 和 outer relations 的哈希表
- 2) 然后把它们存放在磁盘上
- 3) 然后逐个 bucket 比较两个关系 (一个在内存中加载，另一个在内存中逐行读取)

Merge join

merge join 是唯一产生排序结果的连接方式。

注意: 在这个简化的 merge join 中，没有内部表或外部表; 他们都扮演着同样的角色。但是真正的实现会有所不同，例如在处理重复记录时。

合并连接可以分为两个步骤:

1. (可选) 排序连接操作: 两个输入都按连接键排序。
2. Merge join 操作: 将排序后的输入合并在一起。

- Sort

我们已经讨论过归并排序，在这种情况下，归并排序是一种好的算法 (但如果内存并不足够富裕时，就不是最好的算法)。

但有时数据集已经排序，例如:

- 如果表是原生已经排序的，例如连接条件下的索引组织表
- 如果关联的连接条件上是索引
- 如果此关联应用于查询过程中已排序的中间结果
- Merge join

这部分与我们看到的归并排序的归并操作非常相似。但是这一次，我们不是从两个关系中选择每个元素，而是从两个相等的关系中选择元素。这个想法是这样的:

- 1) 比较两个关系中的当前元素 (当前 = 第一次)
- 2) 如果它们相等，那么将两个元素都放入结果中，然后转到两个关系的下一个元素
- 3) 如果不是，则转到下一个元素，查找与排序最低元素的关联 (因为下一个元素可能匹配)
- 4) 重复 1、2、3，直到找到关系中的最后一个元素。

HUBBLE

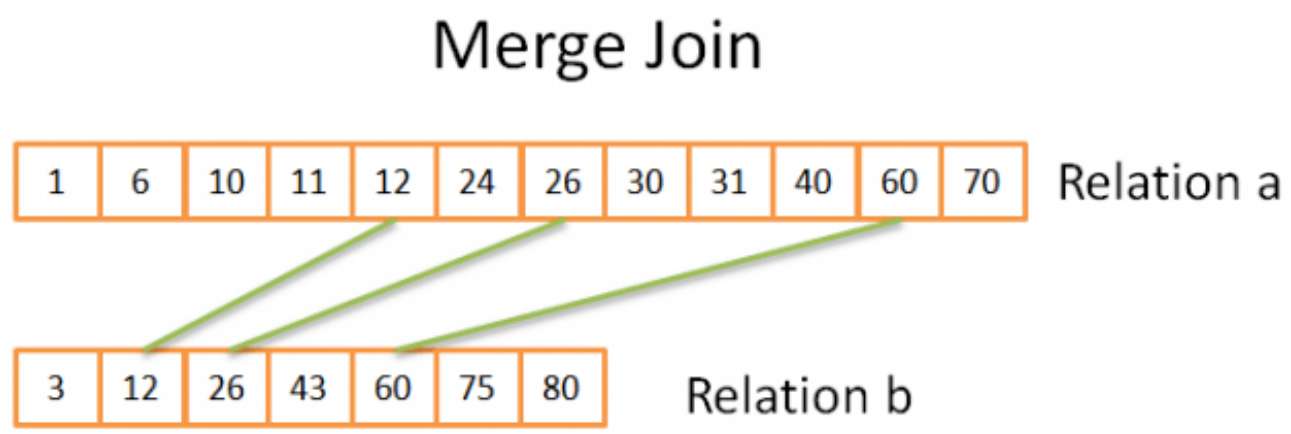


图 78: merge_join

这是可行的，因为两个关系都是排序的，因此您不需要在这些关系中“返回”。

这个算法是一个简化的版本，因为它不处理相同的数据在两个数组中出现多次的情况（也就是多个匹配上的情况）。真正的版本在这种情况下更复杂；这就是为什么我选择了一个简化的版本。

如果两个关联都已排序，则**时间复杂度为 O(N+M)**

如果两个关系都需要排序，那么时间复杂度就是对两个关联排序的代价:**O(N*Log(N) + M*Log(M))**

对于计算机科学爱好者来说，这里有一个可能的算法，可以处理多个匹配（注意：我不是 100% 肯定我的算法）：

```
mergeJoin(relation a, relation b)
  relation output
  integer a_key:=0;
  integer b_key:=0;

  while(a[a_key]!=null or b[b_key]!=null)
    if (a[a_key] < b[b_key])
      a_key++;
    else if (a[a_key] > b[b_key])
      b_key++;
    else//Join predicate satisfied
    //i.e. a[a_key] == b[b_key]

    //count the number of duplicates in relation a
    integer nb_dup_in_a = 1:
    while(a[a_key]==a[a_key+nb_dup_in_a])
      nb_dup_in_a++;
      //count the number of duplicates in relation b
      integer dup_in_b = 1:
      while (b[b_key]==b[b_key+nb_dup_in_b])
        nb_dup_in_b++;

        //write the duplicates in output
        for (int i = 0; i< nb_dup_in_a ; i++)
          for (int j = 0; i< nb_dup_in_b ; i++)
```

HUBBLE

```
            write_result_in_output(a[a_key+i],b[b_key+j])
        a_key=a_key + nb_dup_in_a- 1;
        b_key=b_key + nb_dup_in_b-1;

        end if
      end while
```

哪个更好？

如果有最好的连接类型，就不会有这些多种类型。这个问题很难，因为有很多因素在起作用，比如:

- 可用内存: 没有足够的内存，您可以与强大的 hash join 告别了（至少是完整的内存 hash join)
- 两个数据集的大小。例如，如果您有一个包含非常小的表的大表，那么 nested loop join 将比 hash join 更快，因为 hash join 创建散列的开销很大。如果您有两个非常大的表，nested loop join 将非常占用 CPU 开销。
- 是否有索引。对于两个 b + 树索引，可以明智的选择 merge join
- 如果结果需要排序: 即使你使用无序的数据集, 您可能想要使用一个代价高昂的 merge join, 因为最后的结果排序, 你可以将结果与另一个 merge join 关联
- 如果关联已经排序: 在这种情况下，merge join 是最好的选择方案
- 你正在做的连接的类型: 它是一个等价连接 (即 tableA.col1 = tableB.col2) ？它是一个内连接，一个外连接，一个笛卡尔积还是一个自连接? 一些连接在某些情况下不能工作。
- 数据分布。如果连接条件中的数据是歪斜的 (例如，您按照人们的姓连接他们，但是许多人的姓都是一样的)，那么使用 hash join 将是一场灾难，因为散列函数将创建不均匀分布的 bucket。
- 如果您想要多线程/进程执行联接

想要获取更多的信息，你可以读DB2，ORACLE or SQL Server的文档.

最简例子

我们已经看到了三种类型的连接操作。

现在，假设我们需要连接 5 个表来完整地查看一个人的信息。一个人可以有:

- 多个手机
- 多个邮件
- 多个地址
- 多个银行账号

简而言之，我们需要对下面这个查询，有一个快速的响应

```
SELECT *  from PERSON, MOBILES, MAILS,ADRESSES, BANK_ACCOUNTS WHERE PERSON.
    ↪ PERSON_ID = MOBILES.PERSON_ID AND PERSON.PERSON_ID = MAILS.PERSON_ID AND
    ↪ PERSON.PERSON_ID = ADRESSES.PERSON_ID AND PERSON.PERSON_ID = BANK_ACCOUNTS
    ↪ .PERSON_ID
```

作为一个查询优化器，我必须找到处理数据的最佳方法。但有两个问题:

- 我应该为每个连接使用什么类型的连接?

我有 3 种可能的连接 (Hash Join, Merge Join, Nested Join)，可以使用 0、1 或 2 个索引 (更不用说还有不同类型的索引了)。

- 我应该选择什么顺序来计算连接?

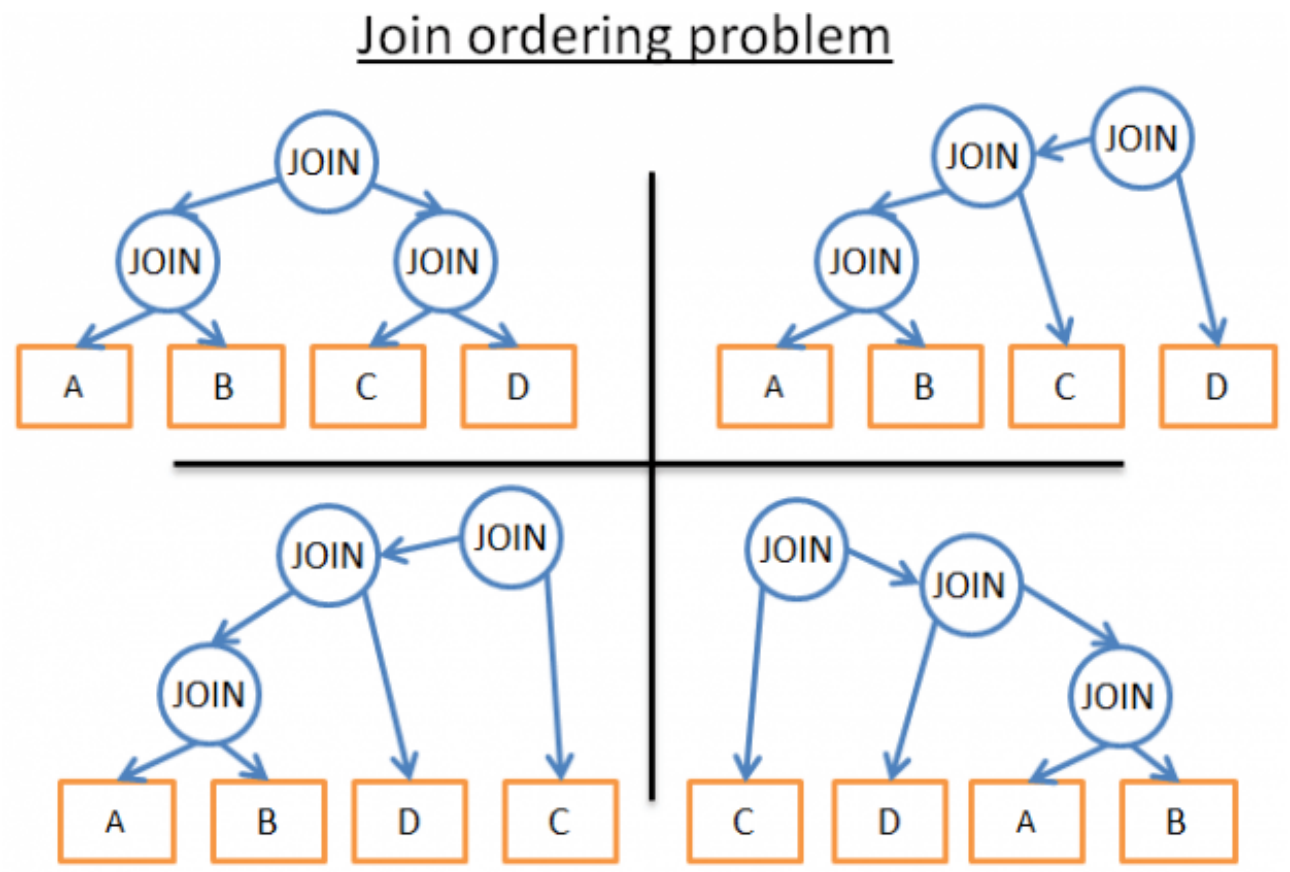


图 79: join__ordering__problem

例如，下图显示了仅针对 4 个表的 3 个连接的不同可能计划

这就是我提出的可能性:

- 1) 我使用蛮力方法

使用数据库统计，我计算每一个可能的计划的成本，我保留最好的一个。但是有很多可能性。对于给定的联接顺序，每个联接都有 3 种可能:HashJoin、MergeJoin、NestedJoin。对于给定的连接顺序有 34 种可能。连接顺序是一个在二叉树上的排列问题，并且有`(2*4)!/(4+1)!`可能的订单。对于这个非常简单的问题，我最终得到 34*(24)*!/(4+1)! 的可能性。

在非极客的术语中，它意味着 27 216 种可能的计划。如果我现在添加了合并连接使用 0、1 或 2 个 B+ 树索引的可能性，那么可能的计划数量就变成了 210 000 个。我忘记说过这个查询非常简单吗?

- 2) 我哭了，辞掉了这份工作

这很诱人，但你不会得到你的结果，我需要钱来支付账单。

- 3) 我只试了几个方案，选择成本最低的那个。

因为我不是超人，所以我无法计算每个计划的成本。相反，我可以从所有可能的计划中任意选择一个子集，计算它们的成本，然后给出这个子集的最佳计划。

- 4) 我应用智能规则来减少可能的计划的数量

有两种类型的规则:

我可以使用“逻辑”规则来排除无用的可能性，但它们不会过滤很多可能的计划。例如:“nested loop join 的内部关联必须是最小的数据集”

我接受找不到最佳解决方案的事实，并采用更激进的规则来大幅减少可能性的数量。例如，“如果关联很小，则使用 nested loop join，而决不使用 merge join 或 hash join”

在这个简单的例子中,我得到了许多可能性。但是一个真正的查询可以有其他的关系操作符,比如 OUTER JOIN、CROSS JOIN、GROUP BY、ORDER BY、PROJECTION、UNION、INTERSECT、DISTINCT ……，这意味着有更多的可能性。

那么，数据库是如何做的?

7.7.4.4.3 动态编程，贪婪算法和启发式

关系数据库尝试了我刚才提到的多种方法。优化器的真正工作是在有限的时间内找到一个好的解决方案。

大多数时候，优化器不是找到最好的解决方案，而是找到一个“好的”解决方案。

对于小的查询，可以使用蛮力方法。但是有一种方法可以避免不必要的计算，这样即使中等的查询也可以使用蛮力方法。这叫做动态规划。

Dynamic Programming

这两个词背后的意思是很多执行计划是非常相似的。如果你看看下面的计划:

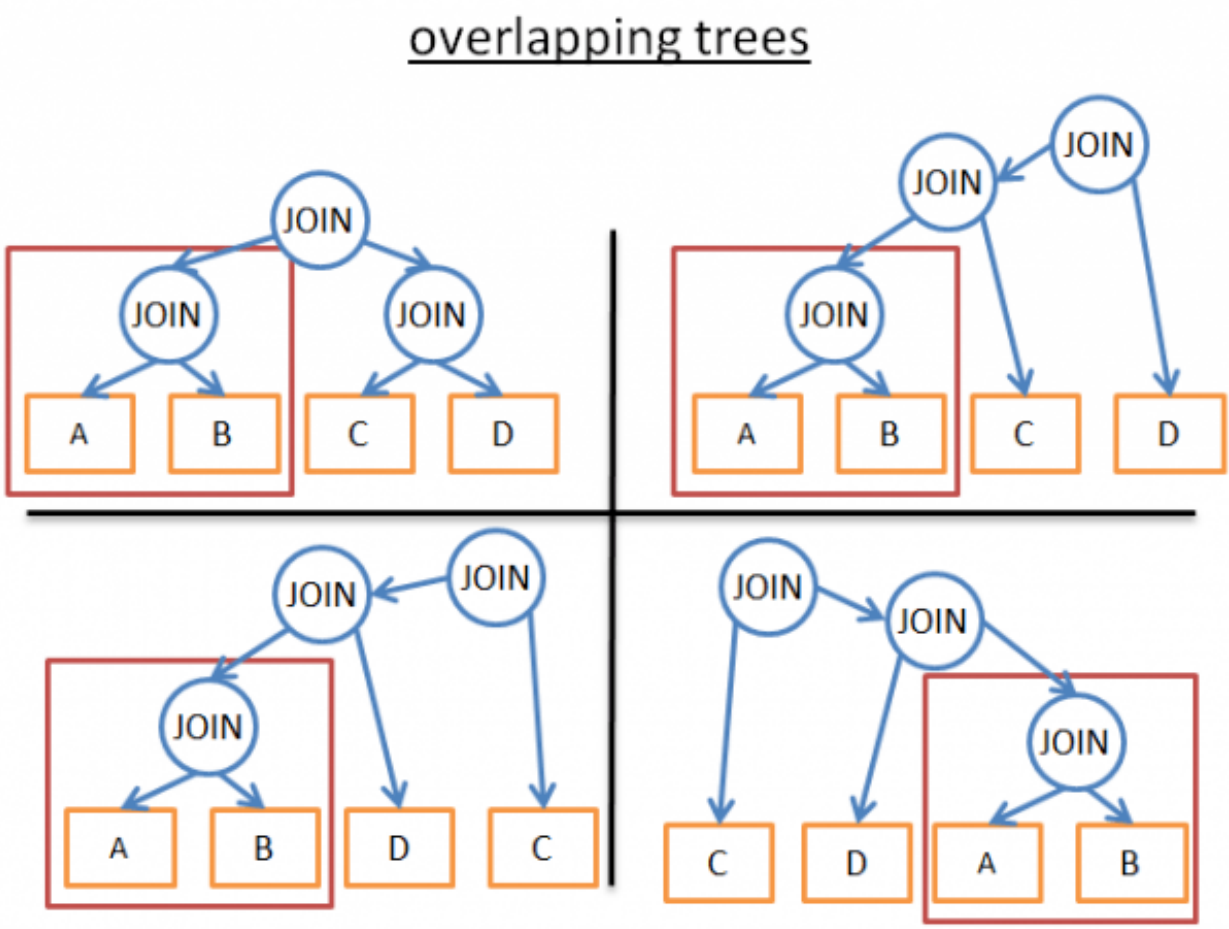


图 80: overlapping trees

它们共享相同的 (A JOIN B) 子树。因此，不必在每个计划中计算这个子树的代价，我们可以计算一次，保存计算出的代价，并在再次看到这个子树时重用它。更正式地说，我们面临的是一个重叠的问题。为了避免部分结果的额外计算，我们使用记忆法。

使用这种技术，而不是 (2*N)!/(N+1)! 时间复杂度，而只有 3N。在我们前面的 4 个连接示例中，它意味着从 336 降低到 81。如果您使用一个包含 8 个连接的更大的查询 (并不大)，这意味着从 57 657 600 降低到 6561。

对于计算机科学极客们，这是我在我已经给你们上过的正式课程中找到的一个算法。我不会解释这个算法，所以只有当你已经知道动态编程或如果你擅长算法时可以阅读这篇文章。

```
procedure findbestplan(S)
if (bestplan[S].cost infinite)
   return bestplan[S]
   // else bestplan[S] has not been computed earlier, compute it now
   if (S contains only 1 relation)
     set bestplan[S].plan and bestplan[S].cost based on the best way
     of accessing S /* Using selections on S and indices on S */
   else  for each non-empty subset S1 of S such that S1 != S
   P1= findbestplan(S1)
   P2= findbestplan(S - S1)
   A = best algorithm for joining results of P1 and P2
   cost = P1.cost + P2.cost + cost of A
   if cost < bestplan[S].cost
     bestplan[S].cost = cost
     bestplan[S].plan =  “execute P1.plan; execute P2.plan;
          join results of P1 and P2 using A”
return bestplan[S]
```

对于更大的查询，你仍然可以使用动态编程方法，但是要使用额外的规则 (或启发式) 来消除可能性

- 如果我们只分析特定类型的计划 (例如: 左深树)，我们最终得到的是 n*2n，而不是 3n

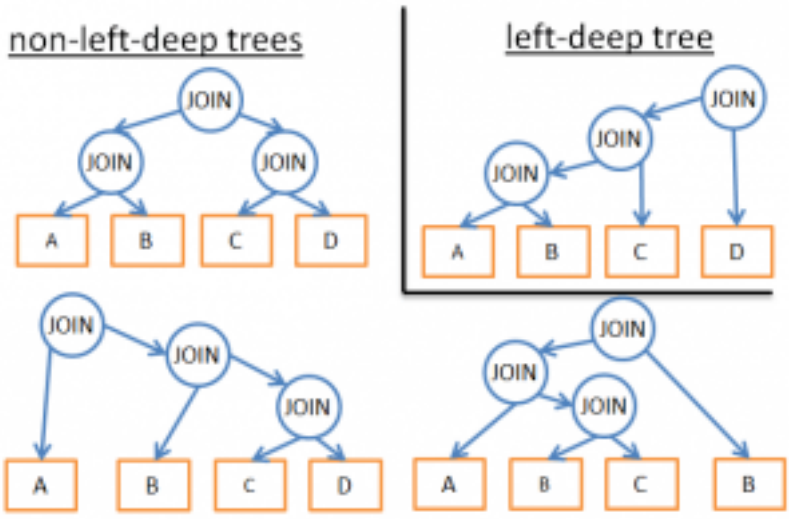


图 81: left_deep_trees

- 如果我们添加逻辑规则来避免某些模式的计划 (比如 “如果一个表是给定谓词的索引，那么不要尝试在表上进行合并连接，而只在索引上尝试”)，那么它将减少可能性的数量，而不会影响获得尽可能好的解决方案。
- 如果我们在流程上添加规则 (比如 “在所有其他关系操作之前执行连接操作”)，它也会减少很多可能性。

贪婪算法

但是对于一个非常大的查询或者有一个非常快的答案 (但不是一个非常快的查询)，则使用另一种类型的算法，即贪心算法。

其思想是遵循规则 (或**启发式**) 以增量方式构建查询计划。有了这个规则，贪婪算法就能一步一步地找到问题的最佳解决方案。该算法使用一个连接启动查询计划。然后，在每个步骤中，算法使用相同的规则向查询计划添加新的连接。

让我们举一个简单的例子。假设我们有一个查询，它包含 5 个表 (a、B、C、D 和 E) 上的 4 个连接。让我们使用“使用成本最低的连接”规则

- 我们任意从 5 个表中的一个开始 (我们选 A)
- 我们计算每个与 A 连接的成本 (A 是内部或外部的关系)。
- 我们发现 A 连接 B 的成本最低。
- 然后我们计算每个与 A JOIN B 子集的连接代价 (A join B 成为 inner relation 或 outer relation)。
- 我们发现 (A jion B) 连接 C 的代价最高。
- 然后我们用 (A jion B) 连接 C 的结果计算每个连接的成本…
- ....
- 最后，我们找到了计划 ((A JOIN B) JOIN C) JOIN D) JOIN E)

因为我们任意地从 A 开始，我们可以对 B 应用相同的算法，然后是 C，然后是 D，然后是 e，然后我们保持这个计划的最低成本。

顺便说一下，这个算法有一个名字: 它叫做最近邻算法。

我不会进入细节, 但有一个很好的建模easily be solved和一种`N * log (N)`的排序用于解决这个问题。该算法的代价是 O(N*log(N)) vs O(3N) 对于完整的动态规划版本。如果您有一个包含 20 个连接的大查询，这意味着 26 对 3 486 784 401，一个很大的区别!

这个算法的问题是，我们假设，找到两个表最佳的连接方式，并在找寻最佳的连接方式的过程中，保留两表连接的前提下，加入新表连接，从而找到多表的最佳连接方式。但是:

- 即使 A 连接 B 给出了 A、B 和 C 之间的最低代价
- (A JOIN C) JOIN B 可能比 (A JOIN B) JOIN C 的结果更好。

为了改善结果，可以使用不同的规则运行多个贪婪算法，并保持最佳计划。

其他算法

寻找最佳方案是许多计算机科学研究人员的一个活跃的研究课题。他们常常试图为更精确的问题/模式找到更好的解决方案。例如,

- 如果查询是星型连接 (它是某种类型的多连接查询)，一些数据库将使用特定的算法。
- 如果查询是并行查询，一些数据库将使用特定的算法
- ...

还有其他算法的研究，用来代替动态规划的大查询。贪心算法属于一个较大的家族，称为启发式算法。贪婪算法遵循一个规则 (或启发式)，保持它在前一步找到的解决方案，并“附用”它来找到当前步骤的解决方案。有些算法遵循一个规则，并以一种循序渐进的方式应用它，但并不总是保持在前一步中找到的最佳解决方案。它们被称为启发式算法。

例如，遗传算法遵循一个规则，但最后一步的最佳解决方案往往不保持:

- 一个解决方案表示一个可能的完整查询计划
- 不是一个解决方案 (即计划)，而是 P 个解决方案 (即计划) 保持在每个步骤。
- 0) P 查询计划是随机创建的
- 1) 只保留代价最低的计划

HUBBLE

- 2) 这些最好的计划混合在一起就能产生 P 个新计划
- 3) 有些新计划是随机修改的
- 4) 步骤 1、2、3 重复 T 次
- 5) 然后从最后一个循环的 P 计划中保留最佳计划。

你做的循环越多，你的计划就会越好。

这是魔法吗? 不，这是自然法则: 适者生存!

顺便说一句，遗传算法是在 PostgreSQL 中实现的，但我无法找到它们是否被默认使用。

数据库中还有其他启发式算法，比如 Simulated Annealing,Iterative Improvement, 两阶段优化……但我不知道它们目前是用于企业数据库，还是只用于研究数据库。

有关更多信息，可以阅读下面的研究文章，其中介绍了更多可能的算法:数据库查询优化中连接排序问题算法综述

7.7.4.4.4 Real optimizers

[你可以跳到下一部分，我要说的并不重要]

但是，所有这些都是理论性的。因为我是一名开发人员，而不是研究员，所以我喜欢具体的例子。

让我们看看SQLite 优化器是如何工作的。这是一个轻量级的数据库，所以它使用了一个简单的优化基于一个贪婪算法与额外的规则来限制可能性的数量:

- SQLite 选择从不在交叉连接操作符中，重新排序表
- 连接以 nested joins 的方式实现
- 外部连接总是按照它们发生的顺序计算
- -…
- -在 3.8.0 版本之前，SQLite 在搜索最佳查询计划时使用 “最近邻贪婪算法

  等等，我们已经看过这个算法了! 真是巧了!
- 自 3.8.0 版本 (2015 年发布) 以来，SQLite 在搜索最佳查询计划时使用了N 近邻贪婪算法

让我们看看另一个优化器是如何工作的。IBM DB2 与所有的企业数据库一样，但我将重点介绍这个数据库，因为它是我在转换到大数据之前真正使用过的最后一个数据库。

如果我们查看官方文档，我们会了解到 DB2 优化器允许您使用 7 种不同级别的优化:

- 使用贪婪算法的连接
  - 0 最小优化，使用索引扫描和嵌套循环连接，并且避免一些查询重写
  - 1 低优化
  - 2 完全优化
- 使用动态编程的连接
  - 3 中度优化和粗略近似
  - 5 充分优化，使用启发式的所有技术
  - 7 完全优化类似于 5，没有启发式
  - 9 最大优化不遗余力/不惜代价考虑所有可能的连接次序，包括笛卡尔产品

  我们可以看到，DB2 使用贪婪算法和动态编程。当然，由于查询优化器是数据库的主要驱动力，所以它们不会共享它们所使用的启发法。

仅供参考，默认级别为 5。默认情况下，优化器使用以下特征:

HUBBLE

- 使用所有可用的统计数据，包括频率值和分位数统计数据。
- 应用所有查询重写规则 (包括物化查询表路由)，但计算密集型规则只适用于非常罕见的情况。
- 动态编程连接枚举，使用:
  - 限制只用复合内部关系
  - 限制使用对涉及查找表的星型模式笛卡尔积
- 广泛的访问方法被考虑，包括列表预取 (注意: 将看到什么是意思)，索引的结束 (注意: 一个特殊的操作与索引)，物化查询表路由。

默认情况下，DB2 对连接顺序使用受启发法限制的动态编程。

其他条件 (GROUP BY、DISTINCT…) 由简单的规则处理。

7.7.4.4.5 Query Plan Cache

由于创建计划需要时间，大多数数据库将计划存储到查询计划缓存中，以避免对相同查询计划进行不必要的重新计算。这是一个很大的主题，因为数据库需要知道何时更新过时的计划。其主旨是设置一个阈值，如果一个表的统计数据超过了这个阈值，那么涉及该表的查询计划将从缓存中清除。

7.7.4.5 查询执行器

在这个阶段，我们有一个已优化过的执行计划。这个计划被编译成一个可执行的代码。然后，如果有足够的资源 (内存、CPU)，则由查询执行程序执行。计划中的操作符 (JOIN、SORT BY…) 可以按顺序或并行方式执行; 这取决于执行器。为了获取和写入数据，查询执行器与数据管理器进行交互，这是本文的下一部分。

7.7.5 数据管理器

data_manager

在此步骤中，查询管理器正在执行查询并需要来自表和索引的数据。它要求数据管理器获取数据，但是有两个问题:

- 关系数据库使用事务模型。因此，您不能在任何时候获取任何数据，因为其他人可能同时在使用/修改数据。
- 数据检索是数据库中最慢的操作，因此数据管理器需要足够智能来获取和保存内存缓冲区中的数据。

在本部分中，我们将看到关系数据库如何处理这两个问题。我不会讨论数据管理器获取数据的方式，因为它不是最重要的 (这篇文章够长了!)

7.7.5.1 缓存管理器

如前所述，数据库的主要瓶颈是磁盘 I/O。为了提高性能，现代数据库使用缓存管理器。

cache_manager

查询执行程序不是直接从文件系统获取数据，而是将数据请求到缓存管理器。缓存管理器有一个名为缓冲池的内存缓存。从内存中获取数据极大地提高了数据库的速度。很难给出一个数量级，因为这取决于你需要做的操作:

- 顺序访问 (例如: 全扫描)vs 随机访问 (例如: 按行 id 访问)，
- 读和写

  数据库使用的磁盘类型:

  - 72k /10k/15k 转/分钟硬盘
  - SSD

– RAID 1/5 /···

但是我要说内存比磁盘快 100 到 100k 倍。

但是，这会导致另一个问题 (与数据库一样·····)。缓存管理器需要在查询执行器使用数据之前获取数据; 否则，查询管理器必须等待来自慢速磁盘的数据。

7.7.5.2 预读取

这个问题称为预读取。查询执行程序知道它需要的数据，因为它知道查询的完整流程，并且知道磁盘上的数据和统计信息。这个主旨是这样的:

- 当查询执行器处理它的第一批数据时
- 它要求缓存管理器预加载第二批数据
- 当它开始处理第批组数据时
- 它要求缓存管理器预加载第三批，并通知缓存管理器的第一批数据可以从缓存中清除。
- ···

缓存管理器将所有这些数据存储在其缓冲池中。为了知道是否仍然需要数据，缓存管理器添加了关于缓存数据的额外信息 (称为 **latch**)。

有时，查询执行器不知道需要什么数据，有些数据库不提供此功能。相反，它们使用推测性的预取 (例如: 如果查询执行器请求数据 1、3、5，它可能在会提前请求数据 7、9、11) 或顺序预取 (在这种情况下，缓存管理器只是在请求的数据之后从磁盘加载下一个连续的数据)。

为了监视预取的工作情况，现代数据库提供了一个名为缓存命中率的监控。命中率显示了在不需要磁盘访问的情况下，在缓冲区缓存中找到请求数据的频率。

注意: 较差的缓存命中率并不总是意味着缓存不能正常工作。要了解更多信息，可以阅读 Oracle documentation。

但是，缓冲区是有限的内存。因此，它需要删除一些数据才能加载新的数据。加载和清除缓存需要付出磁盘和网络 I/O 方面的代价。如果您有一个经常执行的查询，那么总是加载然后清除此查询使用的数据是低效的。为了处理这个问题，现代数据库使用缓冲区替换策略。

5.3. 缓存替换策略

- LRU

LRU 代表 Least Recently Used。此算法背后的思想是将最近使用过的数据保存在缓存中，因此更有可能再次使用。

这是一个虚拟的例子:

为了便于理解，我假设缓冲区中的数据没有被锁存器锁定 (因此可以删除)。在这个简单的例子中，缓冲区可以存储 3 个元素:

- 1: 缓存管理器使用数据 1 并将数据放入空缓冲区
- 2: CM 使用数据 4 并将数据放入半加载的缓冲区
- 3: CM 使用数据 3 并将数据放入半加载的缓冲区
- 4: CM 使用数据 9。缓冲区已满，因此数据 1 被删除，因为它是最后的近期使用的数据。数据 9 被添加到缓冲区中
- 5: CM 使用数据 4。数据 4 已经在缓冲区中，因此它再次成为第一个最近使用的数据。
- 6: CM 使用数据 1。缓冲区已满，因此数据接 9 被删除，因为它是最后的近期使用的数据。将数据 1 添加到缓冲区中
- ···

HUBBLE

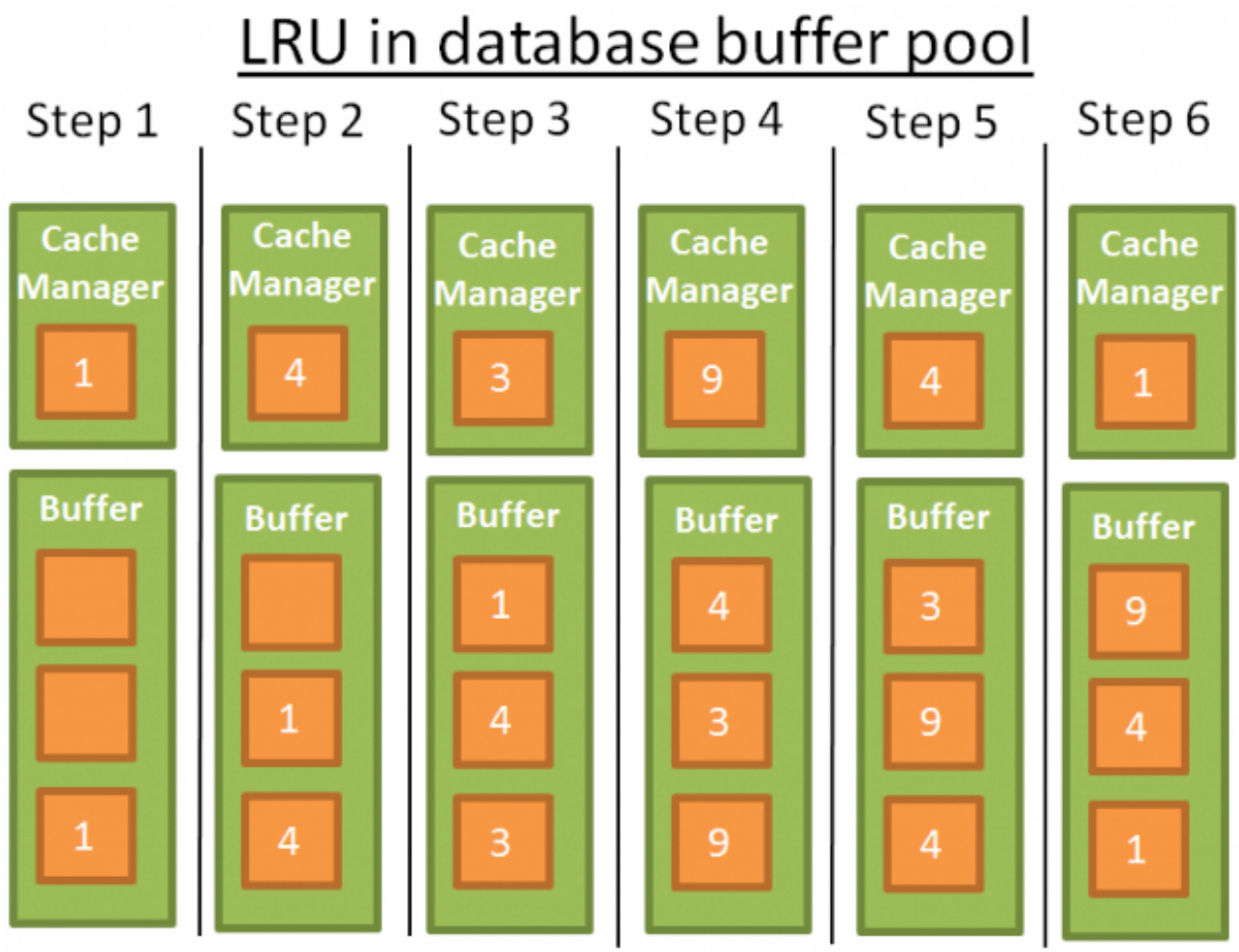

LRU in database buffer pool
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Cache Manager
1
Cache Manager
4
Cache Manager
3
Cache Manager
9
Cache Manager
4
Cache Manager
1
Buffer
1
Buffer
1
4
Buffer
1
4
3
Buffer
4
3
9
Buffer
3
9
4
Buffer
9
4
1

图 82: LRU

该算法运行良好，但存在一定的局限性。如果在一个大的表上有一个完整的扫描呢？换句话说，当表/索引的大小超过缓冲区的大小时，会发生什么情况？使用此算法将删除缓存中的所有以前的值，而来自完整扫描的数据可能只使用一次。

改进

为了防止这种情况发生，一些数据库添加了特定的规则。例如，根据Oracle 文档:

> 对于非常大的表，数据库通常使用直接路径读取，直接加载块，以避免填充缓冲区缓存。对于中等大小的表，数据库可以使用直接读取或缓存读取。如果决定使用缓存读取，那么数据库会将这些块放在 LRU 列表的末尾，以防止扫描有效地清除缓存。

还有其他的可能性，比如使用 LRU 的高级版本 LRU- k。例如，SQL Server 使用 LRU-K 表示 K =2。

这个算法背后的思想是考虑到更多的历史。对于简单的 LRU，算法只考虑最后一次使用数据的时间。LRU-K:

- 它考虑了 K 的数据最后一次使用。
- 一个权重放在的次数的数据被使用
- 如果一堆新数据被加载到缓存中，旧的但经常使用的数据不会被删除 (因为它们的权重更高)。
- 但是如果旧数据不再使用，算法就不能将它们保存在缓存中。
- 因此，权重减少随着时间的推移，如果数据没有使用。

权重的计算是昂贵的，这就是为什么 SQL Server 只使用 K=2。对于可接受的开销，这个值执行得很好。

要更深入地了解 LRU-K，可以阅读原始的研究论文 (1993):数据库磁盘缓冲的 lru-k 页面替换算法

其他算法

当然还有其他的算法来管理缓存:

- 2Q(类 LRU-K 算法)
- 时钟 (类似于 LRU-K 的算法)
- MRU(最近使用，与 LRU 使用相同的逻辑，但有另一个规则)
- LRFU(最近使用频率最高)
- …

一些数据库允许使用默认算法之外的另一种算法。

7.7.5.3 写缓存

我只讨论了在使用数据之前加载数据的读缓冲区。但是在数据库中，也有写缓冲区，它存储数据并将它们按组刷新到磁盘上，而不是逐个写入数据，那样会产生许多单独的磁盘访问。

请记住，缓冲区存储的是页 (最小的数据单元)，而不是行 (这是一种逻辑/人工的数据查看方式)。如果缓冲池中的页已被修改且未写入磁盘，则该页为脏。有多种算法可以决定在磁盘上写入脏页面的最佳时间，但是它与事务的概念高度相关，这是本文的下一部分。

7.7.6 事务管理器

最后，这一部分是关于事务管理器的。我们将看到这个过程如何确保每个查询在自己的事务中执行。但是在此之前，我们需要了解 ACID 事务的概念。

7.7.6.1 ACID

ACID 事务是一个工作单元，它确保 4 件事情:

HUBBLE

- 原子性: 事务是“全部或没有”，即使它持续 10 个小时。如果事务崩溃，则状态回到事务之前 (事务被回滚)。
- 隔离: 如果两个事务 A 和 B 同时运行，无论 A 是否在事务 B 之前/之后/期间完成，事务 A 和 B 的结果必须相同。
- 持久性: 一旦事务**提交** (即成功结束)，无论发生什么 (崩溃或错误)，数据都将保留在数据库中。
- 一致性: 只有有效的数据 (在关系约束和功能约束方面) 被写入数据库。一致性与原子性和隔离性有关。

在同一事务期间，可以运行多个 SQL 查询来读取、创建、更新和删除数据。当两个事务使用相同的数据时，混乱就开始了。经典的例子是账户 a 向账户 b 的转账。假设你有两笔交易:

- 交易 1，从 A 账户取出 100 美元给 B 账户
- 交易 2，从 A 账户取出 50 美元给 B 账户

如果我们回到 ACID 属性:

- 原子性确保无论在 T1 期间发生什么 (服务器崩溃、网络故障……)，您都不会出现从 a 中取出 100 美元而没有给 B 的情况 (这种情况是不一致的状态)。
- 隔离性确保如果 T1 和 T2 发生在同一时间, 最后将 150 B 得到 150 美元, 例如, 一个 150 美元和 B 给 50 美元因为 T2 的行为部分抹去 T1(这种情况下也是一个不一致的状态)。
- 持久性确保 T1 不会消失在空气中，如果数据库崩溃后，只是 T1 被提交。
- 一致性确保在系统中没有创建或销毁任何金钱。

[如果你愿意，你可以跳到下一部分，我要说的内容对这篇文章的其余部分并不重要]

许多现代数据库没有将纯隔离作为默认行为，因为它带来了巨大的性能开销。SQL 规范定义了 4 个隔离级别:

- Serializable (SQLite 中的默认行为): 最高级别的隔离。同时发生的两个事务是 100% 隔离的。每一笔交易都有自己的“世界”。
- Repeatable read (MySQL 中的默认行为): 除了一种情况外，每个事务都有自己的“世界”。如果一个事务成功结束并添加了新数据，这些数据将在另一个事务中可见，并且仍然在运行事务。但是，如果修改了数据并成功结束，那么修改将在仍然运行的事务中不可见。因此，事务间隔离的中断只与新数据有关，而与现有数据无关。

例如，如果事务 a 执行 “SELECT count(1) from TABLE_X”，然后事务 B 在 TABLE_X 中添加并提交一个新数据，如果事务 a 再次执行 count(1)，则该值将不相同。

这称为**幻读**。

- Read committed (Oracle, PostgreSQL 和 SQL Server 的默认行为): 这是一个可重复的读取一个新的隔离。如果一个事务 a 读取一个数据 D，然后这个数据被事务 B 修改 (或删除) 并提交，如果 a 再次读取数据 D，它将看到 B 对数据所做的修改 (或删除)。

这称为**不可重复读**。

- 读取未提交: 隔离的最低级别。这是一种读提交一种隔离的新突破。如果一个事务 a 读取一个数据 D，然后这个数据 D 被一个事务 B 修改 (未提交且仍在运行)，如果 a 再次读取数据 D，它将看到修改后的值。如果事务 B 被回滚，那么由 A 第二次读取的数据 D 没有任何意义，因为它已经被一个从未发生过的事务 B 修改过 (自从它被回滚以来)。

这叫做**脏读**。

大多数数据库都添加了自己的自定义隔离级别 (比如 PostgreSQL、Oracle 和 SQL Server 使用的快照隔离)。而且，大多数数据库并没有实现 SQL 规范的所有级别 (尤其是读未提交级别)。

用户/开发人员可以在连接开始时覆盖默认的隔离级别 (只需添加非常简单的一行代码)。

HUBBLE

7.7.6.2 并发控制

确保隔离、一致性和原子性的真正问题是对相同数据的写操作 (添加、更新和删除):

- 如果所有的事务只是读取数据，他们可以在同一时间工作，而不修改另一个事务的行为。
- 如果 (至少) 其中一个事务正在修改其他事务读取的数据，那么数据库需要找到一种方法来对其他事务隐藏这种修改。此外，它还需要确保这个修改不会被另一个没有看到修改数据的事务删除。

此问题称为并发控制。

解决这个问题最简单的方法是一个一个地运行每个事务 (即按顺序运行)。但那是完全不可扩展的，而且只有一个核心在多处理器/核心服务器上工作，效率不是很高……

解决这个问题的理想方法是，每次创建或取消一个事务时:

- 监察所有事务的所有操作
- 检查两个 (或更多) 事务的部分是否冲突，因为它们正在读取/修改相同的数据。
- 对冲突事务内部的操作进行重新排序，以减少冲突部分的大小
- 按一定顺序执行冲突部分 (非冲突事务仍同时运行)。
- 考虑到一笔交易可以取消。

更正式地说，这是一个如何调度冲突的问题。更具体地说，这是一个非常困难和 cpu 昂贵的优化问题。企业数据库无法承受为每个新事务事件寻找最佳调度而等待数小时的代价。因此，他们使用不太理想的方法，导致在冲突的事务之间浪费更多的时间。

7.7.6.3 锁管理器

为了处理这个问题，大多数数据库使用**锁**和/或**数据版本控制**。由于这是一个很大的主题，所以我将重点讨论锁部分，然后再谈一谈数据版本控制。

7.7.6.3.1 Pessimistic locking

悲观锁

锁背后的思想是:

- 如果事务需要数据
- 它锁定数据
- 如果另一个事务也需要这些数据
- 它将不得不等到第一个事务释放数据。

这称为一个**独占锁**。

但是，为只需要读取数据的事务使用排他锁是非常昂贵的，因为它会迫使只想读取相同数据的其他事务等待。这就是为什么有另一种类型的锁，共享锁。

共享锁:

- 如果一个事务只需要读取一个数据 a，
- 它“共享锁定”数据并读取数据
- 如果第二个事务也只需要读取数据 a，
- 它“共享锁定”数据并读取数据
- 如果第三个事务需要修改数据 a，
- 它“独占锁”数据，但它必须等待，直到 2 个其他事务释放他们的共享锁，以应用其独占锁的数据 A。

但是，如果一个数据是一个排他锁，那么只需要读取数据的事务将不得不等待排他锁的结束，以便在数据上放置一个共享锁。

HUBBLE

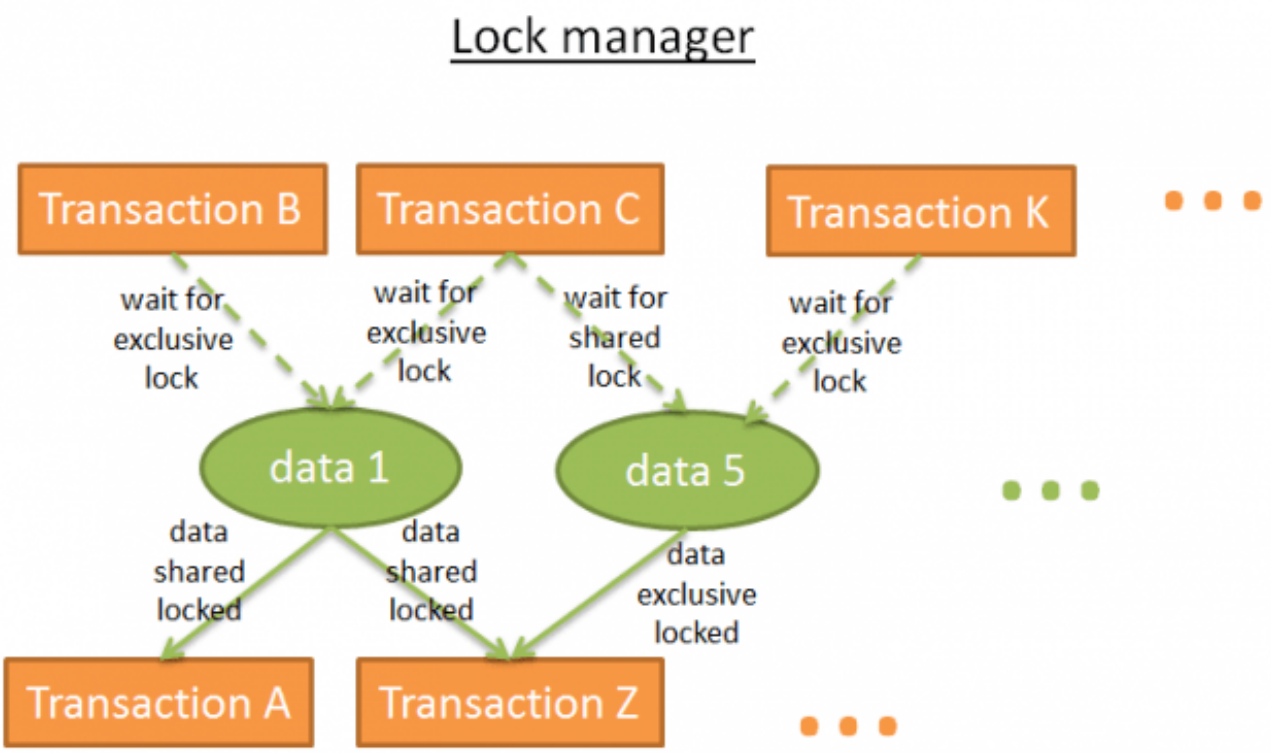


图 83: lock manager in a database

锁管理器是提供和释放锁的进程。在内部，它将锁存储在一个哈希表中 (其中的键是要锁定的数据)，并知道每个数据:

- 哪些事务正在锁定数据
- 哪些事务正在等待数据

7.7.6.3.2 Deadlock

但是使用锁会导致两个事务永远等待一个数据:

在这个图:

- 事务 A 拥有 data1 的独占锁，正在等待获取 data2
- 事务 B 拥有 data2 的独占锁，正在等待获取 data1

这称为**死锁**。

在死锁期间，锁管理器将选择取消 (回滚) 哪个事务以消除死锁。这个决定并不容易:

- 是否最好终止修改最少数据量的事务?
- 如果另一个事务的用户已经等待了更长的时间，那么是否最好终止最年轻的事务?
- 是更好的杀死事务，将需要更少的时间来完成?
- 在回滚的情况下，有多少事务会受到回滚的影响?

但是在做出这个选择之前，它需要检查是否存在死锁。

可以将哈希表视为一个图 (如前面的图所示)。如果图中有一个循环，则会出现死锁。由于检查周期的开销很大 (因为包含所有锁的图非常大)，所以通常使用一种更简单的方法: 使用 **timeout**。如果在此超时内没有给出锁，事务将进入死锁状态。

锁管理器也可以在给锁之前检查这个锁是否会导致死锁。但是完美地完成它在计算上是很昂贵的。因此，这些预先检查通常是一组基本规则。

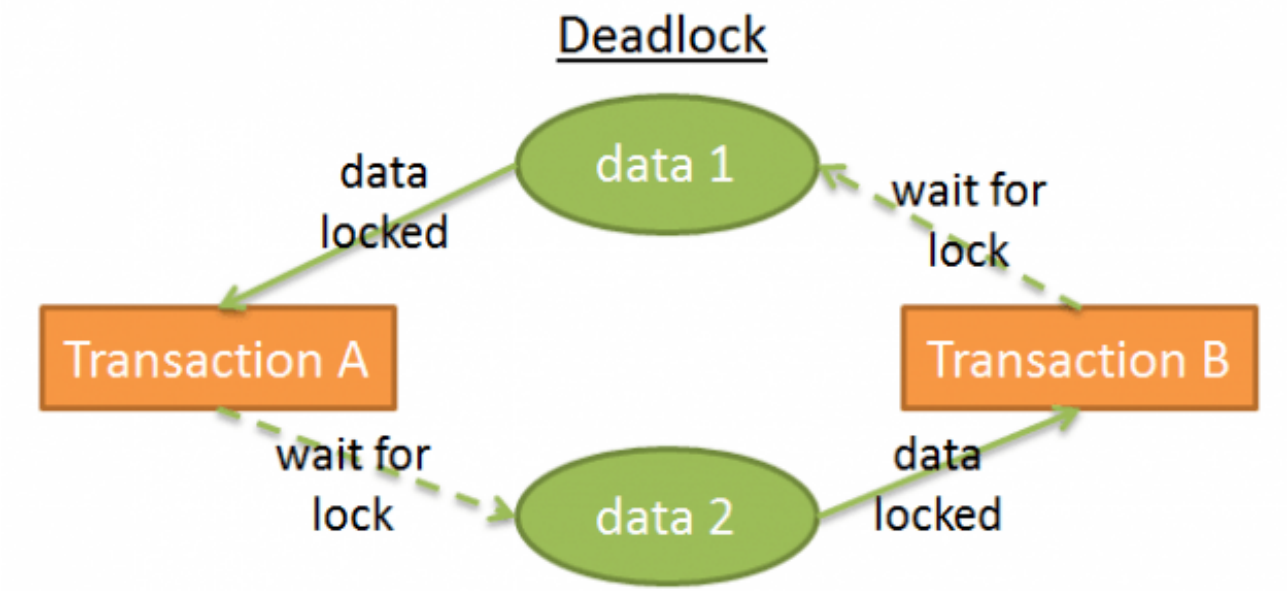


图 84: deadlock with database transactions

7.7.6.3.3 Two-phase locking

确保纯粹隔离的最简单方法是在事务开始时获取锁，并在事务结束时释放锁。这意味着一个事务在开始之前必须等待它的所有锁，并且在事务结束时释放事务持有的锁。它工作，但它**产生了大量的时间浪费**等待所有锁。

一个更快的方法是**两阶段锁协议** (由 DB2 和 SQL Server 使用)，其中一个事务分为两个阶段:

- 增长阶段，其中一个事务可以获得锁，但不能释放任何锁。
- 收缩阶段，一个事务可以释放锁 (对数据，它已经处理，不会再处理)，但不能获得新的锁。

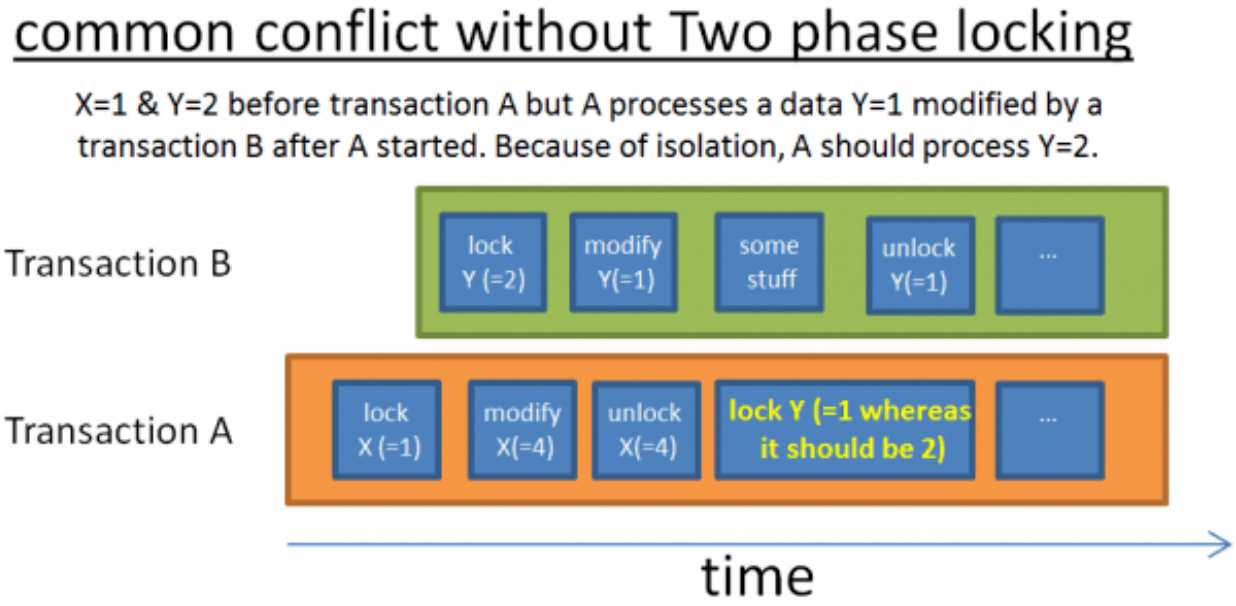


图 85: a problem avoided with two phase locking

这两个简单规则背后的思想是:

- 释放不再使用的锁，以减少其他事务等待这些锁的时间
- 防止在事务开始后修改数据，因此与事务获得的第一个数据不一致的情况。

除非修改数据并释放关联锁的事务被取消 (回滚)，否则此协议可以正常工作。您可能会遇到这样的情况: 另一个事务读取修改后的值，而这个值将被回滚。为了避免这个问题，所有的排他锁必须在事务结束时释放。

7.7.6.3.4 A few words

当然，真正的数据库使用更复杂的系统，包括更多类型的锁 (如意图锁) 和更多粒度 (行、页、分区、表、表空间上的锁)，但原理是一样的。

我只介绍了纯粹的基于锁的方法。数据版本控制是处理这个问题的另一种方法。

版本背后的思想是:

- 每个事务可以在同一时间修改相同的数据
- 每个事务都有自己的数据副本 (或版本)
- 如果两个事务修改相同的数据，只有一个修改将被接受，另一个将被拒绝，相关的事务将被回滚 (可能会重新运行)。

它提高了性能，因为:

- 读取器事务不会阻塞写入器事务
- 写事务不会阻塞读事务
- 没有来自“臃肿与慢”锁管理器的开销

除了两个事务写入相同的数据外，一切都比锁好。此外，您很快就会得到一个巨大的磁盘空间开销。

数据版本控制和锁定是两种不同的视图: 乐观锁定和悲观锁定。它们各有利弊; 这实际上取决于用例 (更多的读和更多的写)。对于一个关于数据版本控制的演讲，我推荐这个非常好的演讲来看看 PostgreSQL 是如何实现多版本并发控制的。

有些数据库，如 DB2(直到 DB2 9.7) 和 SQL Server(快照隔离除外) 只使用锁。其他如 PostgreSQL, MySQL 和 Oracle 使用了一种混合的方法，包括锁和数据版本控制。我不知道数据库只使用数据版本控制 (如果您知道一个基于纯数据版本控制的数据库，请告诉我)。

一位读者告诉我:

> Firebird 和 Interbase 使用版本控制，并没有使用行锁。版本控制对索引有一个有趣的影响: 有时一个惟一的索引包含重复项，该索引的条目可能比表的行还多，等等。

如果您在不同的隔离级别上读取该部分，那么当您增加隔离级别时，您将增加锁的数量，从而增加事务等待它们的锁所浪费的时间。这就是为什么大多数数据库在默认情况下不使用最高的隔离级别 (Serializable)。

像往常一样, 您可以检查主要数据库的文档。MySQL,PostgreSQL,Oracle

7.7.6.4 日志管理器

我们已经看到，为了提高性能，数据库将数据存储在内存缓冲区中。但是，如果在提交事务时服务器崩溃，那么在崩溃期间您将丢失仍然在内存中的数据，这会破坏事务的持久性。

您可以将所有内容写到磁盘上，但是如果服务器崩溃，您将以一半的数据写到磁盘上而告终，这会破坏事务的原子性。

必须撤消或完成事务所写的任何修改。

要解决这个问题，有两种方法:

- 影子副本/页: 每个事务创建自己的数据库副本 (或只是数据库的一部分)，并在这个副本上工作。如果出现错误，则删除该副本。如果成功，数据库立即使用文件系统技巧切换来自副本的数据，然后删除“旧的”数据。
- 事务日志: 事务日志是一个存储空间。在每次写入磁盘之前，数据库会在事务日志中写入一个信息，以便在事务崩溃/取消时，数据库知道如何删除 (或完成) 未完成的事务。

7.7.6.4.1 WAL

当在涉及许多事务的大型数据库上使用影子副本时，会产生巨大的磁盘开销。这就是为什么现代数据库使用事务日志的原因。事务日志必须存储在稳定的存储中。我不会深入讨论存储技术，但是使用 (至少)RAID 磁盘是防止磁盘故障的必要手段。

大多数数据库 (Oracle, SQL Server, DB2, PostgreSQL, MySQL and SQLite) 处理事务日志使用 **Write-Ahead Logging protocol** （WAL）。WAL 协议有 3 条规则:

- 1) 对数据库的每次修改都会产生一条日志记录，在将数据写到磁盘上之前，必须将日志记录写到事务日志中。
- 2) 日志记录必须按顺序书写; 发生在日志记录 B 之前的日志记录 a 必须写在 B 之前
- 3) 事务提交后，必须在事务成功结束之前将提交操作写入事务日志。

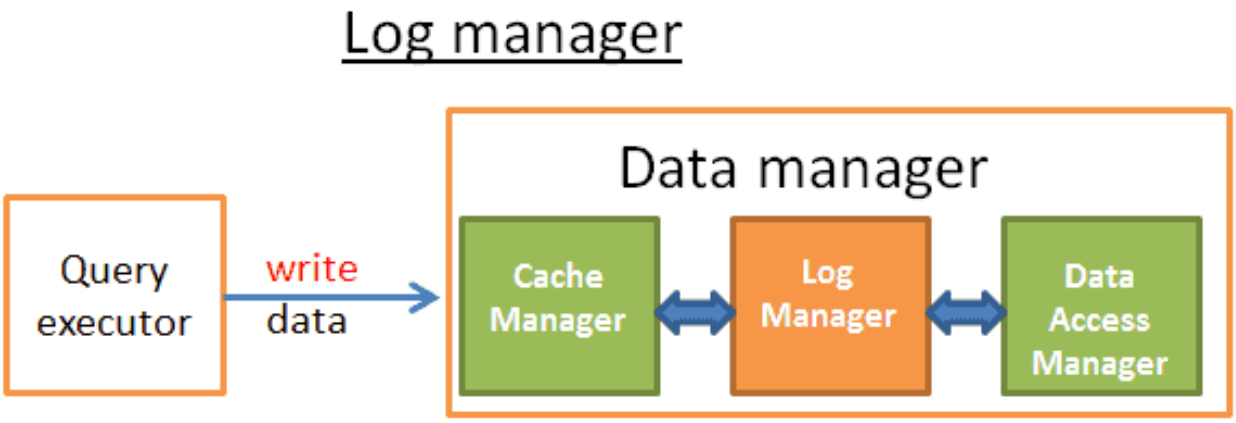


图 86: log manager in a database

这项工作由日志管理器完成。查看它的一个简单方法是，在缓存管理器和数据访问管理器 (将数据写到磁盘上) 之间，日志管理器在将事务日志写到磁盘上之前，将每个更新/删除/创建/提交/回滚写到事务日志上。这很容易，是吧?

其实并不然，！在经历了所有这些之后，您应该知道与数据库相关的一切都受到“数据库效率”的诅咒。更严重的问题是，如何在保持良好性能的同时编写日志。如果事务日志上的写操作太慢，则会降低所有操作的速度。

7.7.6.4.2 ARIES

1992 年，IBM 的研究人员“发明”了一种增强版的 WAL，称为 ARIES。大多数现代数据库或多或少都使用 ARIES。逻辑可能不一样，但背后的概念是无处不在的。我把引用放在了 invented(麻省理工学院的课程) 上，因为根据这门课程 (http://db.csail.mit.edu/6.830/lectures/lec15notes.pdf)，IBM 的研究人员“只不过编写了事务恢复的良好实践”。自从我 5 岁时，ARIES 的论文发表，我才不管那些研究人员的流言蜚语呢。我读了大量关于 ARIES 的研究论文，我发现它非常有趣! 在这一部分，我只会给你一个 ARIES 的概述，但我强烈建议阅读论文，如果你想要一个真正的知识。

ARIES 代表用于恢复和隔离使用语义的算法。

这项技术的目的有两个:

- 1) 写日志时有好的表现
- 2) 具有快速、可靠的恢复能力

数据库必须回滚事务的原因有很多:

- 因为用户取消了它
- 由于服务器或网络故障
- 因为事务破坏了数据库的完整性 (例如，您在一个列上有一个惟一的约束，而事务添加了一个副本)
- 因为死锁

有时 (例如，在网络故障的情况下)，数据库可以恢复事务。

这怎么可能? 要回答这个问题，我们需要理解存储在日志记录中的信息。

The logs

事务中的每个操作 (添加/删除/修改) 生成一个日志。本日志记录由以下几部分组成:

- LSN: 一个独特的日志序列号。这个 LSN 是按时间顺序给出的。这意味着如果操作 A 发生在操作 B 之前那么 LSN (log A) 就会小于 LSN (log B)
- TransID: 产生操作的事务的 id。
- PageID: 修改后的数据在磁盘上的位置。磁盘上的最小数据量是一个页面，因此数据的位置就是包含数据的页面的位置。
- PrevLSN: 同一个事务产生的前一个日志记录的链接。
- 撤销: 一种删除操作效果的方法

  例如，如果操作是一个更新，撤销将在更新之前存储更新元素的值/状态 (物理撤销)，或者存储返回到前一个状态的反向操作 (逻辑撤销)。
- 重做: 重做操作的一种方式
- 同样地，有两种方法可以做到这一点。要么在操作之后存储元素的值/状态，要么在操作本身中重播它。
- (仅供参考，一个 ARIES 日志有 2 个其他字段:UndoNxtLSN 和类型)。

而且，磁盘上的每个页面 (存储数据，而不是日志) 都有最后一个修改数据的操作的日志记录 (LSN) 的 id。

给出 LSN 的方式比较复杂，因为它与日志的存储方式相关联。但想法是一样的。

ARIES 只使用逻辑撤销，因为处理物理撤销实在是一团糟。

注: 据我所知，只有 PostgreSQL 没有使用撤销。它使用一个垃圾收集器守护进程来删除旧版本的数据。这与 PostgreSQL 中数据版本控制的实现有关。

为了让您更好地理解，下面是查询 “UPDATE FROM PERSON SET AGE = 18;” 生成的日志记录的可视化和简化示例。假设这个查询在事务 18 中执行。

每个日志都有一个惟一的 LSN。被链接的日志属于同一事务。日志按时间顺序链接 (链接列表的最后一个日志是最后一个操作的日志)。

Log Buffer

为了避免日志写入成为主要的瓶颈，使用了日志缓冲区。

当查询执行器要求修改时:

- 1) 缓存管理器将修改，存储在其缓冲区中。
- 2) 日志管理器将相关日志，存储在其缓冲区中。

HUBBLE

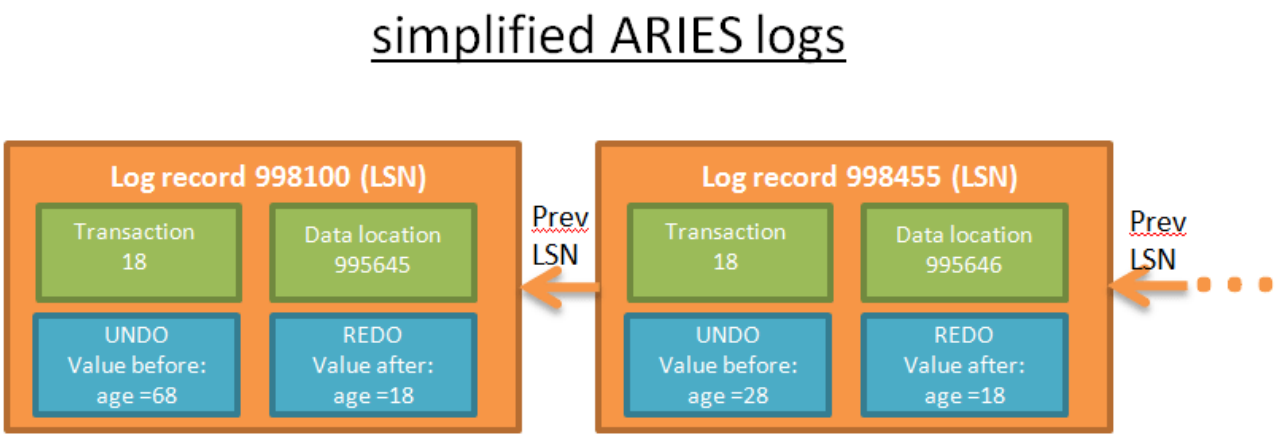


图 87: simplified logs of ARIES protocole

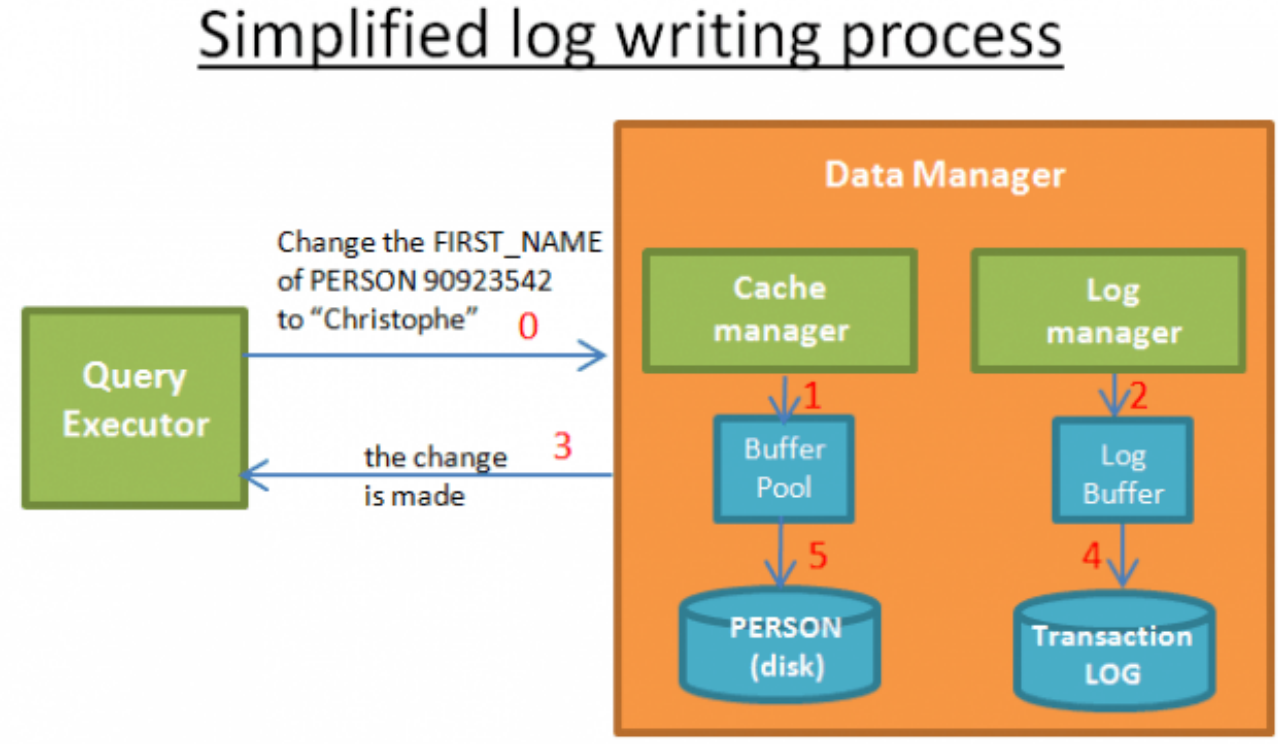


图 88: log writing process in databases

- 3) 在这一步，查询执行程序认为操作已经完成 (因此可以请求其他修改)
- 4) 然后 (稍后) 日志管理器将日志写入事务日志。何时写入日志是由算法决定的。
- 5) 然后 (稍后) 缓存管理器将修改写入磁盘。何时在磁盘上写入数据是由算法决定的。

当一个事务被提交时，它意味着对于事务中的每个操作，步骤 1、2、3、4、5 都被执行。写入事务日志是快速的，因为它只是“在事务日志的某个地方添加了一个日志”，而在磁盘上写入数据则更为复杂，因为它是以一种比读取数据更快的方式写入数据的。

STEAL and FORCE policies

出于性能原因，步骤 5 可能在提交之后执行，因为在发生崩溃的情况下，仍然可以使用重做日志恢复事务。这被称为无强制策略。

数据库可以选择一个 FORCE 策略 (即在提交之前必须执行第 5 步) 来降低恢复期间的工作负载。

另一个问题是选择是否在磁盘上一步一步地写入数据，还是缓冲区管理器需要等到提交命令才立即写入所有内容。在“偷”和“不偷”之间进行选择取决于您想要什么: 使用撤消日志进行长时间恢复的快速写入还是快速恢复?

以下是这些策略对恢复的影响摘要:

- STEAL/NO-FORCE 需要撤销和重做**:** 最高性能，但提供更复杂的日志和恢复过程 (如 ARIES)。这是大多数数据库的选择。注: 我在许多研究论文和课程中都读到过这个事实，但我在官方文件中找不到它 (明确地)。
- STEAL/ FORCE 只需要 UNDO。
- NO-STEAL/NO-FORCE 只需要 REDO。
- NO-STEAL/FORCE 不需要任何东西: 最差的表现和一个巨大的 ram 是需要的。

The recovery part

好的，我们有很好的 log，让我们使用它们!

假设新的实习生破坏了数据。重新启动数据库，恢复过程就开始了。

ARIES 从崩溃中恢复过来需要三个步骤:

- **1) 分析传递**: 恢复过程读取完整的事务日志，以重新创建崩溃期间发生的时间线。它确定要回滚哪些事务 (所有没有提交订单的事务都回滚)，以及崩溃时需要将哪些数据写到磁盘上。
- **2) 重做通过**: 此通过从分析期间确定的日志记录开始，并使用重做将数据库更新到崩溃之前的状态。

在重做阶段，重做日志按时间顺序处理 (使用 LSN)。

对于每个日志，恢复过程读取磁盘上包含要修改的数据的页面的 LSN。

如果 LSN(page_on_disk)>=LSN(log_record)，这意味着在崩溃之前已经在磁盘上写入了数据 (但是该值被在日志之后和崩溃之前发生的操作覆盖了)，所以什么也不做。

如果 LSN(page_on_disk)<LSN(log_record)，则更新磁盘上的页面。

即使对于将要回滚的事务，也会进行重做，因为它简化了恢复过程 (但我相信现代数据库不会这样做)。

- **3) 撤销传递**: 此传递回滚所有在崩溃时未完成的事务。回滚从每个事务的最后一个日志开始，并按反时间顺序处理撤消日志 (使用日志记录的 PrevLSN)。

在恢复期间，必须对事务日志发出关于恢复过程所做的操作的警告，以便磁盘上写的数据与事务日志中写的数据保持同步。解决方案可能是删除正在撤消的事务的日志记录，但这非常困难。相反，ARIES 在事务日志中写入补偿日志，逻辑上删除被删除事务的日志记录。

HUBBLE

当一个事务被“手动”取消，或者被锁管理器取消 (以停止死锁)，或者仅仅是因为网络故障，那么分析传递就不需要了。事实上，关于重做和撤消操作的信息可以在两个内存表中找到:

- 事务表 (存储所有当前事务的状态)
- 脏页表 (存储哪些数据需要写到磁盘上)。

对于每个新的事务事件，缓存管理器和事务管理器将更新这些表。由于它们位于内存中，所以当数据库崩溃时它们将被销毁。

分析阶段的工作是使用事务日志中的信息在崩溃后重新创建两个表。为了加快分析过程，ARIES 提供了检查点的概念。其思想是在磁盘上不定期地写入事务表和脏页表的内容，以及写入时的最后一个 LSN，以便在分析过程中，只分析 LSN 之后的日志。

7.7.7 结论

在写这篇文章之前，我知道这个主题有多重要，我知道要写一篇深入的文章需要时间。结果我花了比预期多两倍的时间，但我学到了很多。

如果你想对数据库有一个很好的概述,我建议你阅读研究论文“[数据库系统的架构体系]”(http://db.cs.berkeley.edu/papers/fn architecture.pdf)。这是一个关于数据库的很好的介绍 (110 页)，而且它是非 cs 人员可以读懂的。这篇文章帮助我为这篇文章找到了一个计划，它不像我的文章那样关注数据结构和算法，而是更多地关注架构概念。

如果您仔细阅读这篇文章，您现在应该了解数据库是多么强大。由于这是一篇很长的文章，让我来回顾我们所看到的:

- B+ 树索引的概述
- 数据库的全局概览
- 概述 CBO，重点关注连接操作符
- 缓冲池管理概述
- 事务管理概述

但是数据库包含了更多的精巧之处。例如，我没有谈论一些敏感的问题，比如:

- 如何管理集群数据库和全局事务
- 如何在做一个快照时，数据库仍保持运行
- 如何有效地存储 (和压缩) 数据
- 如何管理内存

因此，当您必须在有缺陷的 NoSQL 数据库和坚如磐石的关系数据库之间进行选择时，请三思。不要误解我，一些 NoSQL 数据库非常棒。但他们还很年轻，还在处理一些与应用程序有关的具体问题。

总之，如果有人问你数据库是如何工作的，你现在可以回答了。

7.8 分布式 ID 生成器

如果在一个分布式系统中生成类似于 MySQL 自增 ID 这样不断增大，同时又不会重复的 ID，使用传统的单机做法就存在问题了，因为在分布式系统中，每秒产生 10W+ 的 ID，ID 每次增 1 的方式就变得不可取了。考虑到多节点的情况，分布式并发的情况下可能出现问题。

分布式系统中，对于 ID 的设计是在保证不重复的情况下又可以排序，设计思路是在 ID 中带有一些时间信息，这样即使进行了分库分表，分节点生成，也能以时间顺序对这些值进行排序。

通用的做法是使用 twitter 的 snowflake 算法。snowflake 由以下部分组成:

```
未使用                    41位时间戳                    datacenter_id
    ↪ sequence_id
```

HUBBLE

```
  ↓                          ↓                                        ↓
      ↪ ↓
|  0 | 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0 | 00000 | 00000| 0000
    ↪  0000 0000|

                                                                          ↑
                                                                      worker_id
```

整个数值的长度是 64 位，int64 类型，被划分为四部分，不含开头的第一个 bit，因为这个 bit 是符号位。用 41 位来表示收到请求时的时间戳，单位为毫秒，接着五位来表示数据中心的 id，后边五位表示机器实例 ID，最后的 12 位为循环自增 id。达到 111111111111 后会归 0。

这样的机制可以支持在同一台机器上，同一毫秒产生 2^{12}=4096 条消息。一秒产生 409.6 万个 id。从值域的角度来讲是完全够用了。

数据中心加上节点 id 共有 10 位，可以支持每个数据中心 32 台机器，所有数据中心共 1024 个节点。表示 timestamp 的 41 位可以支持使用 69 年。如果使用 1970 年的计数方式，那么跑到 2039 年就不能用了，这里的 timestamp 是相对一个时间的增量，也就是偏移量。

7.8.1 worker_id 的分配

`timestamp`、`datacenter_id`、`worker_id`和`sequence_id`这四个字段中，`timestamp`和`sequence_id`是由程序生成的，`datacenter_id`和`worker_id`可以真实的对应部署节点和集群的编号，也可以由程序生成，不同点在于，一旦启动，值就不可更改了，从而解决冲突问题。如果可以随意修改，可能被不慎修改成重复的值，造成生成的 id 有冲突。

一般情况下，既可以通过制定 id 也可以通过程序来给定。

> 在 hubble 中也是采用类似于 snowflake 的生成方法，总体思路就是时间戳 + 网卡，对于复杂的可能加入了 DCE 及命名空间等，如果有长短要求的情况下还会有生成短 id 等不同的处理方式。

7.8.2 开源实现

`github.com/bwmarrin/snowflake`是一个用 go 语言实现的 snowflake 包，对各位的定义为：

1 位符号位，41 位时间戳，10 位 nodeid，12 位序列。和标准的 snowflake 完全一致。

该包的使用

```
package main

import (
    "fmt"
    "os"

    "github.com/bwmarrin/snowflake"
)

func main() {
    n, err := snowflake.NewNode(1)
    if err != nil {
```

```
        println(err)
        os.Exit(1)
    }

    for i :=0; i < 3; i++ {
        id := n.Generate()
        fmt.Println("id", id)
        fmt.Println(
            "node: ", id.Node(),
            "step: ", id.Step(),
            "time: ", id.Time(),
            "\n",
        )
  }
}
```

这个库的可定制部分主要是这些长度:

```
    // Epoch is set to the twitter snowflake epoch of Nov 04 2010 01:42:54 UTC
    // You may customize this to set a different epoch for your application.
    Epoch int64 = 1288834974657

    // Number of bits to use for Node
    // Remember, you have a total 22 bits to share between Node/Step
    NodeBits uint8 = 10

    // Number of bits to use for Step
    // Remember, you have a total 22 bits to share between Node/Step
    StepBits uint8 = 12
```

Epoch时间戳，起始时间。NodeBits机器编号位，StepBits自增序列长。

7.8.3 总结

现在 snowflake 各类语言都有自己的实现，重要的是领会其中的逻辑，根据实际情况定制自己的 id，同时还要考虑生成效率，多思考，例如，为什么传统的 seq 不行呢，在分布式系统中传统序列有什么缺陷。

7.9 分布式追踪系统

7.9.1 1. 介绍

7.9.1.1 为什么需要 tracing

开发和工程团队因为系统组件水平扩展、开发团队小型化、敏捷开发、CD(持续集成)、解耦等各种需求，正在使用现代的架构替代老旧的单机系统。也就是说，当一个生产系统面对真正的高并发，或者解耦成大量微服务时，以前很容易实现的重点任务变得困难了。过程中需要面临一系列问题：用户体验优化、后台真实错误原因分析，分布式系统中各组件的调用情况等。

当代分布式追踪系统（例如 Zipkin,Dapper,Htrace,X-trace 等）旨在解决这些问题，但是他们使用不兼容的 API 来实现各自的应用需求。尽管这些分布式追踪系统有着相似的 API 语法，但各种语言的开发人员依然很难将他们各自的系统（使用不同语言和技术）和特定的分布式追踪系统进行整合。

HUBBLE

7.9.1.2 为什么需要 OpenTracing

OpenTracing 通过提供平台无关、厂商无关的 API，使得开发人员能够方便的添加（或更换）追踪系统的实现。OpenTracing 提供了用于运营支撑系统的和针对特定平台的辅助程序库。程序库的具体信息请参考详细的规范。

7.9.1.3 什么是一个 Trace

在广义上，一个 trace 代表了一个事务或者流程在（分布式）系统中的执行过程。在 OpenTracing 标准中，trace 是多个 span 组成的一个有向无环图（DAG），每一个 span 代表 trace 中被命名并计时的连续性的执行片段。

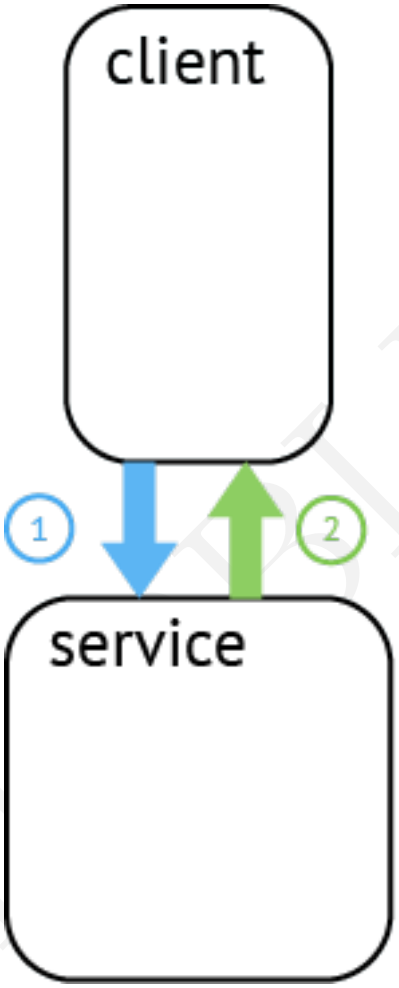


图 89: img

分布式追踪中的每个组件都包含自己的一个或者多个 span。例如，在一个常规的 RPC 调用过程中，OpenTracing 推荐在 RPC 的客户端和服务端，至少各有一个 span，用于记录 RPC 调用的客户端和服务端信息。

一个父级的 span 会显示的并行或者串行启动多个子 span。在 OpenTracing 标准中，甚至允许一个子 span 有多个父 span （例如：并行写入的缓存，可能通过一次刷新操作写入动作）。

7.9.1.4 一个典型的 Trace 案例

在一个分布式系统中，追踪一个事务或者调用流一般如上图所示。虽然这种图对于看清各组件的组合关系是很有用的，但是，它不能很好显示组件的调用时间，是串行调用还是并行调用，如果展现更复杂的调用关系，会更加复杂，甚至无法画出这样的图。另外，这种图也无法显示调用间的时间间隔以及是否通过定时调用来启动调用。一种更有效的展现一个典型的 trace 过程，如下图所示：

这种展现方式增加显示了执行时间的上下文，相关服务间的层次关系，进程或者任务的串行或并行调用关系。这

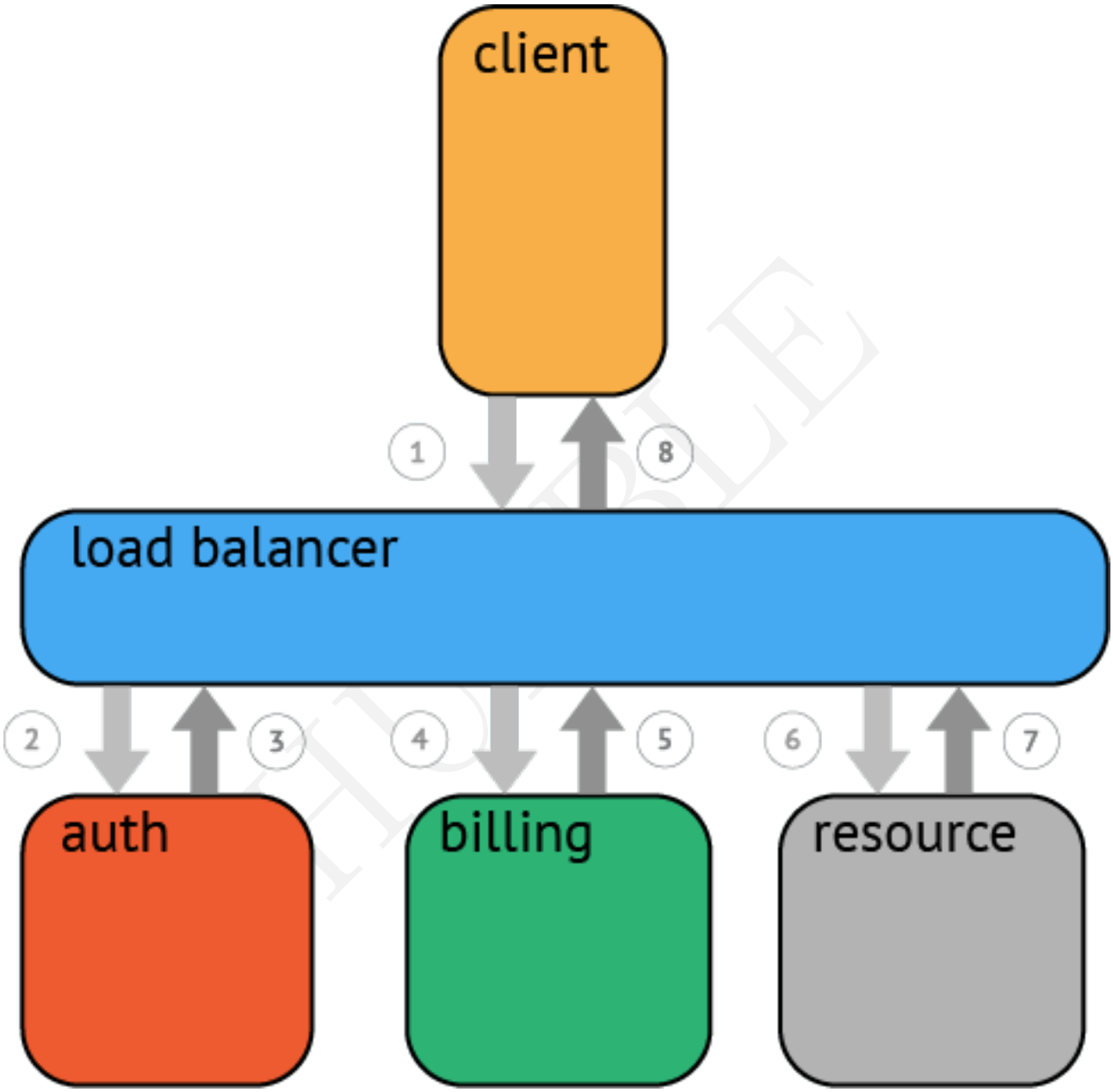


图 90: img

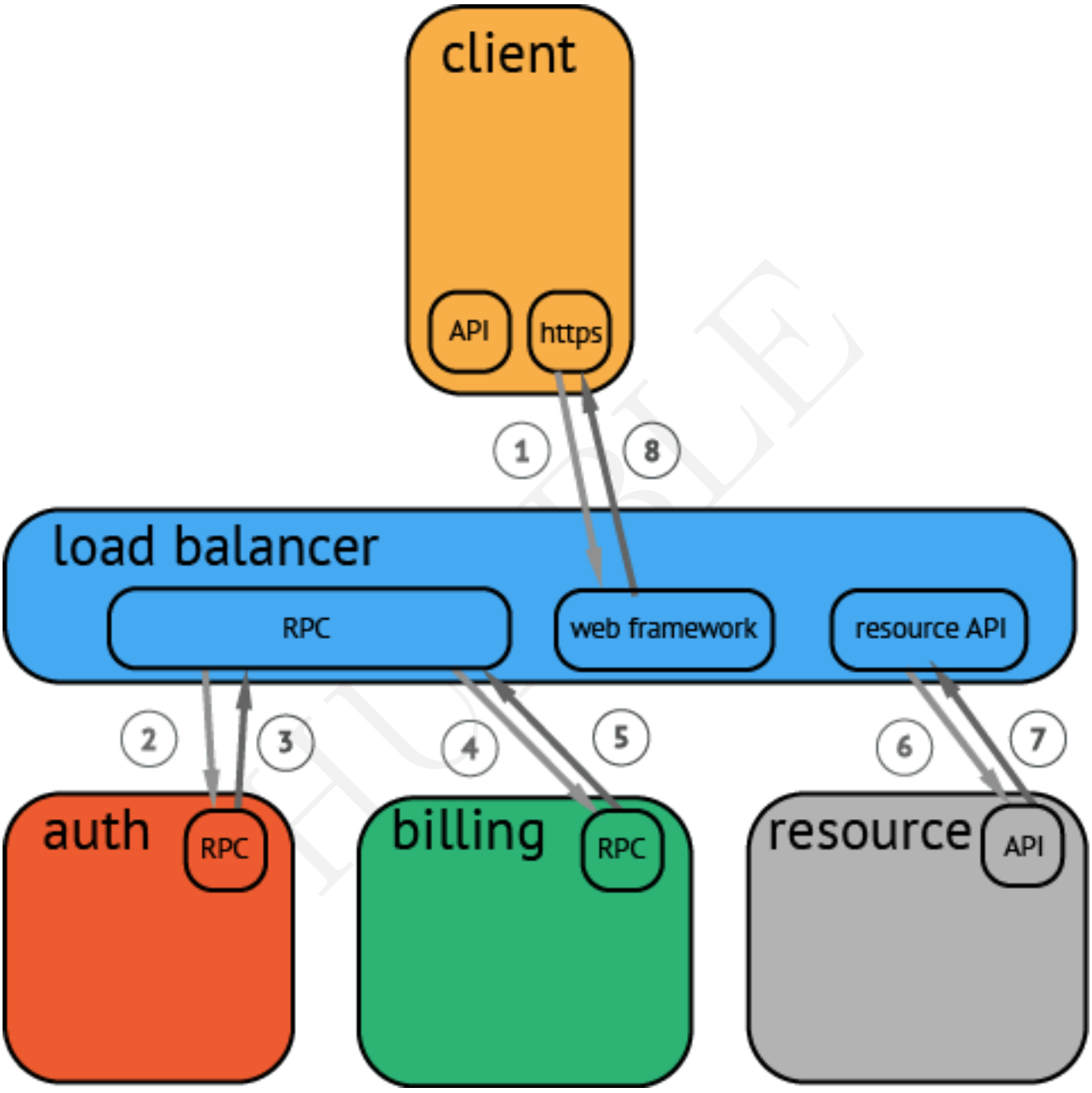

client
API
https
1
8
load balancer
RPC
web framework
resource API
2
3
4
5
6
7
auth
RPC
billing
RPC
resource
API

图 91: img

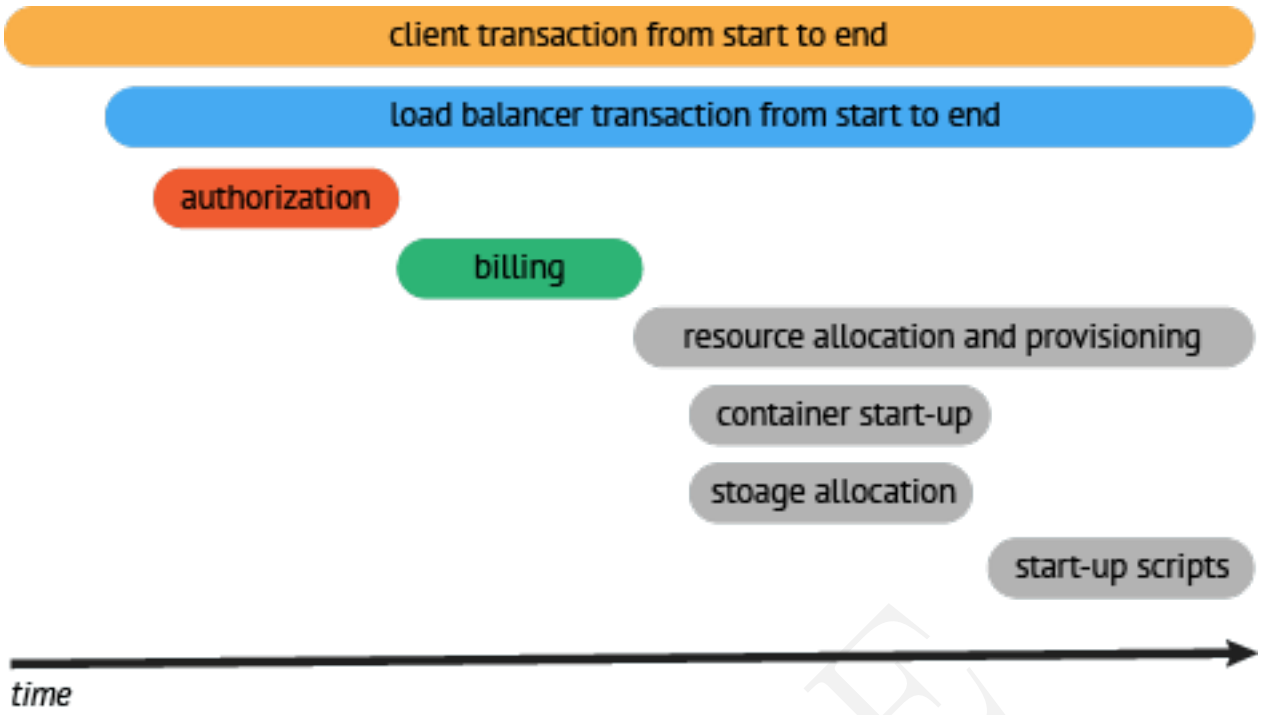


图 92: img

样的视图有助于发现系统调用的关键路径。通过关注关键路径的执行过程，项目团队可能专注于优化路径中的关键位置，最大幅度的提升系统性能。例如：可以通过追踪一个资源定位的调用情况，明确底层的调用情况，发现哪些操作有阻塞的情况。

7.9.2 2. 快速入门

随着并发和异步成为现代软件应用的必然特性，分布式追踪系统成为有效监控的一个必须的组成部分。尽管如此，监控并追踪一个系统的调用情况，至今仍是一个耗时而复杂的任务。随着系统的调用分布式程度（超过 10 个进程）和并发度越来越高，移动端与 web 端、客户端到服务端的调用关系越来越复杂，追踪调用关系带来的好处是显而易见的。但是选择和部署一个追踪系统的过程十分复杂。OpenTracing 标准将改变这一点，OpenTracing 尽力让监控一个分布式调用过程简单化。

试想一个简单的 web 网站。当用户访问你的首页时，web 服务器发起两个 HTTP 调用，其中每个调用又访问了数据库。这个过程是否简单直白，我们可以不费什么力气就能发现请求缓慢的原因。如果你考虑到调用延迟，你可以为每个调用分布式唯一的 ID，并通过 HTTP 头进行传递。如果请求耗时过长，你通过使用唯一 ID 来 grep 日志文件，发现问题出在哪里。现在，想想一下，你的 web 网站变得流行起来，你开始使用分布式架构，你的应用需要跨越多个机器，多个服务来工作。随着机器和服务数量的增长，日志文件能明确解决问题的机会越来越少。确定问题发生的原因将越来越困难。这时，你发现投入调用流程追踪能力是非常有价值的。

正如我提到的，OpenTracing 因为 standardizes instrumentation, 监控标准化，会使得追踪过程变得容易。它意味着，你可以先进行追踪，再决定最终的实现方案。

以AppDash为例，你可以根据如下的步骤，从编译 web 项目到查看追踪信息。或者，你可以直接使用 Appdash 来完成追踪并查看追踪信息。

```
docker run --rm -ti -p 8080:8080 -p 8700:8700 bg451/opentracing-example
```

这将启动一个测试的本地的 Appdash 实例。点击查看源码

如果你想看到完成的实例，你可以根据下面的步骤，自己构建 webapp，使用 OpenTracing 设置追踪，绑定到一个追踪系统（如 AppDash），并最终查看调用情况。

7.9.2.1 创建一个 web 工程

在开始之前，先写几个简单的调用点:

```
// Acts as our index page
func indexHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte(`<a href="/home"> Click here to start a request </a>`))
}
func homeHandler(w http.ResponseWriter, r *http.Request) {
     w.Write([]byte("Request started"))
    go func() {
        http.Get("http://localhost:8080/async")
    }()
    http.Get("http://localhost:8080/service")
    time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
    w.Write([]byte("Request done!"))
}
// Mocks a service endpoint that makes a DB call
func serviceHandler(w http.ResponseWriter, r *http.Request) {
    // ...
    http.Get("http://localhost:8080/db")
    time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
    // ...
}
// Mocks a DB call
func dbHandler(w http.ResponseWriter, r *http.Request) {
    time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
    // here would be the actual call to a DB.
}
```

将这些调用点组合成一个 server

```
func main() {
    port := 8080
    addr := fmt.Sprintf(":%d", port)
    mux := http.NewServeMux()
    mux.HandleFunc("/", indexHandler)
    mux.HandleFunc("/home", homeHandler)
    mux.HandleFunc("/async", serviceHandler)
    mux.HandleFunc("/service", serviceHandler)
    mux.HandleFunc("/db", dbHandler)
    fmt.Printf("Go to http://localhost:%d/home to start a request!\n", port)
    log.Fatal(http.ListenAndServe(addr, mux))
}
```

将这些放到main.go文件中，运行go run main.go。

7.9.2.2 监控应用程序

HUBBLE

现在，你有了一个可以工作的 web 应用服务器，你可以开始监控它了。你可以开始像下面这样，在入口设置一个 span：

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
  span := opentracing.StartSpan("/home") // Start a span using the global, in
      ↪ this case noop, tracer
  defer span.Finish()
  // ... the rest of the function
}
```

这个 span 记录 homeHandler 方法完成所需的时间，这只是可以记录的信息的冰山一角。OpenTracing 允许你为每一个 span 设置 tags 和 logs。例如：你可以通过 homeHandler 方法是否正确返回，决定是否记录方法调用的错误信息：

```
// The ext package provides a set of standardized tags available for use.
import "github.com/opentracing/opentracing-go/ext"

func homeHandler(w http.ResponseWriter, r *http.Request) {
// ...
// We record any errors now.
    _, err := http.Get("http://localhost:8080/service")
    if err != nil {
        ext.Error.Set(span, true) // Tag the span as errored
        span.LogEventWithPayload("GET service error", err) // Log the error
    }
// ...
}
```

你也可以添加其他事件信息，如：发生的重要事件，用户 id，浏览器类型。

然而，这只是其中的一个功能。为了构建真正的端到端追踪，你需要包含调用 HTTP 请求的客户端的 span 信息。在我们的示例中，你需要在端到端过程中传递 span 的上下文信息，使得各端中的 span 可以合并到一个追踪过程中。这就是 API 中 Inject/Extract 的职责。homeHandler 方法在第一次被调用时，创建一个根 span，后续过程如下：

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Request started"))
    span := opentracing.StartSpan("/home")
    defer span.Finish()

    // Since we have to inject our span into the HTTP headers, we create a
       ↪ request
    asyncReq, _ := http.NewRequest("GET", "http://localhost:8080/async", nil)
    // Inject the span context into the header
    err := span.Tracer().Inject(span.Context(),
        opentracing.TextMap,
        opentracing.HTTPHeaderTextMapCarrier(asyncReq.Header))
    if err != nil {
        log.Fatalf("Could not inject span context into header: %v", err)
```

HUBBLE

```
    }
    go func() {
        if _, err := http.DefaultClient.Do(asyncReq); err != nil {
            span.SetTag("error", true)
            span.LogEvent(fmt.Sprintf("GET /async error: %v", err))
        }
    }()
    // Repeat for the /service call.
    // ....
}
```

上述代码，在底层实际的执行逻辑是：将关于本地追踪调用的 span 的元信息，被设置到 http 的头上，并准备传递出去。下面会展示如何在 serviceHandler 服务中提取这个元数据信息。

```
func serviceHandler(w http.ResponseWriter, r *http.Request) {
    var sp opentracing.Span
    opName := r.URL.Path
    // Attempt to join a trace by getting trace context from the headers.
    wireContext, err := opentracing.GlobalTracer().Extract(
        opentracing.TextMap,
        opentracing.HTTPHeaderTextMapCarrier(r.Header))
    if err != nil {
        // If for whatever reason we can't join, go ahead an start a new root
        ↪ span.
        sp = opentracing.StartSpan(opName)
    } else {
        sp = opentracing.StartSpan(opName, opentracing.ChildOf(wireContext))
    }
  defer sp.Finish()
  // ... rest of the function
```

HUBBLE

如上述程序所示，你可以通过 http 头获取元数据。你可以重复此步骤，为你需要追踪的调用进行设置，很快，你将可以监控整套系统。如何决定哪些调用需要被追踪呢？你可以考虑你的调用的关键路径。

7.9.2.3 连接到追踪系统

OpenTracing 最重要的作用就是，当你的系统按照标准被监控之后，增加一个追踪系统将变得非常简单！在这个示例，你可以看到，我使用了一个叫做 Appdash 的开源追踪系统。你需要通过在 main 函数中增加一小段代码，来启动 Appdash 实例。但是，你不需要修改任何你关于监控的代码。在你的 main 函数中，加入如下内容：

```
import (
    "sourcegraph.com/sourcegraph/appdash"
     “sourcegraph.com/sourcegraph/appdash/traceapp”
    appdashot "sourcegraph.com/sourcegraph/appdash/opentracing"
)

func main() {
    // ...
      store := appdash.NewMemoryStore()
```

```
    // Listen on any available TCP port locally.
    l, err := net.ListenTCP("tcp", &net.TCPAddr{IP: net.IPv4(127, 0, 0, 1), Port
        ↪ : 0})
    if err != nil {
        log.Fatal(err)
    }
    collectorPort := l.Addr().(*net.TCPAddr).Port
    collectorAdd := fmt.Sprintf(":%d", collectorPort)

    // Start an Appdash collection server that will listen for spans and
    // annotations and add them to the local collector (stored in-memory).
    cs := appdash.NewServer(l, appdash.NewLocalCollector(store))
    go cs.Start()

    // Print the URL at which the web UI will be running.
    appdashPort := 8700
    appdashURLStr := fmt.Sprintf("http://localhost:%d", appdashPort)
    appdashURL, err := url.Parse(appdashURLStr)
    if err != nil {
        log.Fatalf("Error parsing %s: %s", appdashURLStr, err)
    }
    fmt.Printf("To see your traces, go to %s/traces\n", appdashURL)

    // Start the web UI in a separate goroutine.
    tapp, err := traceapp.New(nil, appdashURL)
    if err != nil {
        log.Fatal(err)
    }
    tapp.Store = store
    tapp.Queryer = store
    go func() {
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d", appdashPort), tapp))
    }()

    tracer := appdashot.NewTracer(appdash.NewRemoteCollector(collectorPort))
    opentracing.InitGlobalTracer(tracer)
// ...
}
```

这样你会增加一个嵌入式的 Appdash 实例，并对本地程序进行监控。

如果你想换一个监控系统的实现，如果他们都符合 OpenTracing，你只需要进行一步操作。你只需要修改你的 main 函数，其他所有的监控代码，都可以保持不变。例如，如果你决定使用 Zipkin，你只需要在 main 函数中进行如下修改：

```
import zipkin "github.com/openzipkin/zipkin-go-opentracing"
```

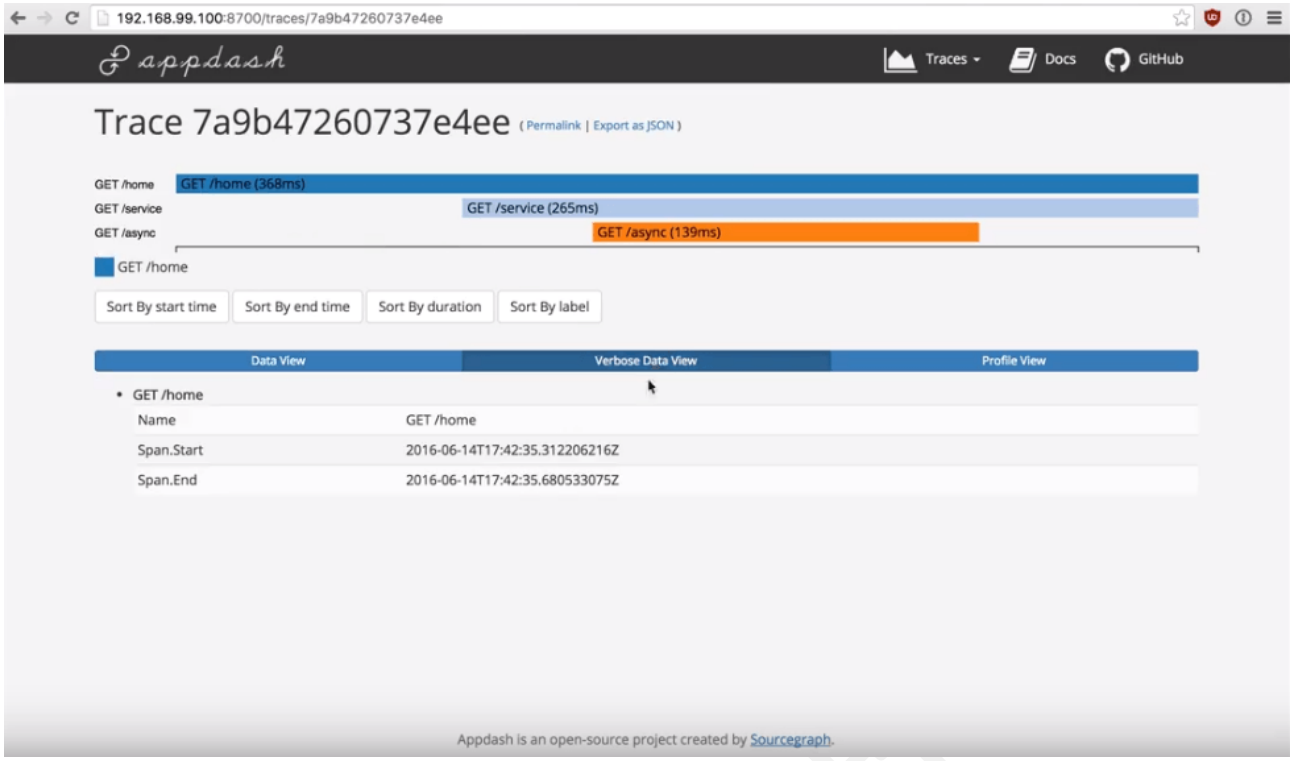


图 93: image alt text

```
func main() {
  // ...
  // Replace Appdash tracer code with this
  collector, err := zipkin.NewKafkaCollector("ZIPKIN_ADDR")
  if err != nil {
    log.Fatal(err)
    return
  }

  tracer, err = zipkin.NewTracer(
    zipkin.NewRecorder(collector, false, "localhost:8000", "example"),
  )
  if err != nil {
    log.Fatal(err)
  }
  opentracing.InitGlobalTracer(tracer)
  // ...
}
```

到目前为止，你会发现，使用 OpenTracing 使得监控你的代码更简单。我推荐在启动一个新项目的研发过程中，就加入监控的代码。因为，即使你的应用很小，追踪数据也可以在你的应用演进，引入分布式的时候，提供数据支持。帮助你在这个过程中，构建一个可持续迭代的产品。

7.9.3 3. 概念和术语

一个 tracer 过程中，各 span 的关系

```
        [Span A]  ←←←(the root span)
            |
     +------+------+
     |             |
 [Span B]      [Span C] ←←←(Span C 是 Span A 的孩子节点, ChildOf)
     |             |
 [Span D]      +---+-------+
               |           |
           [Span E]    [Span F] >>> [Span G] >>> [Span H]
                                       ↑
                                       ↑
                                       ↑
                         (Span G 在 Span F 后被调用, FollowsFrom)
```

上述 tracer 与 span 的时间轴关系

```
––|–––––––|–––––––|–––––––|–––––––|–––––––|–––––––|–––––––|–> time

 [Span A···················································]
   [Span B··············································]
      [Span D··········································]
    [Span C········································]
         [Span E·······]        [Span F··] [Span G··] [Span H··]
```

Traces

一个 trace 代表一个潜在的，分布式的，存在并行数据或并行执行轨迹（潜在的分布式、并行）的系统。一个 trace 可以认为是多个 span 的有向无环图（DAG）。

Spans

一个 span 代表系统中具有开始时间和执行时长的逻辑运行单元。span 之间通过嵌套或者顺序排列建立逻辑因果关系。

Operation Names

每一个 span 都有一个操作名称，这个名称简单，并具有可读性高。（例如：一个 RPC 方法的名称，一个函数名，或者一个大型计算过程中的子任务或阶段）。span 的操作名应该是一个抽象、通用的标识，能够明确的、具有统计意义的名称；更具体的子类型的描述，请使用 Tags。

例如，假设一个获取账户信息的 span 会有如下可能的名称：

操作名	指导意见
get	太抽象
get_account/792	太明确
get_account	正确的操作名，关于 account_id=792 的信息应该使用 tag 操作

Inter-Span References

一个 span 可以和一个或者多个 span 间存在因果关系。OpenTracing 定义了两种关系：ChildOf 和 FollowsFrom。

这两种引用类型代表了子节点和父节点间的直接因果关系。未来，OpenTracing 将支持非因果关系的 span 引用关系。（例如：多个 span 被批量处理，span 在同一个队列中，等等）

ChildOf 引用：一个 span 可能是一个父级 span 的孩子，即"ChildOf" 关系。在"ChildOf" 引用关系下，父级 span 某种程度上取决于子 span。下面这些情况会构成"ChildOf" 关系：

- 一个 RPC 调用的服务端的 span，和 RPC 服务客户端的 span 构成 ChildOf 关系
- 一个 sql insert 操作的 span，和 ORM 的 save 方法的 span 构成 ChildOf 关系
- 很多 span 可以并行工作（或者分布式工作）都可能是一个父级的 span 的子项，他会合并所有子 span 的执行结果，并在指定期限内返回

下面都是合理的表述一个"ChildOf" 关系的父子节点关系的时序图。

```
    [-Parent Span---------]
         [-Child Span----]

    [-Parent Span--------------]
         [-Child Span A----]
          [-Child Span B----]
        [-Child Span C----]
         [-Child Span D---------------]
         [-Child Span E----]
```

FollowsFrom 引用：一些父级节点不以任何方式依然他们子节点的执行结果，这种情况下，我们说这些子 span 和父 span 之间是"FollowsFrom" 的因果关系。"FollowsFrom" 关系可以被分为很多不同的子类型，未来版本的 OpenTracing 中将正式的区分这些类型下面都是合理的表述一个"FollowFrom" 关系的父子节点关系的时序图。

```
    [-Parent Span-]  [-Child Span-]


    [-Parent Span--]
     [-Child Span-]


    [-Parent Span-]
                [-Child Span-]
```

Logs

每个 span 可以进行多次 Logs 操作，每一次 Logs 操作，都需要一个带时间戳的时间名称，以及可选的任意大小的存储结构。标准中定义了一些日志（logging）操作的一些常见用例和相关的 log 事件的键值，可参考 Data Conventions 数据约定指南。

Tags

每个 span 可以有多个键值对（key:value）形式的 Tags，Tags 是没有时间戳的，支持简单的对 span 进行注解和补充。

和使用 Logs 的场景一样，对于应用程序特定场景已知的键值对 Tags，tracer 可以对他们特别关注一下。更多信息，可参考 Data Conventions 数据约定指南。

SpanContext

HUBBLE

每个 span 必须提供方法访问 SpanContext。SpanContext 代表跨越进程边界，传递到下级 span 的状态。(例如，包含<trace_id, span_id, sampled>元组)，并用于封装 Baggage (关于 Baggage 的解释，请参考下文)。SpanContext 在跨越进程边界，和在追踪图中创建边界的时候会使用。(ChildOf 关系或者其他关系，参考 Span 间关系)。

Baggage

Baggage 是存储在 SpanContext 中的一个键值对 (SpanContext) 集合。它会在一条追踪链路上的所有 span 内全局传输，包含这些 span 对应的 SpanContexts。在这种情况下，"Baggage" 会随着 trace 一同传播，他因此得名 (Baggage 可理解为随着 trace 运行过程传送的行李)。鉴于全栈 OpenTracing 集成的需要，Baggage 通过透明化的传输任意应用程序的数据，实现强大的功能。例如：可以在最终用户的手机端添加一个 Baggage 元素，并通过分布式追踪系统传递到存储层，然后再通过反向构建调用栈，定位过程中消耗很大的 SQL 查询语句。

Baggage 拥有强大功能，也会有很大的消耗。由于 Baggage 的全局传输，如果包含的数量量太大，或者元素太多，它将降低系统的吞吐量或增加 RPC 的延迟。

Baggage vs. Span Tags

- Baggage 在全局范围内，(伴随业务系统的调用) 跨进程传输数据。Span 的 tag 不会进行传输，因为他们不会被子级的 span 继承。
- span 的 tag 可以用来记录业务相关的数据，并存储于追踪系统中。实现 OpenTracing 时，可以选择是否存储 Baggage 中的非业务数据，OpenTracing 标准不强制要求实现此特性。

Inject and Extract

SpanContexts 可以通过 Injected 操作向 Carrier 增加，或者通过 Extracted 从 Carrier 中获取，跨进程通讯数据 (例如：HTTP 头)。通过这种方式，SpanContexts 可以跨越进程边界，并提供足够的信息来建立跨进程的 span 间关系 (因此可以实现跨进程连续追踪)。

7.9.3.1 平台无关的 API 语义

OpenTracing 支持了很多不同的平台，当然，每个平台的 API 试图保持各平台和语言的习惯和管理，尽量做到入乡随俗。也就是说，每个平台的 API，都需要根据上述的核心 tracing 概念来建模实现。在这一章中，我们试图描述这些概念和语义，尽量减少语言和平台的影响。

The Span Interface

Span 接口必须实现以下的功能：

- `Get the Span's SpanContext`，通过 span 获取 SpanContext (即使 span 已经结束，或者即将结束)
- `Finish`，完成已经开始的 Span。处理获取 SpanContext 之外，Finish 必须是 span 实例的最后一个被调用的方法。(py: finish, go: Finish)。一些的语言实现方法会在 Span 结束之前记录相关信息，因为 Finish 方法可能不会被调用，因为主线程处理失败或者其他程序错误。在这种情况下，实现应该明确的记录 Span，保证数据的持久化。
- `Set a key:value tag on the Span.`，为 Span 设置 tag。tag 的 key 必须是 string 类型，value 必须是 string，boolean 或数字类型。tag 其他类型的 value 是没有定义的。如果多个 value 对应同一个 key (例如被设置多次)，实现方式是没有被定义的。(py: set_tag, go: SetTag)
- `Add a new log event`，为 Span 增加一个 log 事件。事件名称是 string 类型，参数值可以是任何类型，任何大小。tracer 的实现者不一定保存所有的参数值 (设置可以所有参数值都不保存)。其中的时间戳参数，可以设置当前时间之前的时间戳。(py: log, go: Log)
- `Set a Baggage item`, 设置一个 string:string 类型的键值对。注意，新设置的 Baggage 元素，只保证传递到未来的子级的 Span。参考下图所示。(py: set_baggage_item, go: SetBaggageItem)
- `Get a Baggage item`，通过 key 获取 Baggage 中的元素。(py: get_baggage_item, go: BaggageItem)

```
        [Span A]
            |
    +------+------+
    |             |
[Span B]      [Span C] ←←← (1) BAGGAGE ITEM "X" IS SET ON SPAN C,BUT AFTER SPAN
   ↪  E ALREADY STARTED.
    |             |            为SPAN C设置BAGGAGE元素，值为X，时间点为SPAN E已
       ↪ 经开始运行
[Span D]      +---+-----+
              |         |
          [Span E]  [Span F] >>> [Span G] >>> [Span H]
                                                ↑
                                                ↑
                                                ↑
           (2) BAGGAGE ITEM "X" IS AVAILABLE FOR RETRIEVAL BY SPAN H (A CHILD
              ↪ OF SPAN C), AS WELL AS SPANS F AND G.
               SPAN C元素的F\G\H子级span可以读取到BAGGAGE的元素X
```

The SpanContext Interface

SpanContext 接口必须实现以下功能。用户可以通过 Span 实例或者 Tracer 的 Extract 能力获取 SpanContext 接口实例。

- Iterate over all Baggage items 是一个只读特性。(py: baggage, go: ForeachBaggageItem)
- 虽然以前 SpanContext 是 Tracer 接口的一部分，但是 SpanContext 对于 Inject and Extract 是必不可少的。

The Tracer Interface

Tracer 接口必须实现以下功能：

- `Start a new Span`, 创建一个新的 Span。调用者可以指定一个或多个 SpanContext 关系 (例如 FollowsFrom 或 ChildOf 关系)，显示声明一个开始的时间戳（除"now" 之外），并设置处理化的 Span 的 tags 数据集。(py: start_span, go: StartSpan)
- `Inject a SpanContext`, 将 SpanContext 注入到 SpanContext 对象中，用于进行跨进程的传输。"carrier" 类型需要反射或者明确指定的方式来确定。查看 end-to-end propagation example 端到端传递示例获取更多消息。
- `Extract a SpanContext` ，通过"carrier" 跨进程获取 SpanContext 信息。Extract 会检查 carrier，尝试获取先前通过 Inject 放入的数据，并重建 SpanContext 实例。除非有错误发生，Extract 返回一个包含 SpanContext 实例，此实例可以用来创建一个新的子级 Span。(注意：一些 OpenTracing 实现方式，认为 Span 在 RPC 的两端应该具有相同的 ID，而另一些考虑客户端是父级 span，服务端是子级 span)。"carrier" 类型需要反射或者明确指定的方式来确定。查看 end-to-end propagation example 端到端传递示例获取更多消息。

Global and No-op Tracers

每一个平台的 OpenTracing API 库 (例如 opentracing-go, opentracing-java，等等；不包含 OpenTracing Tracer 接口的实现) 必须提供一个 no-op Tracer（不具有任何操作的 tracer）作为接口的一部分。No-op Tracer 的实现必须不会出错，并且不会有任何副作用，包括 baggage 的传递时，也不会出现任何问题。同样，Tracer 的实现也必须提供 no-op Span 实现；通过这种方法，监控代码不依赖于 Tracer 关于 Span 的返回值，针对 no-op 实

现，不需要修改任何源代码。No-op Tracer 的 Inject 方法永远返回成功，Extract 返回的效果，和”carrier” 中没有找到 SpanContext 时返回的结果一样。

每一个平台的 OpenTracing API 库可能支持配置 (Go: InitGlobalTracer(), py: opentracing.tracer = myTracer) 和获取单例的全局 Tracer 实例 (Go: GlobalTracer(), py: opentracing.tracer)。如果支持全局的 Tracer，默认返回的必须是 no-op Tracer。

7.9.4 4.OpenTracing API

7.9.4.1 OpenTracing 多语言支持

OpenTracing API 支持以下平台：

- Go - https://github.com/opentracing/opentracing-go
- Python - https://github.com/opentracing/opentracing-python
- Javascript - https://github.com/opentracing/opentracing-javascript
- Objective-C - https://github.com/opentracing/opentracing-objc
- Java - https://github.com/opentracing/opentracing-java
- C++ - https://github.com/opentracing/opentracing-cpp

OpenTracing API 以下平台实现正在研发中：

- PHP - link forthcoming
- Ruby - link forthcoming

有关各平台的使用示例，请参考查阅各平台仓库的 README 文档。

7.9.4.2 Data Conventions 数据约定

7.9.4.2.1 介绍

OpenTracing 通过定义的 API，可实现将监控数据记录到一个可插拔的 tracer 上。总体上来说，OpenTracing 不能保证底层追踪系统的实现方式。那么 API 层应该提供什么类型的数据来保证这些底层追踪系统实现的兼容性呢?

监控软件和追踪软件开发者在高层次的共识，将产生巨大的价值：如果在一些通用的应用场景下，都使用某些已知的 tag 的键值对，tracer 程序可以选择对他们进行特别的关注。被 log 的事件，span 的结构也是如此。

例如，考虑基于 HTTP 的应用服务器。应用系统处理的请求中的 URL、HTTP 动作（get/post 等）、返回码，对于应用系统的诊断是非常有帮助的。监控者可以选择使用 tag 方法标记这个参数，命名为 URL 或 http.url，从纯 API 技术角度来说是有效的。但是，如果一个 tracer 需要增加一些高级功能，例如根据 URL 的值建立索引，或者针对特定来源的请求进行采样，你必须知道数据的格式。换句话说，tag 的名字和监控程序的提供方的要求必须是一致的，这样追踪程序才能在收到数据后，提供更加智能的分析结果。

本文档对追踪软件开发和探针软件开发都有通用指导意义。追踪系统的开发者不必严格遵守指南，但是强烈推荐大家这么做。

7.9.4.2.2 Spans

Span Naming, Span 命名

Span 可以包含很多的 tags、logs 和 baggage，但是始终需要一个高度概括的`operation name`。这些应该是一个简单的字符串，代表 span 中进行的工作类型。这个字符串应该是工作类型的逻辑名称，例如代表一个 RPC 或者一次 HTTP 的调用的端点，亦或对于代表 SQL 的 span，使用 SELECT or INSERT 作为逻辑名，等等。

其次，span 可以存在一个可选的 tag，叫做 component，他的值可以典型的代表一个进程、框架、类库或者模块名称。这个 tag 会很有价值。

Span Structure, Span 的结构

Span 的结构也是非常重要的：span 代表了什么，span 和 span 的上下级是什么关系？请参考 3. 概念和术语

Span Tag Use-Cases, Span Tag 操作用例

监控软件开发者，如果试图标注如下特定类型的数据，请使用下面推荐的 tags。tag 名称遵循命名空间的通用结构（即：java 包名的结构）

下面推荐的 tag，在 ext 模块中，都会为每一个实现制定一个 const 常量值。这些 ext 的值应该用来代表下面的字符串，不同的追踪系统，可以为这些通用概念选择不同的底层实现。在每种实现中，这些值的实现方式是十分相似的。(例如：Go, Python)

下面提供的一下 tags 可能包含一些象征大小的值。如何处理这些值是依赖于实现的：追踪系统会需要适当选择，是否要使用、删除或者清空这些 tags 标记。然而，不仅仅追踪程序才需要关注这些值，给追踪系统生成、传递这些值，也可能对应用系统造成不良影响。

监控系统可以只支持其中的部分 tags。

Errors

一个 span 实例的错误状态，通过一个 tag 来标注。

- error - bool
  - true 代表这个 span 是错误状态
  - false 或没有 error tag ，代表 span 没有发生错误

Component Identification, 框架定义

对于任何一个 span，被监控的组件，指定组件的类型是十分有帮助的。十分推荐库或者模块为监控程序提供组件的定义，最终用户可能会拥有一个由框架和第三方混合提供的监控。

- component - string
  - 需要被检测/监控的类库、模块、包的基本名称。
  - Examples:
    - httplib 代表 Python 内建的 httplib 函数功能
    - JDBC 代表 JDBC 数据库连接
    - mongoose 代表 Ruby 的 MongoDB 客户端连接
- span.kind - string
  - client 或 server, 指定这个 span 代表一个客户端还是服务端

HTTP Server Tags

这些 tag 作用于基于 HTTP 的服务入口的 span。

- http.url - string
  - URL 分布式追踪中，这一阶段的调用的 URL 地址, 参考 standard URI format.
  - Protocol 协议，可选
  - Examples:
    - https://domain.net/path/to?resource=here
    - domain.net/path/to/resource
    - http://user:pass@domain.org:8888
- http.method - string
  - HTTP 请求被处理的方法.

HUBBLE

  - Case-insensitive 大小写敏感
  - Examples:
    * GET, POST, HEAD
- http.status_code - integer
  - HTTP 返回值
  - Examples:
    * 200, 503
- span.kind - string
  - server 定义这是服务端类型的 span

Peer Tags

这些 tag 可以被客户端或者服务端提供，用于描述远程请求过程中，请求调用的方向。（客户端记录下行访问，服务端记录上行访问）

- peer.hostname - string
  - 目标 hostname
- peer.ipv4 - string
  - 目标 IP v4 地址
- peer.ipv6 - string
  - 目标 IP v6 地址
- peer.port - integer
  - 目标 port
- peer.service - string
  - 目标服务名称

Sampling, 采样

OpenTracing API 不强调采样的概念，但是大多数追踪系统通过不同方式实现采样。有些情况下，应用系统需要通知追踪程序，这条特定的调用需要被记录，即使根据默认采样规则，它不需要被记录。sampling.priority tag 提供这样的方式。追踪系统不保证一定采纳这个参数，但是会尽可能的保留这条调用。

- sampling.priority - integer
  - 如果大于 0, 追踪系统尽可能保存这条调用链
  - 等于 0, 追踪系统不保存这条调用链
  - 如果此 tag 没有提供，追踪系统使用自己的默认采样规则

Logs

Common fields

OpenTracing 中的每一次日志记录都会包含一个时间戳，并至少包含一个基于键值对的域。以下是一些标准化的域定义 Every Log record in OpenTracing has a timestamp and at least one key:value "field". The following fields are standardized:

- event - string
  - 事件域代表 span 生命周期内某些关键时间点的标识。例如，在浏览器页面加载过程中，获得或释放一个互斥锁就是一个特定的事件域，可参考Performance.timing标准.

7.9.4.3 跨进程追踪

开发者为应用程序增加跨进程追踪能力时，必须理解 the OpenTracing specification 中定义的 Tracer. Inject(...) 和 Tracer.Extract(...) 的能力。这两个方法在概念上十分强大，他允许开发人员正确并抽象的完成跨进程传输的代码，而不需要绑定特定的 OpenTracing 的实现；也就是说，强大的能力带来了巨大的困惑:)

HUBBLE

这篇文档，针对 Inject 和 Extract 设计和用法，提供一个简要的总结，而不考虑特定的 OpenTracingg 规范各语言的实现和基于 OpenTracing 标准的追踪系统。

显示的 trace 传播的“重大作用”

分布式追踪系统最困难的部分就是在分布式的应用环境下保持追踪的正常工作。任何一个追踪系统，都需要理解多个跨进程调用间的因果关系，无论他们是通过 RPC 框架、发布-订阅机制、通用消息队列、HTTP 请求调用、UDP 传输或者其他传输模式。

一些分布式追踪系统（例如，2003 年的 Project5，2006 年的 WAP5，2014 年的 The Mystery Machine）会推断跨进程间的因果关系。当然，这些系统，都需要在基于黑盒的因果关系推断与追踪结果的整合、实时准确展现上，进行处理折衷。处于对准确展现的关注，OpenTracing 是一个明确的分布式追踪系统标准，它更倾向于如果产品的处理方式：2007 年的 X-Trace，2010 年的 Dapper，以及很多开源的追踪系统，如：Zipkin，Appdash 等等

Inject 和 Extract 允许开发者进行跨进程追踪时，不用和特定的 OpenTracing 实现进行紧耦合。

OpenTracing 跨进程传播需求

为了使 Inject 和 Extract 有效，必须遵守如下要求：

如上文所述，OpenTracing 用户处理跨进程的追踪传输时，必须不需要使用 OpenTracing 使用中的特定代码。

基于 OpenTracing 的追踪系统，必须不需要针对每一种已知的跨进程通讯机制都进行处理：这其中包含太多的工作，很多还没有明确的定义。

也就是说，这套传播机制必须是最利于扩展的。

基本方法：Inject, Extract, 和 Carriers

追踪过程中的任何一个 SpanContext 可以被 Injected（注入）到一个 Carrier 中。Carrier 可以是一个接口或者一个数据载体，他对于跨进程通讯（IPC）是十分有帮助的。Carrier 负责将追踪状态从一个进程”carries”（携带，传递）到另一个进程。OpenTracing 标准包含两种必须的 Carrier 格式，尽管，自定义的 Carrier 格式也是可能的。

同样的，对于一个 Carrier，如果已经被 Injected，那么它也可以被 Extracted（提取），从而得到一个 SpanContext 实例。这个 SpanContext 代表着被 Injected 到 Carrier 的信息。

Inject 伪代码示例

```
span_context = ...
outbound_request = ...

# 我们将使用（内建的）基于HTTP_HEADERS的carrier格式。
# 我们在调用`tracer.inject之前，先将一个空的map作为一个carrier
carrier = {}
tracer.inject(span_context, opentracing.Format.HTTP_HEADERS, carrier)

# `carrier` 现在（隐形）包含我们通过网络传输的键值对。
for key, value in carrier:
    outbound_request.headers[key] = escape(value)
```

Extract 伪代码示例

```
inbound_request = ...
```

```
# 我们将再次使用基于（内建的）HTTP_HEADERS carrier格式。
# 按照HTTP_HEADERS的文档，我们可以使用一个map来存储外来的值，
# 允许OpenTracing实现者来根据需要，
# 来查找集合内部的键值对。
#
# 也就是说，我们直接使用基于键值对的`inbound_request.headers`作为carrier。
carrier = inbound_request.headers
span_context = tracer.extract(opentracing.Format.HTTP_HEADERS, carrier)
# Continue the trace given span_context. E.g.,
span = tracer.start_span("...", child_of=span_context)

# （如果 `carrier` 保存着trace的数据， 则现在可以创建`span`了。）
```

Carrier 格式

所有的 Carrier 都有自己的格式。在一些语言的 OpenTracing 实现中，格式必须必须作为一个常量或者字符串来指定；另一些，则通过 Carrier 的静态类型来指定。

Inject/Extract Carrier 所必须的格式

至少，OpenTracing 标准所有平台的实现者支持两种 Carrier 格式：基于"text map"（基于字符串的 map）的格式和基于"binary"（二进制）的格式。

- text map 格式的 Carrier 是一个平台惯用的 map 格式，基于 unicode 编码的字符串对字符串键值对
- binary 格式的 Carrier 是一个不透明的二进制数组（可能更紧凑和有效）

OpenTracing 的实现者选择如何将数据存储到 Carrier 中，OpenTracing 标准没有正式定义，但是，可以推测的是，他们会通过一种方式编码 "追踪状态"，来传递 SpanContext （例如，Dapper 会包含`trace_id`，`span_id`，以及一位掩码标识这个 trace 的采样状态）和 Baggage 中的其他键值对数据。

各种 OpenTracing 实现者，实现跨进程边界方式的互操作性

不能期待不同的 OpenTracing 实现，Inject 和 Extract SpanContexts 采用相互兼容的方式。虽然 OpenTracing 对于实现跨整个分布式系统的追踪系统是无从得知的，为了成功实现跨进程的追踪的我收过程，跨进程追踪的两端应该使用相同的追踪系统实现。（即远程调用的两段，使用同一套 tracer）。

自定义的 Inject/Extract Carrier 格式

任何的基于网络传输的子系统（RPC 库，消息队列等）可能选择引入他们自定义的 Inject/Extract 的 Carrier 格式；根据需要自定义格式，但最终要求返回符合 OpenTracing 格式的结果。这样允许 OpenTracing 的实现者可以优化他们自己的自定义格式，而不需要实现者支持这些子系统的自定义格式。

一些伪代码将可能更明确的说明这个问题。假设我们是 ArrrPC pirate RPC subsystem 的作者，我们希望增加 OpenTracing 的数据在 RPC 请求过程中传输。不考虑异常处理，我们的伪代码可能如下所示：

```
span_context = ...
outbound_request = ...

# 首先，我们使用我们自定义的Carrier： outbound_request
# 如果我们优先支持OpenTracing的实现，这样会更加高效的处理。
# 但是，这不是一个必须的格式要求，我们不能指望基于OpenTracing的
# 追踪程序支持arrrpc.ARRRPC_OT_CARRIER参数
try:
    tracer.inject(span_context, arrrpc.ARRRPC_OT_CARRIER, outbound_request)
```

```
except opentracing.UnsupportedFormatException:
    # If unsupported, fall back on a required OpenTracing format.
    # 如果不支持，则使用OpenTracing支持的格式
    carrier = {}
    tracer.inject(span_context, opentracing.Format.HTTP_HEADERS, carrier)
    # `carrier` 现在包含键值对，我们可以使用任何网络协议，来传输这个键值对即可
    for key, value in carrier:
    outbound_request.headers[key] = escape(value)
```

关于 Carrier 自定义格式的更多内容

"Carrier 的格式" 在不同平台可能是不一样的，但在所有的场景下，他们都会使用一个全局的命名空间。支持一个全新的自定义格式的 carrier 不必修改 OpenTracing 核心平台的 API，尽管每一个实现 OpenTracing 平台 API 时，必须定义符合 OpenTracing 标准要求的 carrier 格式（比如：基于字符串的 map 和二进制块）。例如，ArrrPC RPC 的维护团队定义了一个叫做"ArrrPC" 的 Inject/Extract 格式，他们不需要向 OpenTracing 团队提交 PR（当然 OpenTracing 的实现者不要求一定支持"ArrrPC" 格式）。an end-to-end injector and extractor example below，一个端到端的 injector 和 extractor 示例将更具体的描述这个问题。

一个端到端的 injector 和 extractor 示例

为了让描述更具体，考虑如下的流程：

1. 一个客户端进程有一个 SpanContext 实例，并准备进行通过自制的 HTTP 协议，进行一次 RPC 调用
2. 客户端进程调用 Tracer.Inject(...)，传入 SpanContext 实例，支持基于字符格式的 map 的标识符，以及支持基于字符 map 的 Carrier，三个参数
3. Carrier 将在 Carrier 对基于字符的 map（参数 2）填充必要的数据；客户端应用程序会在自制的 HTTP 协议中，对这个 map 进行编码（如：添加到 HTTP 头中）
4. 进行 HTTP 调用，将数据进行跨进程传输
5. 现在，在服务端进程进行处理。应用程序从自制的 HTTP 协议中解码出上文所述的 map（参数 2），并通过他，初始化基于字符 map 的 Carrier
6. 服务端进程调用 Tracer.Extract(...)，传入需要的操作名（operation name），支持就要字符格式的 map 的标识符，以及上面构建的 Carrier
7. 不考虑数据丢失，和其他错误，服务端现在有了和客户端追踪上下文中，一样的 SpanContext。

在 OpenTracing use cases, OpenTracing 常见用例文档中，可以找到其他使用案例。

7.9.5 5.Best Practices 最佳实践

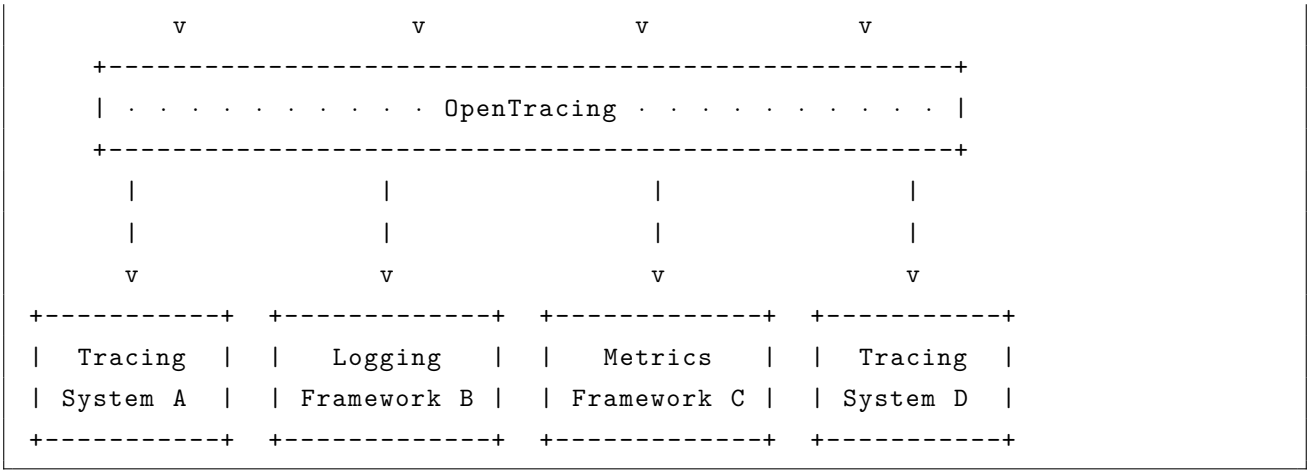
7.9.5.1 常见用例

本章的主要目的是，针对通过使用 OpenTracing API 来监控应用程序或类库的开发者，提供示例说明。

回到伊始：OpenTracing 是为了哪些人建立的？

OpenTracing 是一个轻量级的标准化层，它位于应用程序/类库和追踪或日志分析程序之间。

```
   +-------------+  +---------+  +----------+  +------------+
   | Application |  | Library |  |   OSS    |  |  RPC/IPC   |
   |    Code     |  |  Code   |  | Services |  | Frameworks |
   +-------------+  +---------+  +----------+  +------------+
          |              |             |             |
          |              |             |             |
```



Application Code, 应用程序代码：开发者在开发业务代码时，可以通过 OpenTracing 来描述追踪数据间的因果关系，控制流程，增加细粒度的日志信息。

Library Code, 类库代码：类似的，类库程序作为请求控制的中介媒介，也可以通过 OpenTracing 来描述追踪数据间的因果关系，控制流程，增加细粒度的日志信息。例如：一个 web 中间件类库，可以使用 OpenTracing，在请求被处理时新增 span；或者，一个 ORM 类库，可以使用 OpenTracing 来描述高级别的 ORM 语义和特定 SQL 查询间的关系。

OSS Services, OSS 服务（运营支持服务）：除嵌入式类库以外，整个 OSS 服务可以采取 OpenTracing 标准来，集成分布式追踪系统来处理一个大型的分布式系统中的复杂调用关系。例如，一个 HTTP 的负载均衡器可以使用 OpenTracing 标准来设置请求（如：设置请求图），或者一个基于键值对的存储系统使用 OpenTracing 来解读系统的读写性能。

RPC/IPC Frameworks，RPC/IPC 框架（远程调用框架）：任何一个跨进程的子任务，都可以通过使用 OpenTracing，来标准化追踪数据注入到传输协议中的格式。

所有上面这些，都应该使用 OpenTracing 来描述和传递分布式追踪数据，而不需要了解 OpenTracing 的实现。

OpenTracing 优先级

由于 OpenTracing 层的上层有更多的应用程序和开发者（而不是下层），API 和用例的易用性也倾向于他们。这篇文档中的用例将面向 OpenTracing API 调用者（而非被调者），帮助他们在建立辅助的类库和各种抽象模型，最终有利于为 OpenTracing 实现者节省时间和精力。

7.9.5.1.1 用例

追踪 Function（函数）

```
def top_level_function():
    span1 = tracer.start_span('top_level_function')
    try:
        . . . # business logic, 业务逻辑
    finally:
        span1.finish()
```

后续，作为业务逻辑的一部分，我们调用了 function2 方法，也想被追踪。为了让这个追踪附着在正在进行的追踪上（和上述的追踪形成一根调用链）。我们将在后面的章节讨论如何实现，现在，我们假设一个get_current_span ↪ 函数可以完成这个功能：

```
def function2():
```

```
    span2 = get_current_span().start_child('function2') \
        if get_current_span() else None
    try:
        . . . # business logic
    finally:
        if span2:
            span2.finish()
```

我们假设，如果这个追踪还未被启动，无论什么原因，开发者都不想在这个函数内启动一个新的追踪，所以我们考虑到 get_current_span 函数可能返回 None。

这两个例子都非常的简单。通常情况下，应用程序不希望追踪代码和业务代码混在一起，而使用其他方式，例如：标注等，参考function decorator in Python:

```
@traced_function
def top_level_function():
    ... # business logic
```

服务端追踪

当一个应用服务器要追踪一个请求的执行情况，他一般需要以下几步：

1. 试图从请求中获取传输过来的 SpanContext（防止调用链在客户端已经开启），如果无法获取 SpanContext，则新开启一个追踪。
2. 在 request context 中存储最新 d 创建的 span，request context 会通过应用程序代码或者 RPC 框架进行传输
3. 最终，当服务端完成请求处理后，使用 span.finish() 关闭 span。

从请求中获取（Extracting）SpanContext

假设，我们有一个 HTTP 服务器，SpanContext 通过 HTTP 头从客户端传递到服务端，可通过 request.headers 访问到：

```
extracted_context = tracer.extract(
    format=opentracing.HTTP_HEADER_FORMAT,
    carrier=request.headers
)
```

这里，我们使用 headers 中的 map 作为 carrier。追踪程序知道需要 hearder 的哪些内容，用来重新构建 tracer 的状态和 Baggage。

从请求中获取一个已经存在的追踪，或者开启一个新的追踪

如果无法在请求的相关的头信息中获取所需的值，上文中的 extracted_context 可能为 None：此时我们假设客户端没有发送他们。在这种情况下，服务端需要新创建一个追踪（新调用链）。

```
extracted_context = tracer.extract(
    format=opentracing.HTTP_HEADER_FORMAT,
    carrier=request.headers
)
if extracted_context is None:
    span = tracer.start_span(operation_name=operation)
else:
```

HUBBLE

```
    span = tracer.start_span(operation_name=operation, child_of=
        ↪ extracted_context)
span.set_tag('http.method', request.method)
span.set_tag('http.url', request.full_url)
```

可以通过调用 set_tag，在 Span 中记录请求的附加信息。

上面提到的 operation 是通过提供的服务名指定 Span 的名称。例如，如果 HTTP 请求到/save_user/123，那么 operation 名称应该被设置为 post:/save_user/。OpenTracing API 不会强制要求应用程序如何给 span 命名。

进程内请求上下文传输

请求的上下文传输是指，对于一个请求，所有处理这个请求的层都需要可以访问到同一个 context（上下文）。可以通过特定值，例如：用户 id、token、请求的截止时间等，获取到这个 context（上下文）。也可以通过这种方法获取正在追踪的 Span。

请求 context（上下文）的传输不属于 OpenTracing API 的范围，但是，这里提到他，是为了让大家更好的理解后面的章节。下面有两种常用的上下文传输技术：

- 隐式传输

  隐式传输技术要求 context（上下文）需要被存储到平台特定的位置，允许从应用程序的任何地方获取这个值。常用的 RPC 框架会利用 thread-local 或 continuation-local 存储机制，或者全局变量（如果是单线程处理）。

  这种方式的缺点在于，有明显的性能损耗，有些平台比如 Go 不知道基于 thread-local 的存储，隐式传输将几乎不可能实现。

- 显示传输

  显示传输技术要求应用程序代码，包装并传递 context（上下文）对象：

```
func HandleHttp(w http.ResponseWriter, req *http.Request) {
    ctx := context.Background()
    ...
    BusinessFunction1(ctx, arg1, ...)
}

func BusinessFunction1(ctx context.Context, arg1...) {
    ...
    BusinessFunction2(ctx, arg1, ...)
}

func BusinessFunction2(ctx context.Context, arg1...) {
    parentSpan := opentracing.SpanFromContext(ctx)
    childSpan := opentracing.StartSpan(
        "...", opentracing.ChildOf(parentSpan.Context()), ...)
    ...
}
```

  显示传输的缺点在于，它向应用程序代码，暴露了底层的实现。Go blog post这边文章提供了这种方式的深层次的解析。

HUBBLE

追踪客户端调用

当一个应用程序作为一个 RPC 客户端时，它可能希望在发起调用之前，启动一个新的追踪的 span，并将这个心的 span 随请求一起传输。下面，通过一个 HTTP 请求的实例，展现如何做到这点。

```
def traced_request(request, operation, http_client):
    # retrieve current span from propagated request context
    parent_span = get_current_span()

    # start a new span to represent the RPC
    span = tracer.start_span(
        operation_name=operation,
        child_of=parent_span.context,
        tags={'http.url': request.full_url}
    )

    # propagate the Span via HTTP request headers
    tracer.inject(
        span.context,
        format=opentracing.HTTP_HEADER_FORMAT,
        carrier=request.headers)

    # define a callback where we can finish the span
    def on_done(future):
        if future.exception():
            span.log(event='rpc exception', payload=exception)
        span.set_tag('http.status_code', future.result().status_code)
        span.finish()

    try:
        future = http_client.execute(request)
        future.add_done_callback(on_done)
        return future
    except Exception e:
        span.log(event='general exception', payload=e)
        span.finish()
        raise
```

- get_current_span() 函数不是 OpenTracing API 的一部分。它仅仅代表一个工具类的方法，通过当前的请求上下文获取当前的 span。(在 Python 一般会这样用)。
- 我们假定 HTTP 请求是异步的，所以他会返回一个 Future。我们为这次调用增加的成功回调函数，在回调函数内部完成当前的 span。
- 如果 HTTP 客户端返回一个异常，则通过 log 方法将异常记录到 span 中。
- 因为 HTTP 请求可以在返回 Future 后发生异常，我们使用 try/catch 块，在任何情况下都会完成 span，保证这个 span 会被上报，并避免内存溢出。

使用 Baggage / 分布式上下文传输

上面通过网络在客户端和服务端间传输的 Span 和 Trace，包含了任意的 Baggage。客户端可以使用 Baggage 将

HUBBLE

一些额外的数据传递到服务端，以及这个服务端的下游其他服务器。

```
# client side
span.context.set_baggage_item('auth-token', '.....')

# server side (one or more levels down from the client)
token = span.context.get_baggage_item('auth-token')
```

Logging 事件

我们在客户端 span 的示例代码中，已经使用过 log。事件被记录不会有额外的负载，也不一定必须在 span 创建或完成时进行操作。例如，应用通过可以在执行过程中，通过获取当前请求的当前 span，记录一个缓存未命中事件：

```
span = get_current_span()
span.log(event='cache-miss')
```

tracer 会为事件自动增加一个时间戳，这点和 Span 的 tag 操作时不同的。也可以将外部的时间戳和事件相关联，例如，Log (Go)。

使用外部的时间戳，记录 Span

因为多种多样的原因，有些场景下，会将 OpenTracing 兼容的 tracer 集成到一个服务中。例如，一个用户有一个日志文件，其中包含大量的来自黑盒进程（如：HAProxy）产生的 span。为了让这些数据接入 OpenTracing 兼容的系统，API 需要提供一种方法通过外部的时间戳记录 span 的信息。

```
explicit_span = tracer.start_span(
    operation_name=external_format.operation,
    start_time=external_format.start,
    tags=external_format.tags
)
explicit_span.finish(
    finish_time=external_format.finish,
    bulk_logs=map(..., external_format.logs)
)
```

在追踪开始之前，设置采样优先级

很多分布式追踪系统，通过采样来降低追踪数据的数量。有时，开发者想有一种方式，确保这条 trace 一定会被记录（采样），例如：HTTP 请求中包含特定的参数，如 debug=true。OpenTracing API 标准化了一些有用的 tag，其中一个被叫做”sampling priority”（采样优先级）：精确的语义是由追踪系统的实现者决定的，但是任何值大于 0（默认）代表一条 trace 的高优先级。为了将 debug 属性传递给追踪系统，需要在追踪前进行预处理，如下面所写的这样：

```
if request.get('debug'):
    span = tracer.start_span(
        operation_name=operation,
        tags={tags.SAMPLING_PRIORITY: 1}
    )
```

7.9.5.2 追踪大型分布式系统

在阅读如何使用 OpenTracing 标准，监控大规模分布式系统之前，确保你已经阅读过概念和术语章节。

7.9.5.2.1 Spans 和它们之间的关系

实现 OpenTracing 完成分布式追踪的两个基本概念就是 Spans 和 Relationships(span 间关系)：

- Span 是系统中的一个逻辑工作单元，包含这个工作单元启动时间和执行时间。在一条追踪链路中，各个 span 与系统中的不同组件有关，并体现这些组件的执行路径。

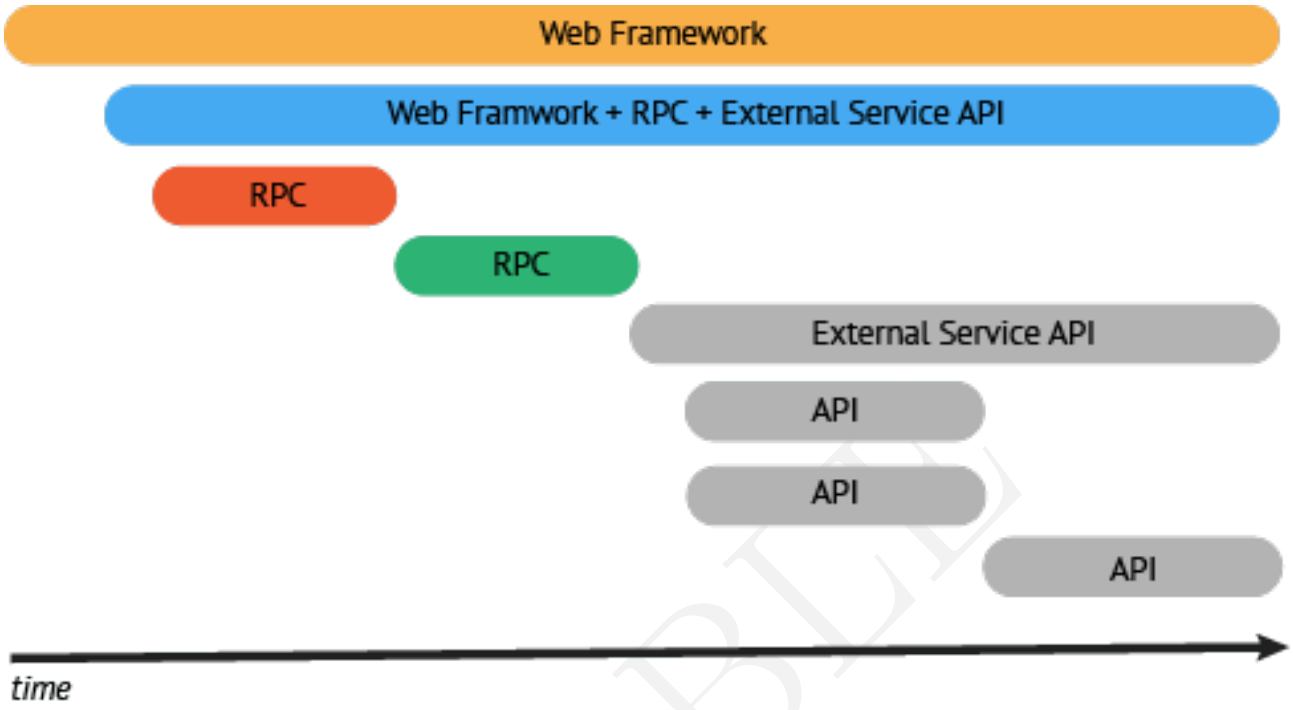


图 94: OTHT_0.png

- Relationships 是 span 间的连接关系。一个 span 可以和 0-n 个组件存在因果关系。这种关系是的各个 span 被串接起来，并用来帮助定位追踪链路的关键路径。

你所期待的结束状态，是获取你所有组件的 span，以及它们之间的关系。当开始建立你的分布式追踪系统时，最好的方法是从服务框架（如：RPC 层）或者其他和复杂执行路径有关的组件开始。

你可以从使用支持 OpenTracing 标准的服务框架开始 (如：gRPC)。

专注高价值区域

如上面提到的，从 RPC 层和你的 web 框架开始构建追踪，是一个好方法。这两部分将包含事务路径中的大部分内容。

下一步，你应该着手在没有被服务框架覆盖的事务路径上。为足够多的组件增加监控，为高价值的事务创建一条关键链路的追踪轨迹。

你监控的首要目标，是基于关键路径上的 span，寻找最耗时的操作，为可量化的优化操作提供最重要的数据支持。例如，对于只占用事务时间 1% 的操作（一个大粒度的 span）增加更细粒度的监控，对于你理解端到端的延迟（性能问题）不会有太大意义。

先走再跑，逐步提高

如果你正在构建你的跨应用追踪系统实现，使用这套系统建立高价值的关键事务与平衡关键事务和代码覆盖率的概念。最大的价值，在于为关键事务生成端到端的追踪。可视化展现你的追踪结果是非常重要的。它可能帮助你确定那块区域（代码块/系统模块）需要更细粒度的追踪。

一旦你有了端到端的监控，你很容易评估在哪些区域增加投入，进行更细粒度的追踪，并能确定事情的优先级。

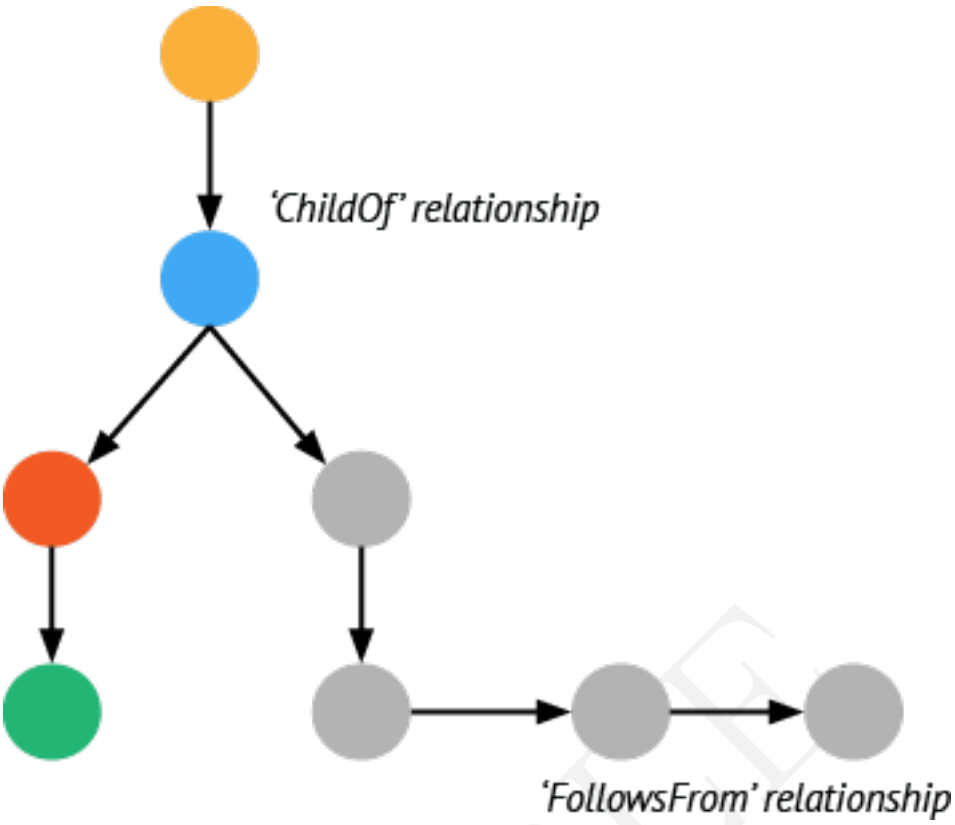


图 95: OTHT_1.png

如果你开始深入处理监控问题，可以考虑哪些部分能够复用。通过这些复用建立一套可以在多个服务间服用的监控类库。

这种方法可以提供广泛的覆盖（如：RPC，web 框架等），也能为关键业务的事务增加高价值的埋点。即使有些埋点（生成 span）的代码是一次性工作，也能通过这种模式发现未来工作的优先级，优化工作效率。

7.9.5.2.2 示例实例

下面的例子让上述的概念更具体一些：

在这个例子中，我们想追踪一个，由手机端发起，调用了多个服务的调用链。

1. 首先，我们必须说明这个事务的大体情况。在我们的例子中，事务如下所示：

2. 现在，我们对事务的大概情况了解，我们去监控一些通用的协议和框架。最好的选择是从 RPC 服务框架开始，这将是收集 web 请求背后发生的调用情况的最好方式。(或者说，任何在分布式过程中发生的问题，都会在直接体现在 RPC 服务中)

3. 下一个重点监控的组件应该是 web 框架。通过增加 web 框架的监控，我们能够得到一个端到端的追踪链路。虽然这点追踪链路有点粗，但是至少，我们的追踪系统获取到了完整的调用栈。

4. 通过上面的工作，我们可以看到所需的调用链，并评估我们细化哪一块的追踪。在我们的例子中，我们可以看到，请求中最耗时的操作时获取资源的操作。所以，我们应该细化这块的监控粒度，监控资源定位内部的组件。一旦我们完成资源请求的监控，我们可以看到资源请求被分解成下图所示的情况

5. 一旦我们完整资源组件的追踪，我们可以看到大量的时间消耗在提供上，下一步，我们深入分析，如果可能，我们优化资源获取程序，使用并行处理替代串行处理。

6. 现在我们有了一条基于端到端调用流程的可视化展现以及基线，我们可以为这个服务建立明确的 SLO。另外，为内部服务建立 SLO，可以成为对服务正常和错误运行的时间的讨论的基础。

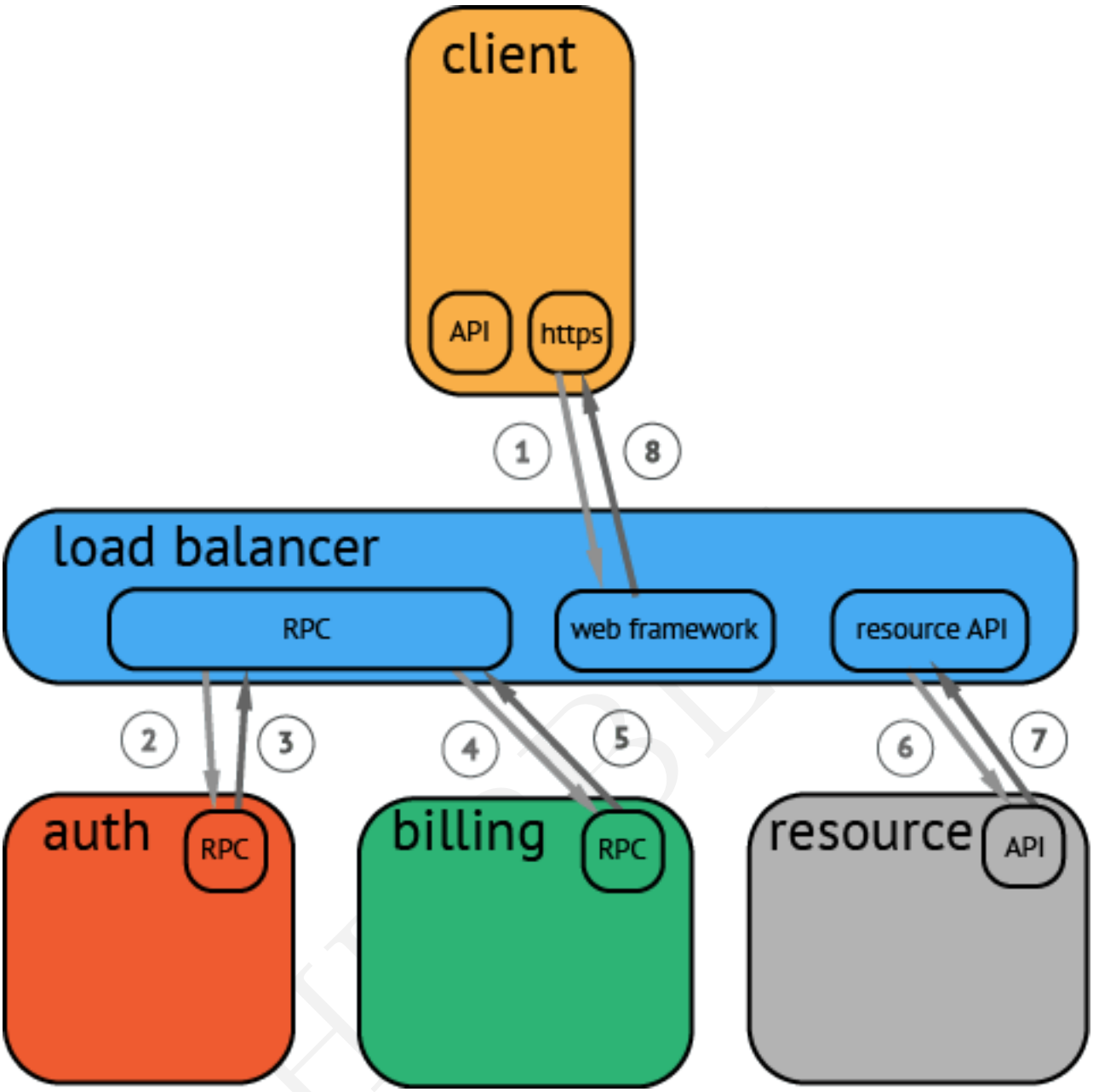


图 96: OTHT_2.png

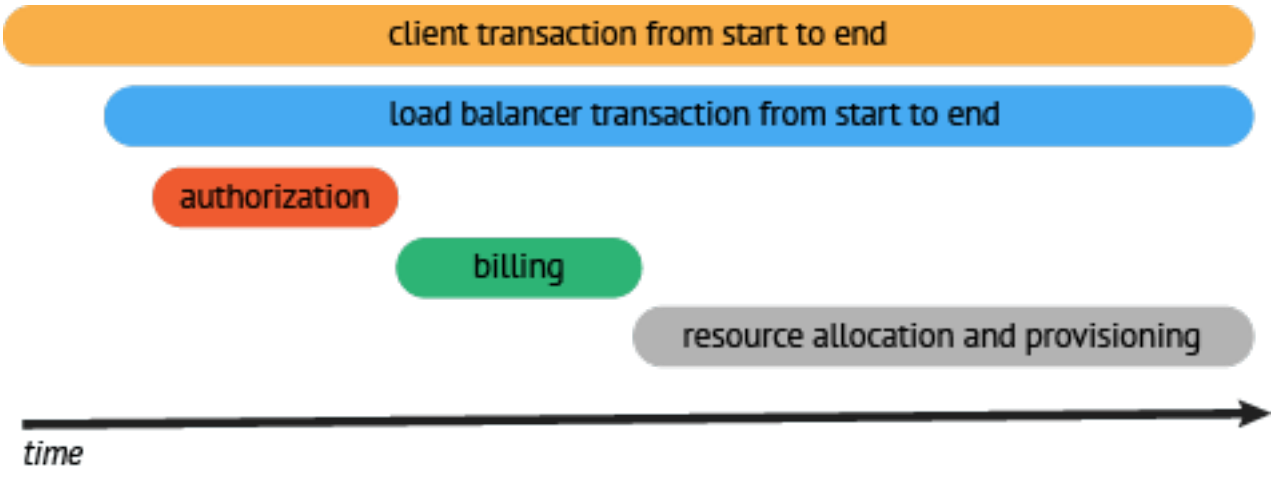


图 97: OTHT_3.png

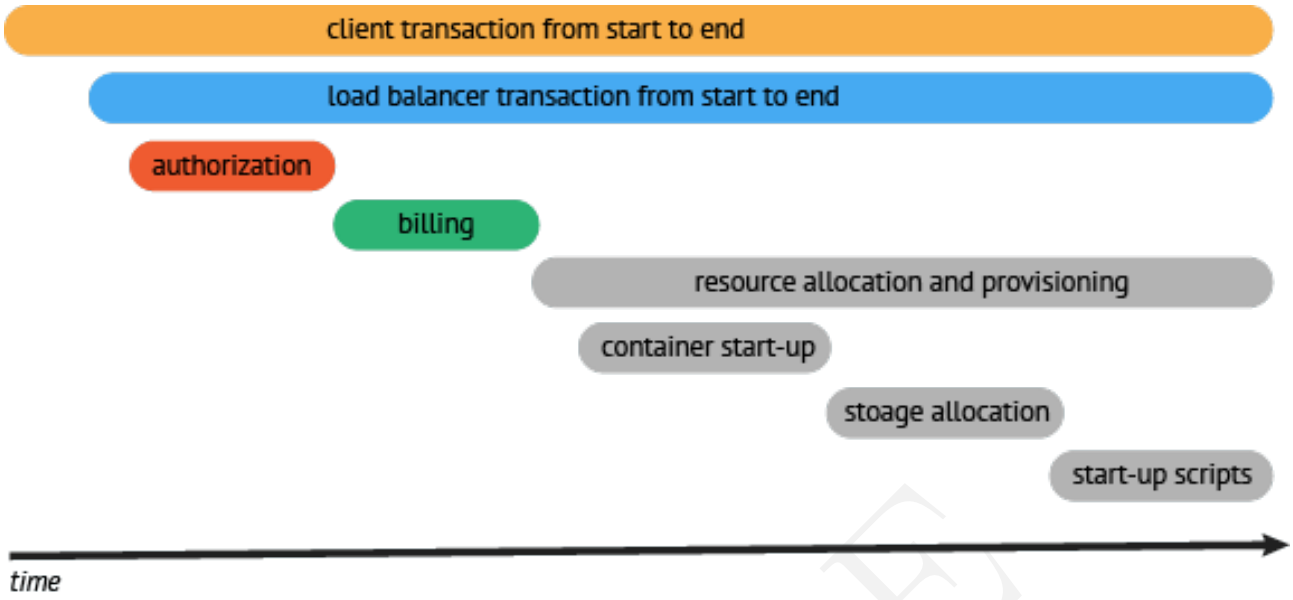


图 98: OTHT_4.png

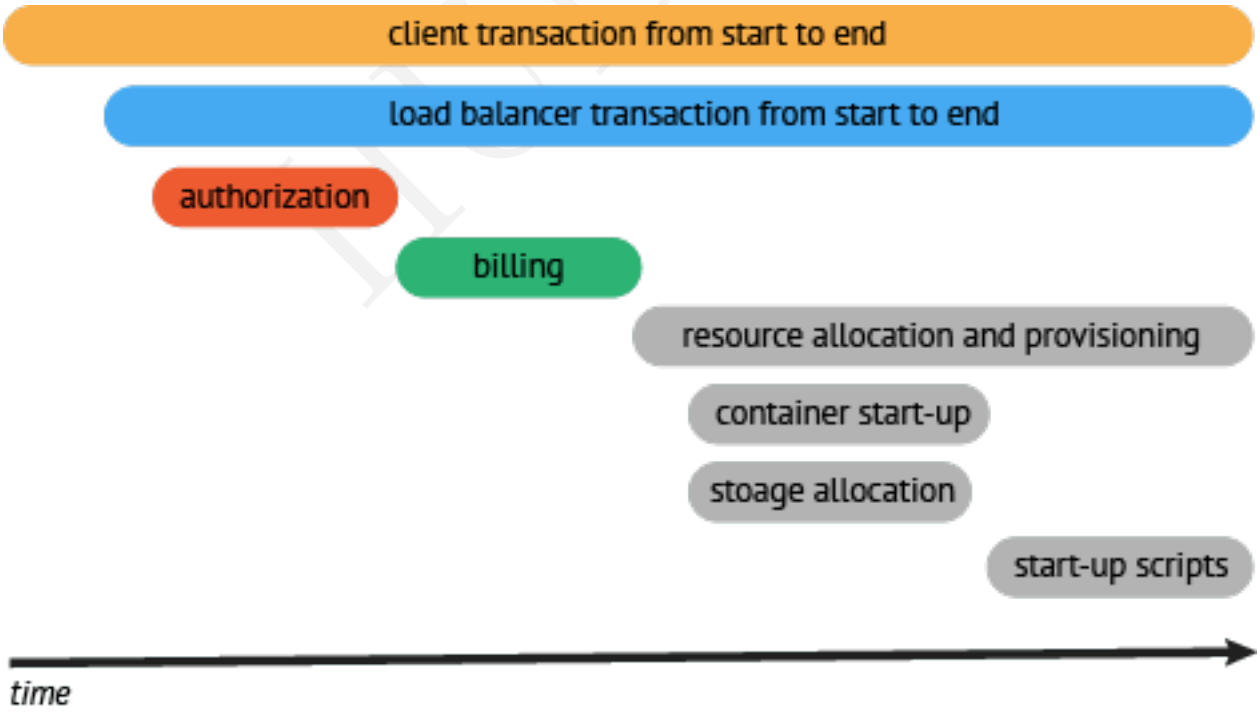


图 99: OTHT_5.png

HUBBLE

7. 下一次跌倒，我们回到最顶层的追踪，去寻找下一个长耗时的任务，但是没有明细展现，这时需要更细粒度的追踪。如果展现的粒度已经足够，我们可以进行下一个关键事务的追踪和调优处理了。

8. 重复上述步骤.

7.10 工具连接

7.10.1 下载

下载地址为 https://dbeaver.io/download/

7.10.2 安装步骤

安装步骤比较简单，全部下一步或者接受即可

7.10.3 工具连接案例

7.10.3.1 步骤一: 新建连接

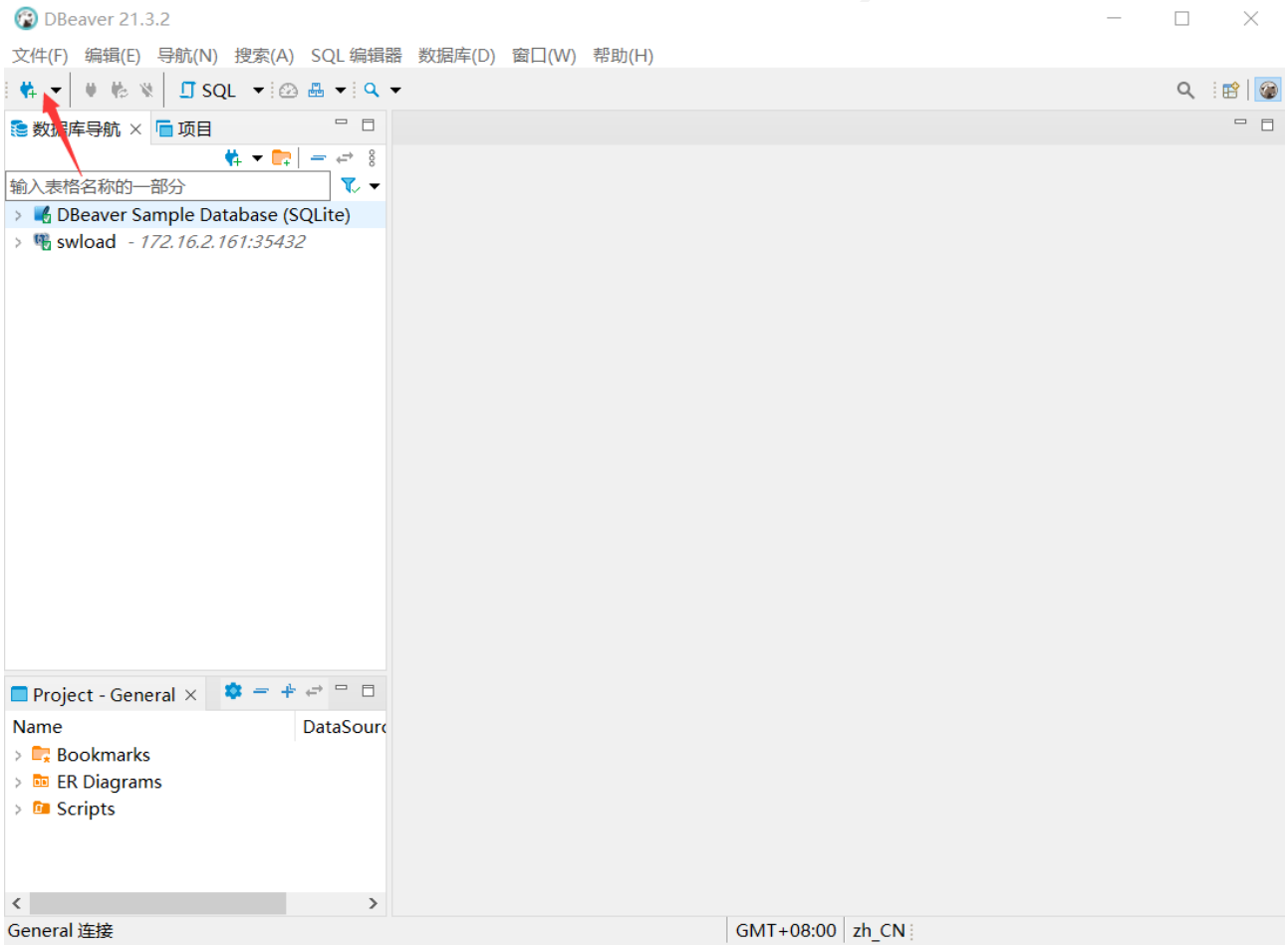


图 100: 图 1

7.10.3.2 步骤二: 选择 SQL 下的 postgresql

7.10.3.3 步骤三: 测试

7.10.3.4 步骤四: 新建模式

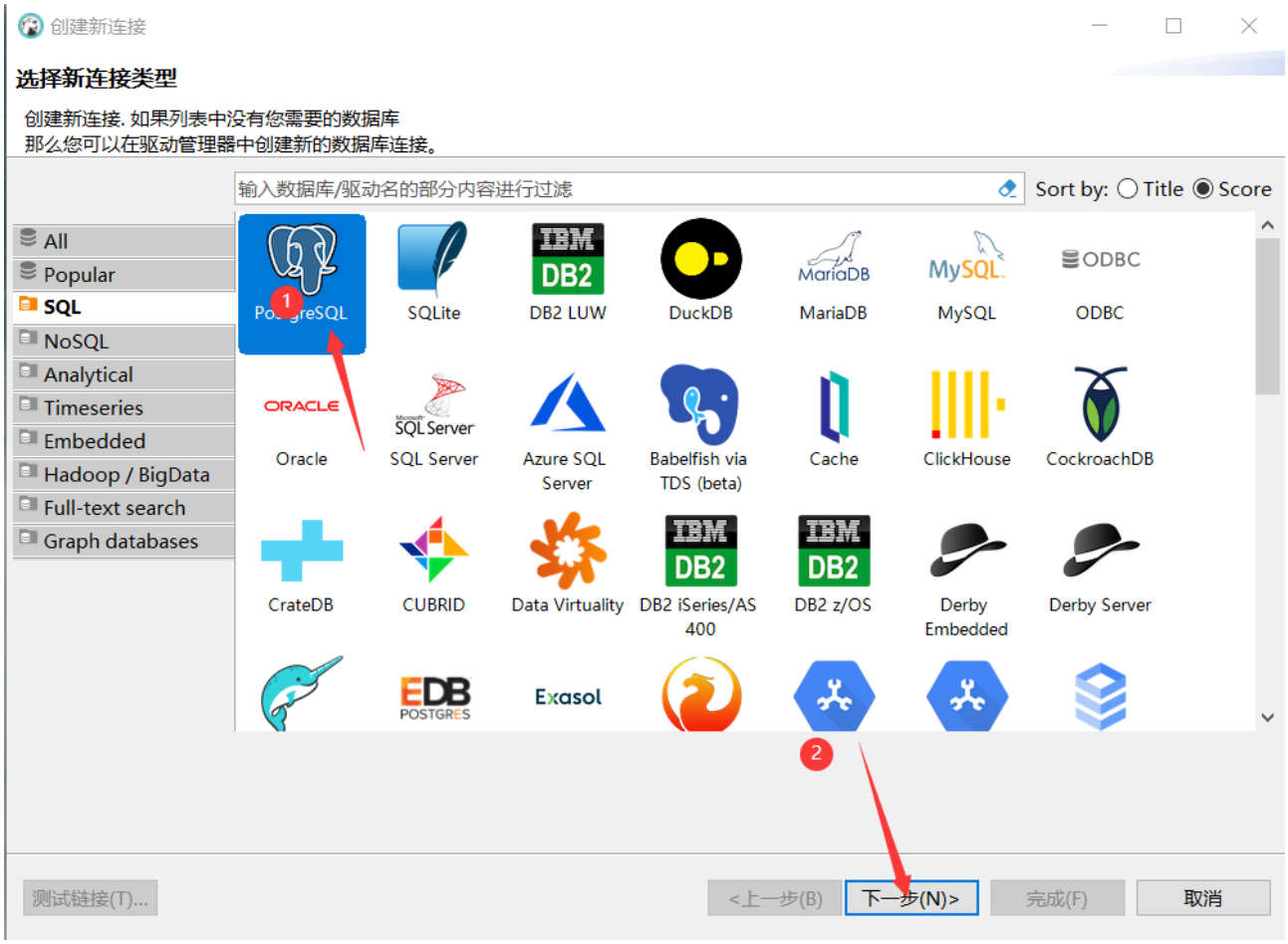

创建新连接
选择新连接类型
创建新连接, 如果列表中没有您需要的数据库
那么您可以在驱动管理器中创建新的数据库连接。
输入数据库/驱动名的部分内容进行过滤
Sort by: Title Score
All
Popular
SQL
NoSQL
Analytical
Timeseries
Embedded
Hadoop / BigData
Full-text search
Graph databases
PostgreSQL
SQLite
DB2 LUW
DuckDB
MariaDB
MySQL
ODBC
Oracle
SQL Server
Azure SQL Server
Babelfish via TDS (beta)
Cache
ClickHouse
CockroachDB
CrateDB
CUBRID
Data Virtuality
DB2 iSeries/AS 400
DB2 z/OS
Derby Embedded
Derby Server
EDB POSTGRES
Exasol
1
2
测试链接(T)...
<上一步(B)
下一步(N)>
完成(F)
取消

图 101: 图 2

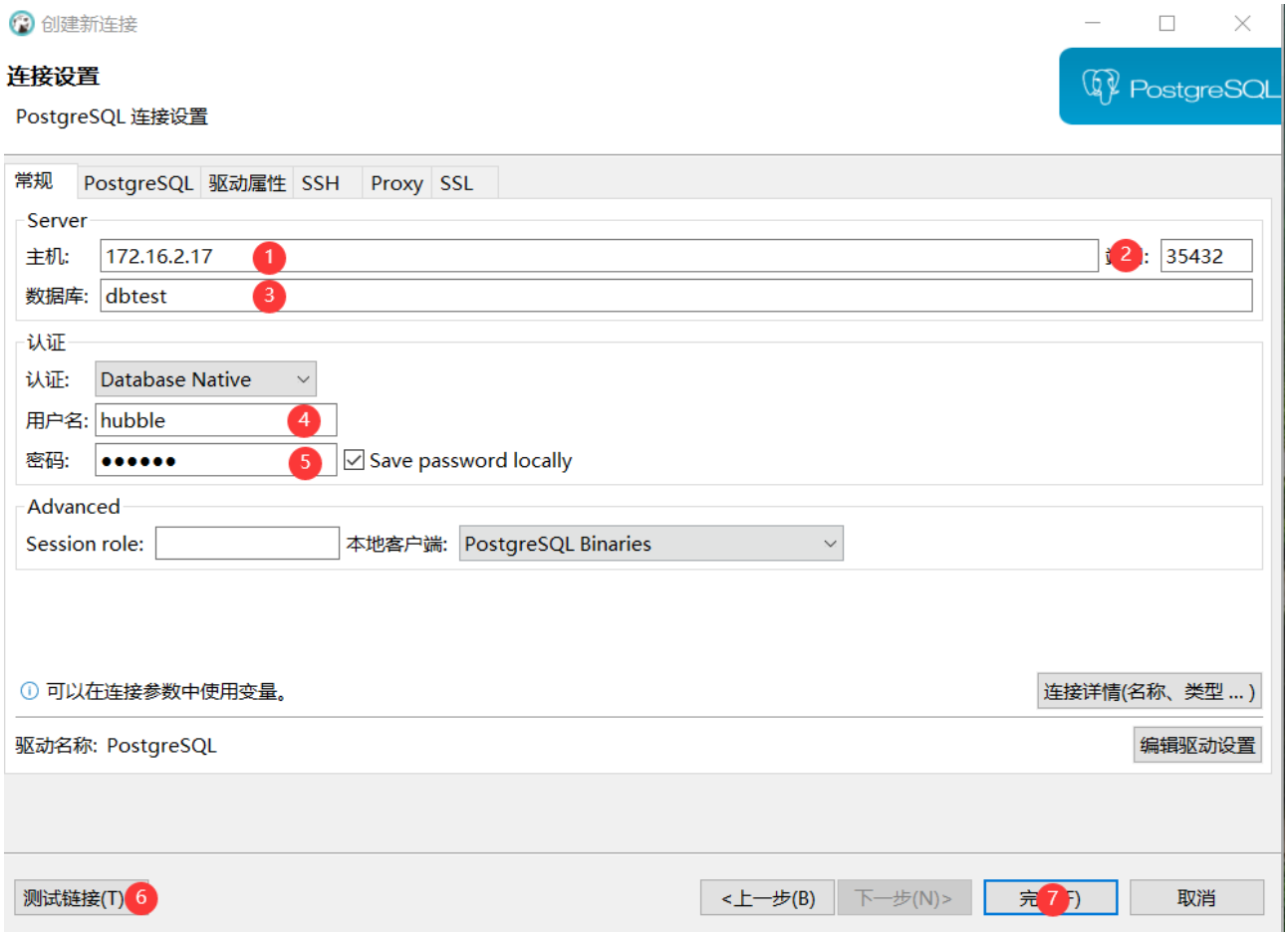

创建新连接
连接设置
PostgreSQL 连接设置
PostgreSQL
常规
PostgreSQL
驱动属性
SSH
Proxy
SSL
Server
主机:
172.16.2.17
1
2
35432
数据库:
dbtest
3
认证
认证:
Database Native
用户名:
hubble
4
密码:
5
Save password locally
Advanced
Session role:
本地客户端:
PostgreSQL Binaries
可以在连接参数中使用变量。
连接详情(名称、类型 ...)
驱动名称: PostgreSQL
编辑驱动设置
测试链接(T)
6
<上一步(B)
下一步(N)>
7
取消

图 102: 图 3

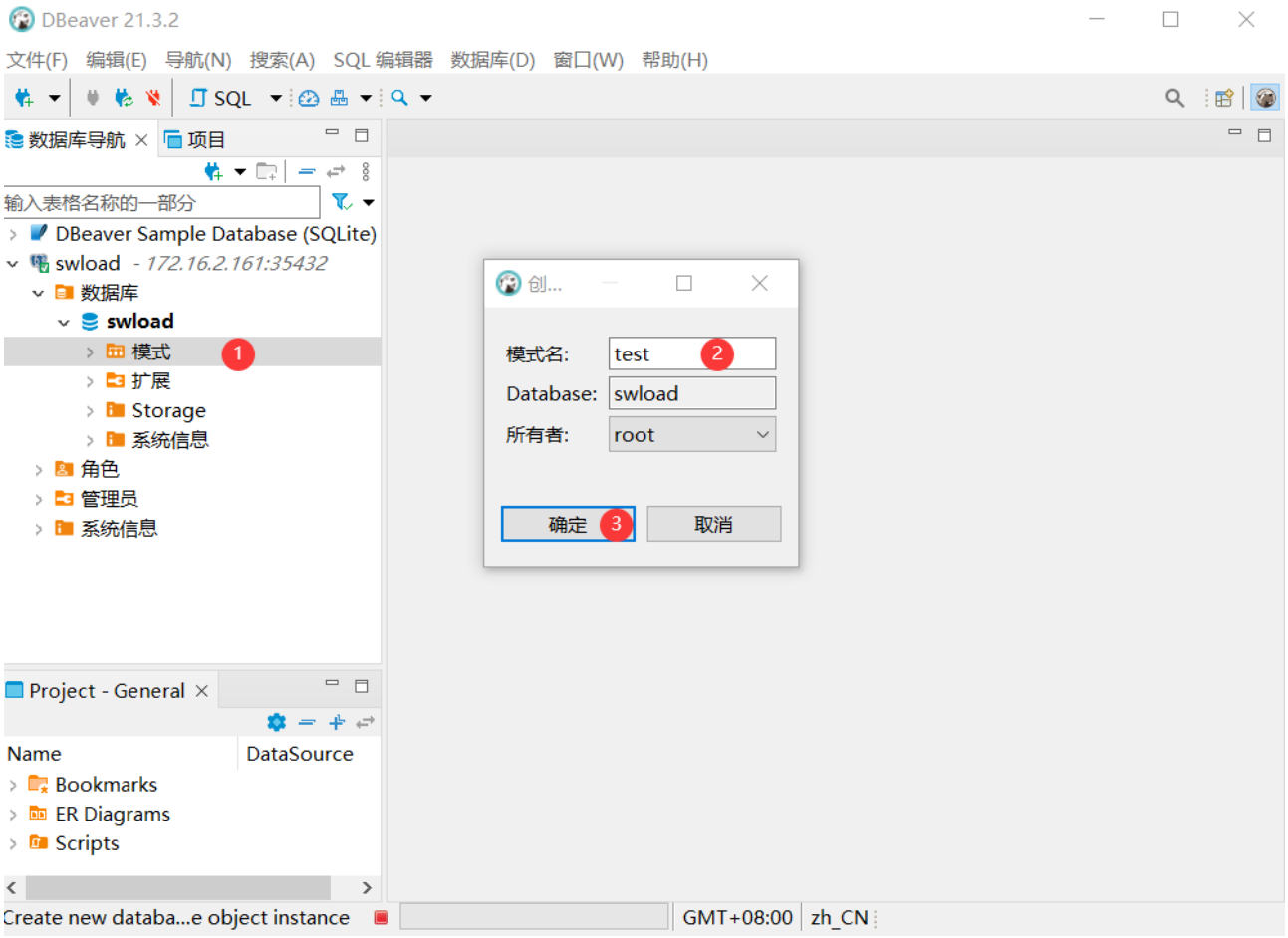

DBeaver 21.3.2
文件(F) 编辑(E) 导航(N) 搜索(A) SQL 编辑器 数据库(D) 窗口(W) 帮助(H)
SQL
数据库导航
项目
输入表格名称的一部分
DBeaver Sample Database (SQLite)
swload - 172.16.2.161:35432
数据库
swload
模式
扩展
Storage
系统信息
角色
管理员
系统信息
创...
模式名: test
Database: swload
所有者: root
确定
取消
Project - General
Name
DataSource
Bookmarks
ER Diagrams
Scripts
Create new databa...e object instance
GMT+08:00
zh_CN

图 103: 图 4

7.10.3.5 步骤五: 新建 SQL 编辑器

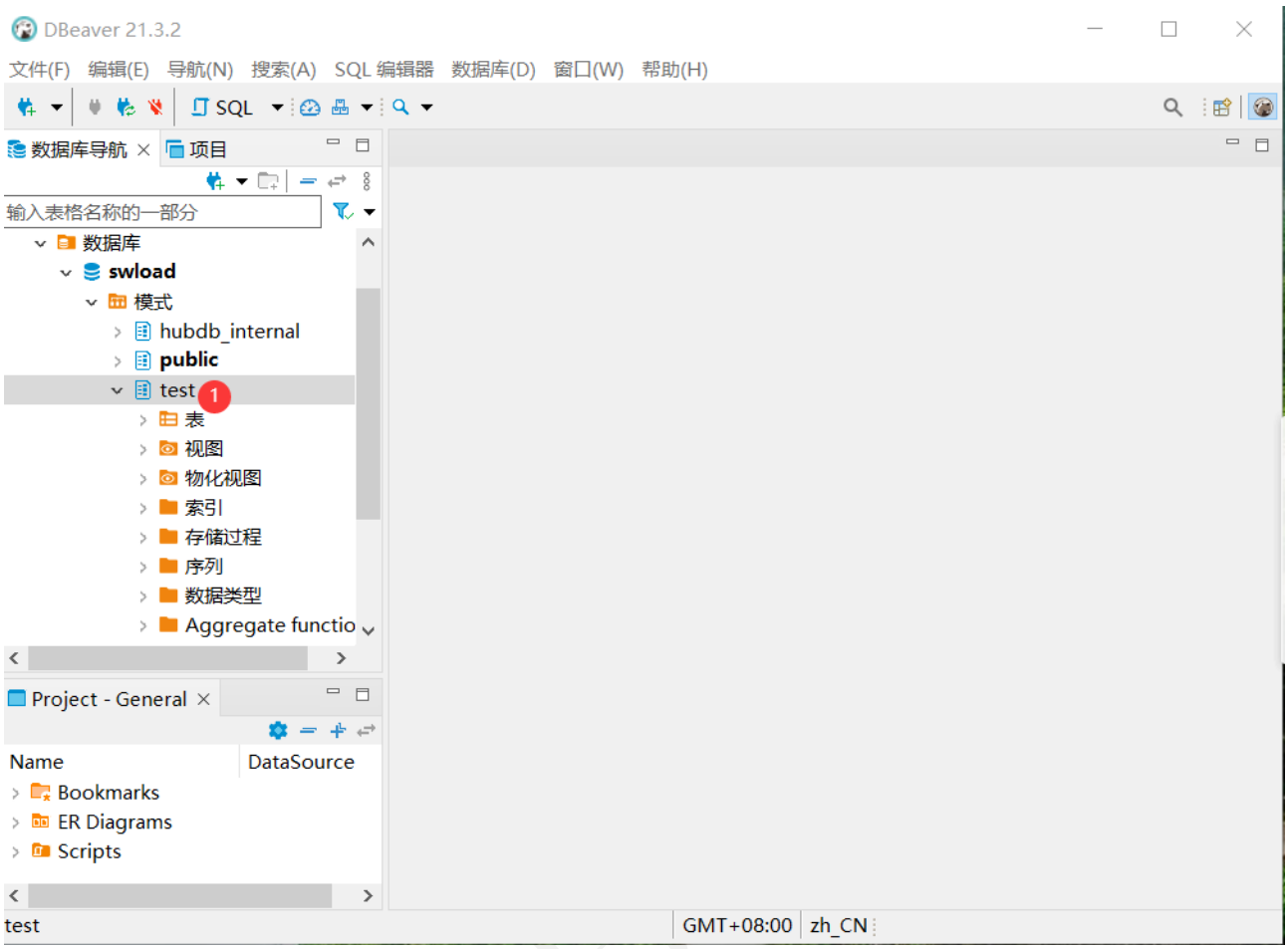


图 104: 图 5

sql 样例 (根据图片的标记，首先确认库名和模式)

8 论文

8.1 raft

寻找一种易于理解的一致性算法（扩展版）

英文论文地址

8.1.1 摘要

Raft 是一种为了管理复制日志的一致性算法。它提供了和 Paxos 算法相同的功能和性能，但是它的算法结构和 Paxos 不同，使得 Raft 算法更加容易理解并且更容易构建实际的系统。为了提升可理解性，Raft 将一致性算法分解成了几个关键模块，例如领导人选举、日志复制和安全性。同时它通过实施一个更强的一致性来减少需要考虑的状态的数量。从一个用户研究的结果可以证明，对于学生而言，Raft 算法比 Paxos 算法更加容易学习。Raft 算法还包括一个新的机制来允许集群成员的动态改变，它利用重叠的大多数来保证安全性。

8.1.2 介绍

一致性算法允许一组机器像一个整体一样工作，即使其中一些机器出现故障也能够继续工作下去。正因为如此，一致性算法在构建可信赖的大规模软件系统中扮演着重要的角色。在过去的 10 年里，Paxos 算法统治着一致性

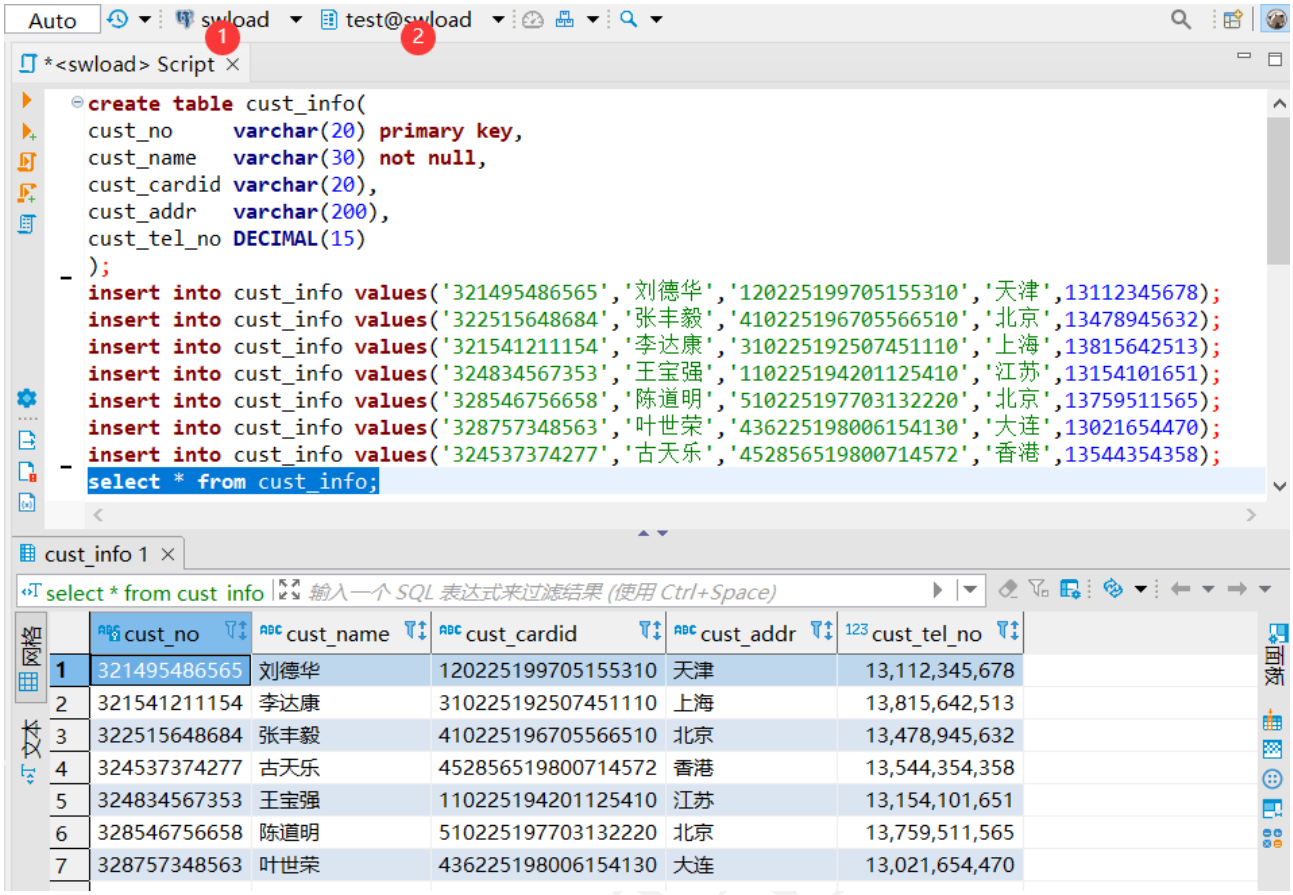


图 105: 图 6

算法这一领域：绝大多数的实现都是基于 Paxos 或者受其影响。同时 Paxos 也成为了教学领域里讲解一致性问题时的示例。

但是不幸的是，尽管有很多工作都在尝试降低它的复杂性，但是 Paxos 算法依然十分难以理解。并且，Paxos 自身的算法结构需要进行大幅的修改才能够应用到实际的系统中。这些都导致了工业界和学术界都对 Paxos 算法感到十分头疼。

和 Paxos 算法进行过努力之后，我们开始寻找一种新的一致性算法，可以为构建实际的系统和教学提供更好的基础。我们的做法是不寻常的，我们的首要目标是可理解性：我们是否可以在实际系统中定义一个一致性算法，并且能够比 Paxos 算法以一种更加容易的方式来学习。此外，我们希望该算法方便系统构建者的直觉的发展。不仅一个算法能够工作很重要，而且能够显而易见的知道为什么能工作也很重要。

Raft 一致性算法就是这些工作的结果。在设计 Raft 算法的时候，我们使用一些特别的技巧来提升它的可理解性，包括算法分解（Raft 主要被分成了领导人选举，日志复制和安全三个模块）和减少状态机的状态（相对于 Paxos，Raft 减少了非确定性和服务器互相处于非一致性的方式）。一份针对两所大学 43 个学生的研究表明 Raft 明显比 Paxos 算法更加容易理解。在这些学生同时学习了这两种算法之后，和 Paxos 比起来其中 33 个学生能够回答有关于 Raft 的问题。

Raft 算法在许多方面和现有的一致性算法都很相似（主要是 Oki 和 Liskov 的 Viewstamped Replication），但是它也有一些独特的特性：

- **强领导者**：和其他一致性算法相比，Raft 使用一种更强的领导能力形式。比如，日志条目只从领导者发送给其他的服务器。这种方式简化了对复制日志的管理并且使得 Raft 算法更加易于理解。
- **领导选举**：Raft 算法使用一个随机计时器来选举领导者。这种方式只是在任何一致性算法都必须实现的心跳机制上增加了一点机制。在解决冲突的时候会更加简单快捷。

- **成员关系调整**：Raft 使用一种共同一致的方法来处理集群成员变换的问题，在这种方法下，处于调整过程中的两种不同的配置集群中大多数机器会有重叠，这就使得集群在成员变换的时候依然可以继续工作。

我们相信，Raft 算法不论出于教学目的还是作为实践项目的基础都是要比 Paxos 或者其他一致性算法要优异的。它比其他算法更加简单，更加容易理解；它的算法描述足以实现一个现实的系统；它有好多开源的实现并且在很多公司里使用；它的安全性已经被证明；它的效率和其他算法比起来也不相上下。

接下来，这篇论文会介绍以下内容：复制状态机问题（第 2 节），讨论 Paxos 的优点和缺点（第 3 节），讨论我们为了可理解性而采取的方法（第 4 节），阐述 Raft 一致性算法（第 5-8 节），评价 Raft 算法（第 9 节），以及一些相关的工作（第 10 节）。

8.1.3 复制状态机

一致性算法是从复制状态机的背景下提出的（参考英文原文引用 37）。在这种方法中，一组服务器上的状态机产生相同状态的副本，并且在一些机器宕掉的情况下也可以继续运行。复制状态机在分布式系统中被用于解决很多容错的问题。例如，大规模的系统中通常都有一个集群领导者，像 GFS、HDFS 和 RAMCloud，典型应用就是一个独立的的复制状态机去管理领导选举和存储配置信息并且在领导人宕机的情况下也要存活下来。比如 Chubby 和 ZooKeeper。

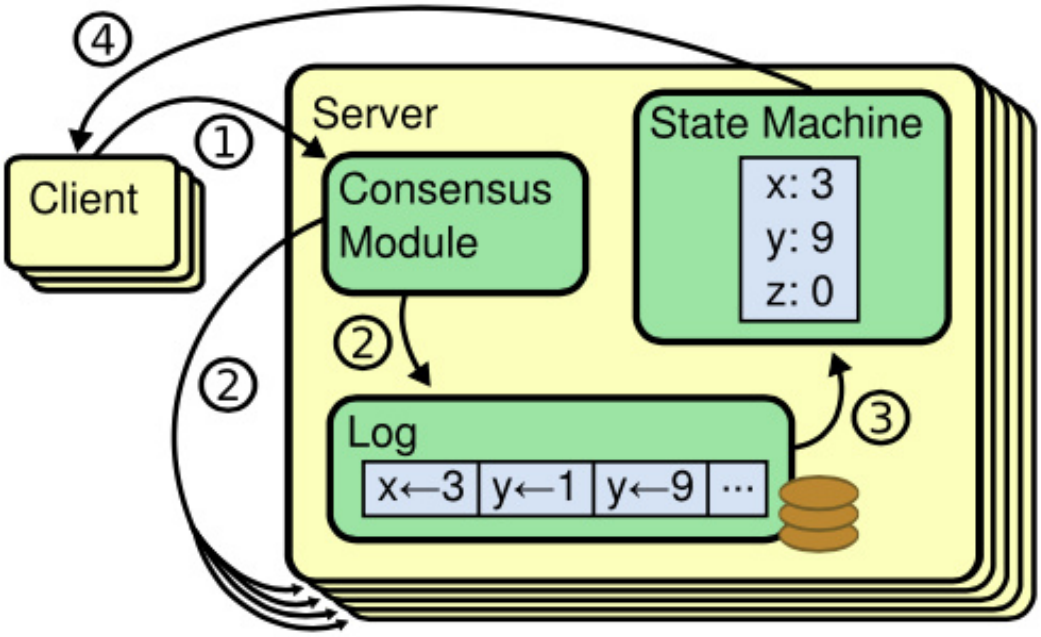


图 106: 图 1

图 1：复制状态机的结构。一致性算法管理着来自客户端指令的复制日志。状态机从日志中处理相同顺序的相同指令，所以产生的结果也是相同的。

复制状态机通常都是基于复制日志实现的，如图 1。每一个服务器存储一个包含一系列指令的日志，并且按照日志的顺序进行执行。每一个日志都按照相同的顺序包含相同的指令，所以每一个服务器都执行相同的指令序列。因为每个状态机都是确定的，每一次执行操作都产生相同的状态和同样的序列。

保证复制日志相同就是一致性算法的工作了。在一台服务器上，一致性模块接收客户端发送来的指令然后增加到自己的日志中去。它和其他服务器上的一致性模块进行通信来保证每一个服务器上的日志最终都以相同的顺序包含相同的请求，尽管有些服务器会宕机。一旦指令被正确的复制，每一个服务器的状态机按照日志顺序处理他们，然后输出结果被返回给客户端。因此，服务器集群看起来形成一个高可靠的状态机。

实际系统中使用的一致性算法通常含有以下特性：

- 安全性保证（绝对不会返回一个错误的结果）：在非拜占庭错误情况下，包括网络延迟、分区、丢包、冗余和乱序等错误都可以保证正确。
- 可用性：集群中只要有大多数的机器可运行并且能够相互通信、和客户端通信，就可以保证可用。因此，一个典型的包含 5 个节点的集群可以容忍两个节点的失败。服务器被停止就认为是失败。他们当有稳定的存储的时候可以从状态中恢复回来并重新加入集群。
- 不依赖时序来保证一致性：物理时钟错误或者极端的消息延迟只有在最坏情况下才会导致可用性问题。
- 通常情况下，一条指令可以尽可能快的在集群中大多数节点响应一轮远程过程调用时完成。小部分比较慢的节点不会影响系统整体的性能。

8.1.4 Paxos 算法的问题

在过去的 10 年里，Leslie Lamport 的 Paxos 算法几乎已经成为一致性的代名词：Paxos 是在课程教学中最经常使用的算法，同时也是大多数一致性算法实现的起点。Paxos 首先定义了一个能够达成单一决策一致的协议，比如单条的复制日志项。我们把这一子集叫做单决策 Paxos。然后通过组合多个 Paxos 协议的实例来促进一系列决策的达成。Paxos 保证安全性和活性，同时也支持集群成员关系的变更。Paxos 的正确性已经被证明，在通常情况下也很高效。

不幸的是，Paxos 有两个明显的缺点。第一个缺点是 Paxos 算法特别的难以理解。完整的解释是出了名的不透明；通过极大的努力之后，也只有少数人成功理解了这个算法。因此，有了几次用更简单的术语来解释 Paxos 的尝试。尽管这些解释都只关注了单决策的子集问题，但依然很具有挑战性。在 2012 年 NSDI 的会议中的一次调查显示，很少有人对 Paxos 算法感到满意，甚至在经验老道的研究者中也是如此。我们自己也尝试去理解 Paxos；我们一直没能理解 Paxos 直到我们读了很多对 Paxos 的简化解释并且设计了我们自己的算法之后，这一过程花了近一年时间。

我们假设 Paxos 的不透明性来自它选择单决策问题作为它的基础。单决策 Paxos 是晦涩微妙的，它被划分成了两种没有简单直观解释和无法独立理解的情景。因此，这导致了很难建立起直观的感受为什么单决策 Paxos 算法能够工作。构成多决策 Paxos 增加了很多错综复杂的规则。我们相信，在多决策上达成一致性的问题（一份日志而不是单一的日志记录）能够被分解成其他的方式并且更加直接和明显。

Paxos 算法的第二个问题就是它没有提供一个足够好的用来构建一个现实系统的基础。一个原因是还没有一种被广泛认同的多决策问题的算法。Lamport 的描述基本上都是关于单决策 Paxos 的；他简要描述了实施多决策 Paxos 的方法，但是缺乏很多细节。当然也有很多具体化 Paxos 的尝试，但是他们都互相不一样，和 Paxos 的概述也不同。例如 Chubby 这样的系统实现了一个类似于 Paxos 的算法，但是大多数的细节并没有被公开。

而且，Paxos 算法的结构也不是十分易于构建实践的系统；单决策分解也会产生其他的结果。例如，独立的选择一组日志条目然后合并成一个序列化的日志并没有带来太多的好处，仅仅增加了不少复杂性。围绕着日志来设计一个系统是更加简单高效的；新日志条目以严格限制的顺序增添到日志中去。另一个问题是，Paxos 使用了一种对等的点对点的方式作为它的核心（尽管它最终提议了一种弱领导人的方法来优化性能）。在只有一个决策会被制定的简化世界中是很有意义的，但是很少有现实的系统使用这种方式。如果有一系列的决策需要被制定，首先选择一个领导人，然后让他去协调所有的决议，会更加简单快速。

因此，实际的系统中很少有和 Paxos 相似的实践。每一种实现都是从 Paxos 开始研究，然后发现很多实现上的难题，再然后开发了一种和 Paxos 明显不一样的结构。这样是非常费时和容易出错的，并且理解 Paxos 的难度使得这个问题更加糟糕。Paxos 算法在理论上被证明是正确可行的，但是现实的系统和 Paxos 差别是如此的大，以至于这些证明没有什么太大的价值。下面来自 Chubby 实现非常典型：

> 在 Paxos 算法描述和实现现实系统中间有着巨大的鸿沟。最终的系统建立在一种没有经过证明的算法之上。

由于以上问题，我们认为 Paxos 算法既没有提供一个良好的基础给实践的系统，也没有给教学很好的帮助。基于一致性问题在大规模软件系统中的重要性，我们决定看看我们是否可以设计一个拥有更好特性的替代 Paxos 的一致性算法。Raft 算法就是这次实验的结果。

8.1.5 为了可理解性的设计

设计 Raft 算法我们有几个初衷：它必须提供一个完整的实际的系统实现基础，这样才能大大减少开发者的工作；它必须在任何情况下都是安全的并且在大多数的情况下都是可用的；并且它的大部分操作必须是高效的。但是我们最重要也是最大的挑战是可理解性。它必须保证对于普遍的人群都可以十分容易的去理解。另外，它必须能够让人形成直观的认识，这样系统的构建者才能够在现实中进行必然的扩展。

在设计 Raft 算法的时候，有很多的点需要我们在各种备选方案中进行选择。在这种情况下，我们评估备选方案基于可理解性原则：解释各个备选方案有多大的难度（例如，Raft 的状态空间有多复杂，是否有微妙的暗示）？对于一个读者而言，完全理解这个方案和暗示是否容易？

我们意识到对这种可理解性分析上具有高度的主观性；尽管如此，我们使用了两种通常适用的技术来解决这个问题。第一个技术就是众所周知的问题分解：只要有可能，我们就将问题分解成几个相对独立的，可被解决的、可解释的和可理解的子问题。例如，Raft 算法被我们分成领导人选举，日志复制，安全性和角色改变几个部分。

我们使用的第二个方法是通过减少状态的数量来简化需要考虑的状态空间，使得系统更加连贯并且在可能的时候消除不确定性。特别的，所有的日志是不允许有空洞的，并且 Raft 限制了日志之间变成不一致状态的可能。尽管在大多数情况下我们都试图去消除不确定性，但是也有一些情况下不确定性可以提升可理解性。尤其是，随机化方法增加了不确定性，但是他们有利于减少状态空间数量，通过处理所有可能选择时使用相似的方法。我们使用随机化去简化 Raft 中领导人选举算法。

8.1.6 Raft 一致性算法

Raft 是一种用来管理章节 2 中描述的复制日志的算法。图 2 为了参考之用，总结这个算法的简略版本，图 3 列举了这个算法的一些关键特性。图中的这些元素会在剩下的章节逐一介绍。

Raft 通过选举一个高贵的领导人，然后给予他全部的管理复制日志的责任来实现一致性。领导人从客户端接收日志条目，把日志条目复制到其他服务器上，并且当保证安全性的时候告诉其他的服务器应用日志条目到他们的状态机中。拥有一个领导人大大简化了对复制日志的管理。例如，领导人可以决定新的日志条目需要放在日志中的什么位置而不需要和其他服务器商议，并且数据都从领导人流向其他服务器。一个领导人可以宕机，可以和其他服务器失去连接，这时一个新的领导人会被选举出来。

通过领导人的方式，Raft 将一致性问题分解成了三个相对独立的子问题，这些问题会在接下来的子章节中进行讨论：

- **领导选举**：一个新的领导人需要被选举出来，当现存的领导人宕机的时候（章节 5.2）
- **日志复制**：领导人必须从客户端接收日志然后复制到集群中的其他节点，并且强制要求其他节点的日志保持和自己相同。
- **安全性**：在 Raft 中安全性的关键是在图 3 中展示的状态机安全：如果有任何的服务器节点已经应用了一个确定的日志条目到它的状态机中，那么其他服务器节点不能在同一个日志索引位置应用一个不同的指令。章节 5.4 阐述了 Raft 算法是如何保证这个特性的；这个解决方案涉及到一个额外的选举机制（5.2 节）上的限制。

在展示一致性算法之后，这一章节会讨论可用性的一些问题和计时在系统的作用。

状态：

状态	所有服务器上持久存在的
currentTerm	服务器最后一次知道的任期号（初始化为 0，持续递增）
votedFor	在当前获得选票的候选人的 Id
log[]	日志条目集；每一个条目包含一个用户状态机执行的指令，和收到时的任期号

状态	所有服务器上经常变的
commitIndex	已知的最大的已经被提交的日志条目的索引值
lastApplied	最后被应用到状态机的日志条目索引值（初始化为 0，持续递增）

状态	在领导人里经常改变的（选举后重新初始化）
nextIndex[]	对于每一个服务器，需要发送给他的下一个日志条目的索引值（初始化为领导人最后索引值加一）
matchIndex[]	对于每一个服务器，已经复制给他的日志的最高索引值

附加日志 RPC：

由领导人负责调用来复制日志指令；也会用作 heartbeat

参数	解释
term	领导人的任期号
leaderId	领导人的 Id，以便于跟随者重定向请求
prevLogIndex	新的日志条目紧随之前的索引值
prevLogTerm	prevLogIndex 条目的任期号
entries[]	准备存储的日志条目（表示心跳时为空；一次性发送多个是为了提高效率）
leaderCommit	领导人已经提交的日志的索引值

返回值	解释
term	当前的任期号，用于领导人去更新自己
success	跟随者包含了匹配上 prevLogIndex 和 prevLogTerm 的日志时为真

接收者实现：

1. 如果 `term < currentTerm` 就返回 false （5.1 节）
2. 如果日志在 prevLogIndex 位置处的日志条目的任期号和 prevLogTerm 不匹配，则返回 false （5.3 节）
3. 如果已经存在的日志条目和新的产生冲突（索引值相同但是任期号不同），删除这一条和之后所有的（5.3 节）
4. 附加日志中尚未存在的任何新条目

HUBBLE

5. 如果 `leaderCommit > commitIndex`，令 commitIndex 等于 leaderCommit 和新日志条目索引值中较小的一个

请求投票 RPC：

由候选人负责调用用来征集选票（5.2 节）

参数	解释
term	候选人的任期号
candidateId	请求选票的候选人的 Id
lastLogIndex	候选人的最后日志条目的索引值
lastLogTerm	候选人最后日志条目的任期号

返回值	解释
term	当前任期号，以便于候选人去更新自己的任期号
voteGranted	候选人赢得了此张选票时为真

接收者实现：

1. 如果`term < currentTerm`返回 false （5.2 节）
2. 如果 votedFor 为空或者为 candidateId，并且候选人的日志至少和自己一样新，那么就投票给他（5.2 节，5.4 节）

所有服务器需遵守的规则：

所有服务器：

- 如果`commitIndex > lastApplied`，那么就 lastApplied 加一，并把`log[lastApplied]`应用到状态机中（5.3 节）
- 如果接收到的 RPC 请求或响应中，任期号`T > currentTerm`，那么就令 currentTerm 等于 T，并切换状态为跟随者（5.1 节）

跟随者（5.2 节）：

- 响应来自候选人和领导者的请求
- 如果在超过选举超时时间的情况之前都没有收到领导人的心跳，或者是候选人请求投票的，就自己变成候选人

候选人（5.2 节）：

- 在转变成候选人后就立即开始选举过程
  - 自增当前的任期号（currentTerm）
  - 给自己投票
  - 重置选举超时计时器
  - 发送请求投票的 RPC 给其他所有服务器
- 如果接收到大多数服务器的选票，那么就变成领导人
- 如果接收到来自新的领导人的附加日志 RPC，转变成跟随者
- 如果选举过程超时，再次发起一轮选举

领导人：

HUBBLE

- 一旦成为领导人：发送空的附加日志 RPC（心跳）给其他所有的服务器；在一定的空余时间之后不停的重复发送，以阻止跟随者超时（5.2 节）
- 如果接收到来自客户端的请求：附加条目到本地日志中，在条目被应用到状态机后响应客户端（5.3 节）
- 如果对于一个跟随者，最后日志条目的索引值大于等于 nextIndex，那么：发送从 nextIndex 开始的所有日志条目：
  - 如果成功：更新相应跟随者的 nextIndex 和 matchIndex
  - 如果因为日志不一致而失败，减少 nextIndex 重试
- 如果存在一个满足`N > commitIndex`的 N，并且大多数的`matchIndex[i] N`成立，并且`log[N].term == currentTerm`成立，那么令 commitIndex 等于这个 N （5.3 和 5.4 节）

图 2：一个关于 Raft 一致性算法的浓缩总结（不包括成员变换和日志压缩）。

特性	解释
选举安全特性	对于一个给定的任期号，最多只会有一个领导人被选举出来（5.2 节）
领导人只附加原则	领导人绝对不会删除或者覆盖自己的日志，只会增加（5.3 节）
日志匹配原则	如果两个日志在相同的索引位置的日志条目的任期号相同，那么我们就认为这个日志从头到这个索引位置之间全部完全相同（5.3 节）
领导人完全特性	如果某个日志条目在某个任期号中已经被提交，那么这个条目必然出现在更大任期号的所有领导人中（5.4 节）
状态机安全特性	如果一个领导人已经在给定的索引值位置的日志条目应用到状态机中，那么其他任何的服务器在这个索引位置不会提交一个不同的日志（5.4.3 节）

图 3：Raft 在任何时候都保证以上的各个特性。

8.1.6.1 Raft 基础

一个 Raft 集群包含若干个服务器节点；通常是 5 个，这允许整个系统容忍 2 个节点的失效。在任何时刻，每一个服务器节点都处于这三个状态之一：领导人、跟随者或者候选人。在通常情况下，系统中只有一个领导人并且其他的节点全部都是跟随者。跟随者都是被动的：他们不会发送任何请求，只是简单的响应来自领导者或者候选人的请求。领导人处理所有的客户端请求（如果一个客户端和跟随者联系，那么跟随者会把请求重定向给领导人）。第三种状态，候选人，是用来在 5.2 节描述的选举新领导人时使用。图 4 展示了这些状态和他们之间的转换关系；这些转换关系会在接下来进行讨论。

图 4：服务器状态。跟随者只响应来自其他服务器的请求。如果跟随者接收不到消息，那么他就会变成候选人并发起一次选举。获得集群中大多数选票的候选人将成为领导者。在一个任期内，领导人一直都会是领导人直到自己宕机了。

图 5：时间被划分成一个个的任期，每个任期开始都是一次选举。在选举成功后，领导人会管理

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

图 107: 图 2

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

图 108: 图 3

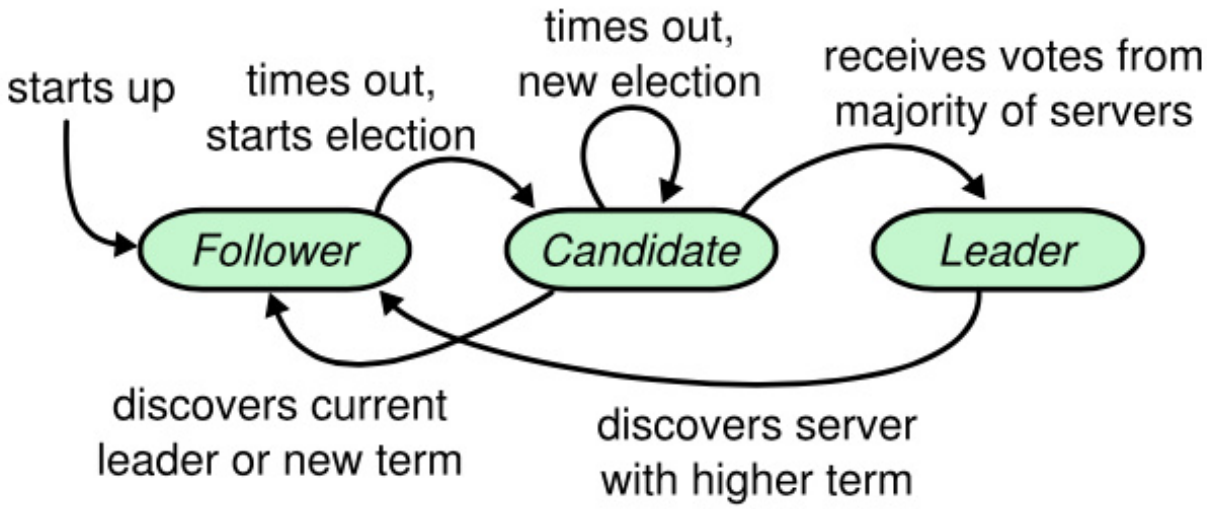


图 109: 图 4

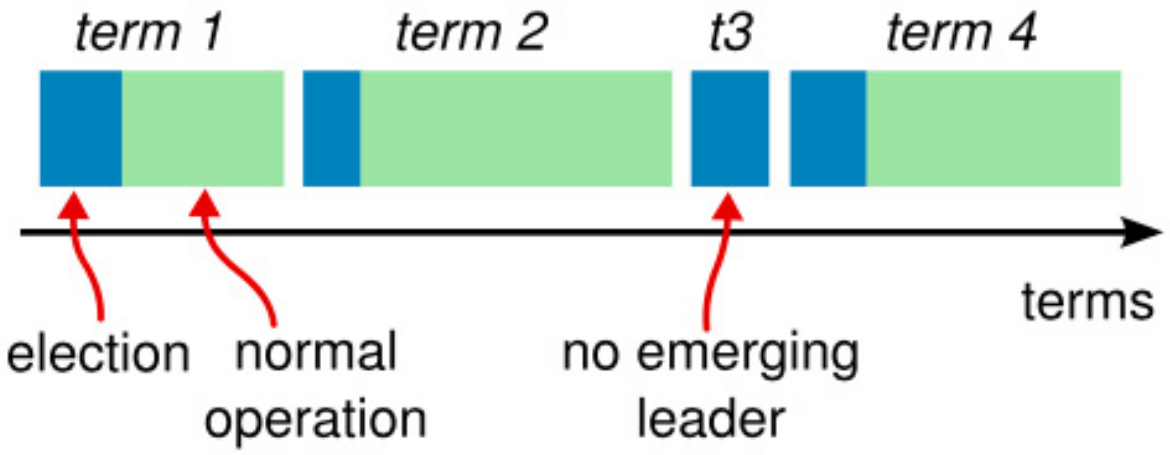


图 110: 图 5

> 整个集群直到任期结束。有时候选举会失败，那么这个任期就会没有领导人而结束。任期之间的切换可以在不同的时间不同的服务器上观察到。

Raft 把时间分割成任意长度的**任期**，如图 5。任期用连续的整数标记。每一段任期从一次**选举**开始，就像章节 5.2 描述的一样，一个或者多个候选人尝试成为领导者。如果一个候选人赢得选举，然后他就在接下来的任期内充当领导人的职责。在某些情况下，一次选举过程会造成选票的瓜分。在这种情况下，这一任期会以没有领导人结束；一个新的任期（和一次新的选举）会很快重新开始。Raft 保证了在一个给定的任期内，最多只有一个领导者。

不同的服务器节点可能多次观察到任期之间的转换，但在某些情况下，一个节点也可能观察不到任何一次选举或者整个任期全程。任期在 Raft 算法中充当逻辑时钟的作用，这会允许服务器节点查明一些过期的信息比如陈旧的领导者。每一个节点存储一个当前任期号，这一编号在整个时期内单调的增长。当服务器之间通信的时候会交换当前任期号；如果一个服务器的当前任期号比其他人小，那么他会更新自己的编号到较大的编号值。如果一个候选人或者领导者发现自己的任期号过期了，那么他会立即恢复成跟随者状态。如果一个节点接收到一个包含过期的任期号的请求，那么他会直接拒绝这个请求。

Raft 算法中服务器节点之间通信使用远程过程调用（RPCs），并且基本的一致性算法只需要两种类型的 RPCs。请求投票（RequestVote）RPCs 由候选人在选举期间发起（章节 5.2），然后附加条目（AppendEntries）RPCs 由领导人发起，用来复制日志和提供一种心跳机制（章节 5.3）。第 7 节为了在服务器之间传输快照增加了第三种 RPC。当服务器没有及时的收到 RPC 的响应时，会进行重试，并且他们能够并行的发起 RPCs 来获得最佳的性能。

8.1.6.2 领导人选举

Raft 使用一种心跳机制来触发领导人选举。当服务器程序启动时，他们都是跟随者身份。一个服务器节点继续保持着跟随者状态只要他从领导人或者候选者处接收到有效的 RPCs。领导者周期性的向所有跟随者发送心跳包（即不包含日志项内容的附加日志项 RPCs）来维持自己的权威。如果一个跟随者在一段时间里没有接收到任何消息，也就是**选举超时**，那么他就会认为系统中没有可用的领导者，并且发起选举以选出新的领导者。

要开始一次选举过程，跟随者先要增加自己的当前任期号并且转换到候选人状态。然后他会并行的向集群中的其他服务器节点发送请求投票的 RPCs 来给自己投票。候选人会继续保持着当前状态直到以下三件事情之一发生：(a) 他自己赢得了这次的选举，(b) 其他的服务器成为领导者，(c) 一段时间之后没有任何一个获胜的人。这些结果会分别的在下面的段落里进行讨论。

当一个候选人从整个集群的大多数服务器节点获得了针对同一个任期号的选票，那么他就赢得了这次选举并成为领导人。每一个服务器最多会对一个任期号投出一张选票，按照先来先服务的原则（注意：5.4 节在投票上增加了一点额外的限制）。要求大多数选票的规则确保了最多只会有一个候选人赢得此次选举（图 3 中的选举安全性）。一旦候选人赢得选举，他就立即成为领导人。然后他会向其他的服务器发送心跳消息来建立自己的权威并且阻止新的领导人的产生。

在等待投票的时候，候选人可能会从其他的服务器接收到声明它是领导人的附加日志项 RPC。如果这个领导人的任期号（包含在此次的 RPC 中）不小于候选人当前的任期号，那么候选人会承认领导人合法并回到跟随者状态。如果此次 RPC 中的任期号比自己小，那么候选人就会拒绝这次的 RPC 并且继续保持候选人状态。

第三种可能的结果是候选人既没有赢得选举也没有输：如果有多个跟随者同时成为候选人，那么选票可能会被瓜分以至于没有候选人可以赢得大多数人的支持。当这种情况发生的时候，每一个候选人都会超时，然后通过增加当前任期号来开始一轮新的选举。然而，没有其他机制的话，选票可能会被无限的重复瓜分。

Raft 算法使用随机选举超时时间的方法来确保很少会发生选票瓜分的情况，就算发生也能很快的解决。为了阻止选票起初就被瓜分，选举超时时间是从一个固定的区间（例如 150-300 毫秒）随机选择。这样可以把服务器都分散开以至于在大多数情况下只有一个服务器会选举超时；然后他赢得选举并在其他服务器超时之前发送心跳包。同样的机制被用在选票瓜分的情况下。每一个候选人在开始一次选举的时候会重置一个随机的选举超时时间，然后在超时时间内等待投票的结果；这样减少了在新的选举中另外的选票瓜分的可能性。9.3 节展示了这种方案能够快速的选出一个领导人。

领导人选举这个例子，体现了可理解性原则是如何指导我们进行方案设计的。起初我们计划使用一种排名系统：每一个候选人都被赋予一个唯一的排名，供候选人之间竞争时进行选择。如果一个候选人发现另一个候选人拥有更高的排名，那么他就会回到跟随者状态，这样高排名的候选人能够更加容易的赢得下一次选举。但是我们发现这种方法在可用性方面会有一点问题（如果高排名的服务器宕机了，那么低排名的服务器可能会超时并再次进入候选人状态。而且如果这个行为发生得足够快，则可能会导致整个选举过程都被重置掉）。我们针对算法进行了多次调整，但是每次调整之后都会有新的问题。最终我们认为随机重试的方法是更加明显和易于理解的。

8.1.6.3 日志复制

一旦一个领导人被选举出来，他就开始为客户端提供服务。客户端的每一个请求都包含一条被复制状态机执行的指令。领导人把这条指令作为一条新的日志条目附加到日志中去，然后并行的发起附加条目 RPCs 给其他的服务器，让他们复制这条日志条目。当这条日志条目被安全的复制（下面会介绍），领导人会应用这条日志条目到它的状态机中然后把执行的结果返回给客户端。如果跟随者崩溃或者运行缓慢，再或者网络丢包，领导人会不断的重复尝试附加日志条目 RPCs （尽管已经回复了客户端）直到所有的跟随者都最终存储了所有的日志条目。

> 图 6：日志由有序序号标记的条目组成。每个条目都包含创建时的任期号（图中框中的数字），和一个状态机需要执行的指令。一个条目当可以安全的被应用到状态机中去的时候，就认为是可以提交了。

日志以图 6 展示的方式组织。每一个日志条目存储一条状态机指令和从领导人收到这条指令时的任期号。日志中的任期号用来检查是否出现不一致的情况，同时也用来保证图 3 中的某些性质。每一条日志条目同时也都有一个整数索引值来表明它在日志中的位置。

领导人来决定什么时候把日志条目应用到状态机中是安全的；这种日志条目被称为**已提交**。Raft 算法保证所有已提交的日志条目都是持久化的并且最终会被所有可用的状态机执行。在领导人将创建的日志条目复制到大多数的服务器上的时候，日志条目就会被提交（例如在图 6 中的条目 7）。同时，领导人的日志中之前的所有日志条目也都会被提交，包括由其他领导人创建的条目。5.4 节会讨论某些当在领导人改变之后应用这条规则的隐晦内容，同时他也展示了这种提交的定义是安全的。领导人跟踪了最大的将会被提交的日志项的索引，并且索引

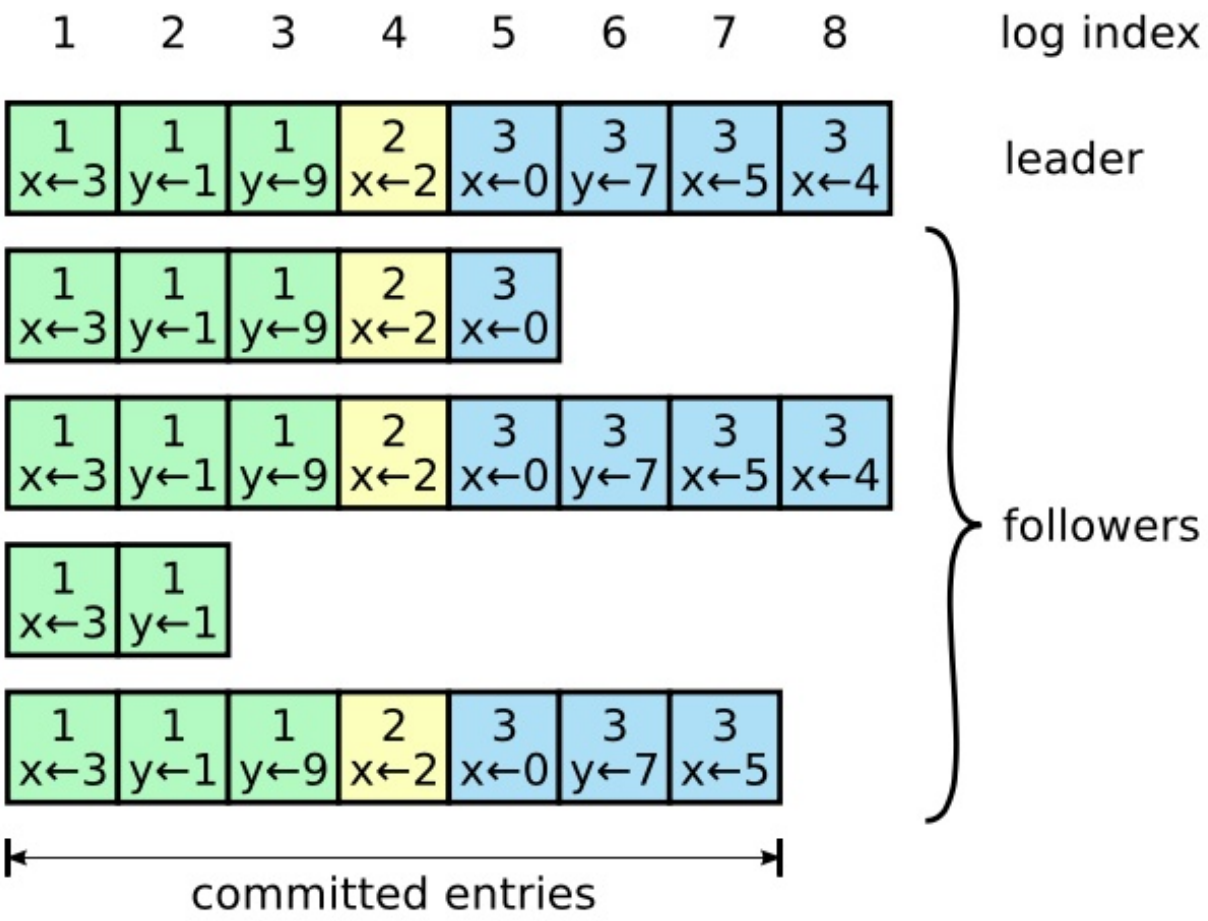

1 2 3 4 5 6 7 8
log index
1 x←3
1 y←1
1 y←9
2 x←2
3 x←0
3 y←7
3 x←5
3 x←4
leader
followers
committed entries

图 111: 图 6

值会被包含在未来的所有附加日志 RPCs （包括心跳包），这样其他的服务器才能最终知道领导人的提交位置。一旦跟随者知道一条日志条目已经被提交，那么他也会将这个日志条目应用到本地的状态机中（按照日志的顺序）。

我们设计了 Raft 的日志机制来维护一个不同服务器的日志之间的高层次的一致性。这么做不仅简化了系统的行为也使得更加可预计，同时他也是安全性保证的一个重要组件。Raft 维护着以下的特性，这些同时也组成了图 3 中的日志匹配特性：

- 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们存储了相同的指令。
- 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们之前的所有日志条目也全部相同。

第一个特性来自这样的一个事实，领导人最多在一个任期里在指定的一个日志索引位置创建一条日志条目，同时日志条目在日志中的位置也从来不会改变。第二个特性由附加日志 RPC 的一个简单的一致性检查所保证。在发送附加日志 RPC 的时候，领导人会把新的日志条目紧接着之前的条目的索引位置和任期号包含在里面。如果跟随者在它的日志中找不到包含相同索引位置和任期号的条目，那么他就会拒绝接收新的日志条目。一致性检查就像一个归纳步骤：一开始空的日志状态肯定是满足日志匹配特性的，然后一致性检查保护了日志匹配特性当日志扩展的时候。因此，每当附加日志 RPC 返回成功时，领导人就知道跟随者的日志一定是和自己相同的了。

在正常的操作中，领导人和跟随者的日志保持一致性，所以附加日志 RPC 的一致性检查从来不会失败。然而，领导人崩溃的情况会使得日志处于不一致的状态（老的领导人可能还没有完全复制所有的日志条目）。这种不一致问题会在领导人和跟随者的一系列崩溃下加剧。图 7 展示了跟随者的日志可能和新的领导人不同的方式。跟随者可能会丢失一些在新的领导人中有的日志条目，他也可能拥有一些领导人没有的日志条目，或者两者都发生。丢失或者多出日志条目可能会持续多个任期。

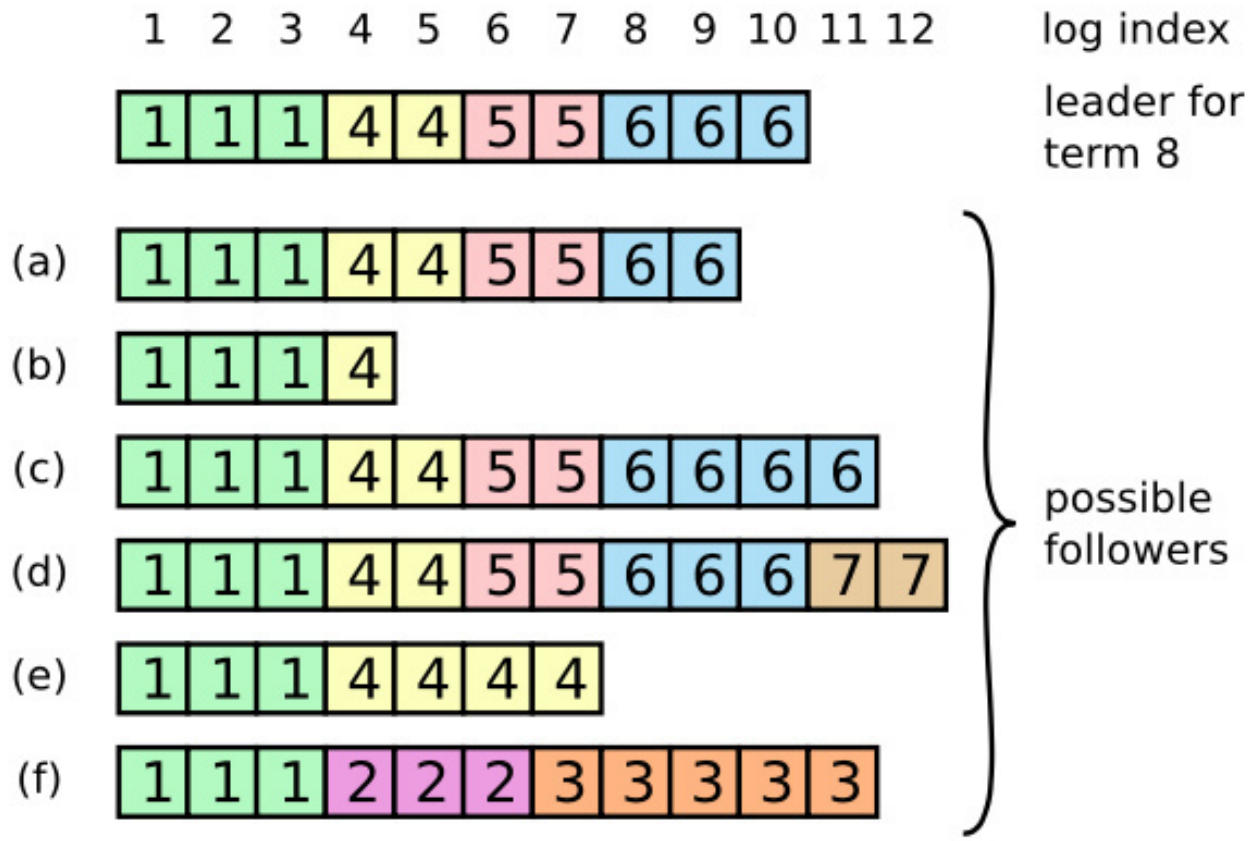


图 112: 图 7

图 7：当一个领导人成功当选时，跟随者可能是任何情况（a-f）。每一个盒子表示是一个日志条目；里面的数字表示任期号。跟随者可能会缺少一些日志条目（a-b），可能会有一些未被提交的日志条目（c-d），或者两种情况都存在（e-f）。例如，场景 f 可能会这样发生，某服务器在任期 2 的时候是领导人，已附加了一些日志条目到自己的日志中，但在提交之前就崩溃了；很快这个机器就被重启了，在任期 3 重新被选为领导人，并且又增加了一些日志条目到自己的日志中；在任期 2 和任期 3 的日志被提交之前，这个服务器又宕机了，并且在接下来的几个任期里一直处于宕机状态。

在 Raft 算法中，领导人处理不一致是通过强制跟随者直接复制自己的日志来解决了。这意味着在跟随者中的冲突的日志条目会被领导人的日志覆盖。5.4 节会阐述如何通过增加一些限制来使得这样的操作是安全的。

要使得跟随者的日志进入和自己一致的状态，领导人必须找到最后两者达成一致的地方，然后删除从那个点之后的所有日志条目，发送自己的日志给跟随者。所有的这些操作都在进行附加日志 RPCs 的一致性检查时完成。领导人针对每一个跟随者维护了一个 **nextIndex**，这表示下一个需要发送给跟随者的日志条目的索引地址。当一个领导人刚获得权力的时候，他初始化所有的 nextIndex 值为自己的最后一条日志的 index 加 1（图 7 中的 11）。如果一个跟随者的日志和领导人不一致，那么在下一次的附加日志 RPC 时的一致性检查就会失败。在被跟随者拒绝之后，领导人就会减小 nextIndex 值并进行重试。最终 nextIndex 会在某个位置使得领导人和跟随者的日志达成一致。当这种情况发生，附加日志 RPC 就会成功，这时就会把跟随者冲突的日志条目全部删除并且加上领导人的日志。一旦附加日志 RPC 成功，那么跟随者的日志就会和领导人保持一致，并且在接下来的任期里一直继续保持。

如果需要的话，算法可以通过减少被拒绝的附加日志 RPCs 的次数来优化。例如，当附加日志 RPC 的请求被拒绝的时候，跟随者可以包含冲突的条目的任期号和自己存储的那个任期的最早的索引地址。借助这些信息，领导人可以减小 nextIndex 越过所有那个任期冲突的所有日志条目；这样就变成每个任期需要一次附加条目 RPC 而不是每个条目一次。在实践中，我们十分怀疑这种优化是否是必要的，因为失败是很少发生的并且也不大可能会有这么多不一致的日志。

通过这种机制，领导人在获得权力的时候就不需要任何特殊的操作来恢复一致性。他只需要进行正常的操作，然后日志就能自动的在回复附加日志 RPC 的一致性检查失败的时候自动趋于一致。领导人从来不会覆盖或者删除自己的日志（图 3 的领导人只附加特性）。

日志复制机制展示出了第 2 节中形容的一致性特性：Raft 能够接受，复制并应用新的日志条目只要大部分的机器是工作的；在通常的情况下，新的日志条目可以在一次 RPC 中被复制给集群中的大多数机器；并且单个的缓慢的跟随者不会影响整体的性能。

8.1.6.4 安全性

前面的章节里描述了 Raft 算法是如何选举和复制日志的。然而，到目前为止描述的机制并不能充分的保证每一个状态机会按照相同的顺序执行相同的指令。例如，一个跟随者可能会进入不可用状态同时领导人已经提交了若干的日志条目，然后这个跟随者可能会被选举为领导人并且覆盖这些日志条目；因此，不同的状态机可能会执行不同的指令序列。

这一节通过在领导选举的时候增加一些限制来完善 Raft 算法。这一限制保证了任何的领导人对于给定的任期号，都拥有了之前任期的所有被提交的日志条目（图 3 中的领导人完整特性）。增加这一选举时的限制，我们对于提交时的规则也更加清晰。最终，我们将展示对于领导人完整特性的简要证明，并且说明领导人是如何领导复制状态机的做出正确行为的。

8.1.6.4.1 选举限制

在任何基于领导人的一致性算法中，领导人都必须存储所有已经提交的日志条目。在某些一致性算法中，例如

Viewstamped Replication，某个节点即使是一开始并没有包含所有已经提交的日志条目，它也能被选为领导者。这些算法都包含一些额外的机制来识别丢失的日志条目并把他们传送给新的领导人，要么是在选举阶段要么在之后很快进行。不幸的是，这种方法会导致相当大的额外的机制和复杂性。Raft 使用了一种更加简单的方法，它可以保证所有之前的任期号中已经提交的日志条目在选举的时候都会出现在新的领导人中，不需要传送这些日志条目给领导人。这意味着日志条目的传送是单向的，只从领导人传给跟随者，并且领导人从不会覆盖自身本地日志中已经存在的条目。

Raft 使用投票的方式来阻止一个候选人赢得选举除非这个候选人包含了所有已经提交的日志条目。候选人为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目在这些服务器节点中肯定存在于至少一个节点上。如果候选人的日志至少和大多数的服务器节点一样新（这个新的定义会在下面讨论），那么他一定持有了所有已经提交的日志条目。请求投票 RPC 实现了这样的限制：RPC 中包含了候选人的日志信息，然后投票人会拒绝掉那些日志没有自己新的投票请求。

Raft 通过比较两份日志中最后一条日志条目的索引值和任期号定义谁的日志比较新。如果两份日志最后的条目的任期号不同，那么任期号大的日志更加新。如果两份日志最后的条目任期号相同，那么日志比较长的那个就更加新。

8.1.6.4.2 提交之前任期内的日志条目

如同 5.3 节介绍的那样，领导人知道一条当前任期内的日志记录是可以被提交的，只要它被存储到了大多数的服务器上。如果一个领导人在提交日志条目之前崩溃了，未来后续的领导人会继续尝试复制这条日志记录。然而，一个领导人不能断定一个之前任期里的日志条目被保存到大多数服务器上的时候就一定已经提交了。图 8 展示了一种情况，一条已经被存储到大多数节点上的老日志条目，也依然有可能会被未来的领导人覆盖掉。

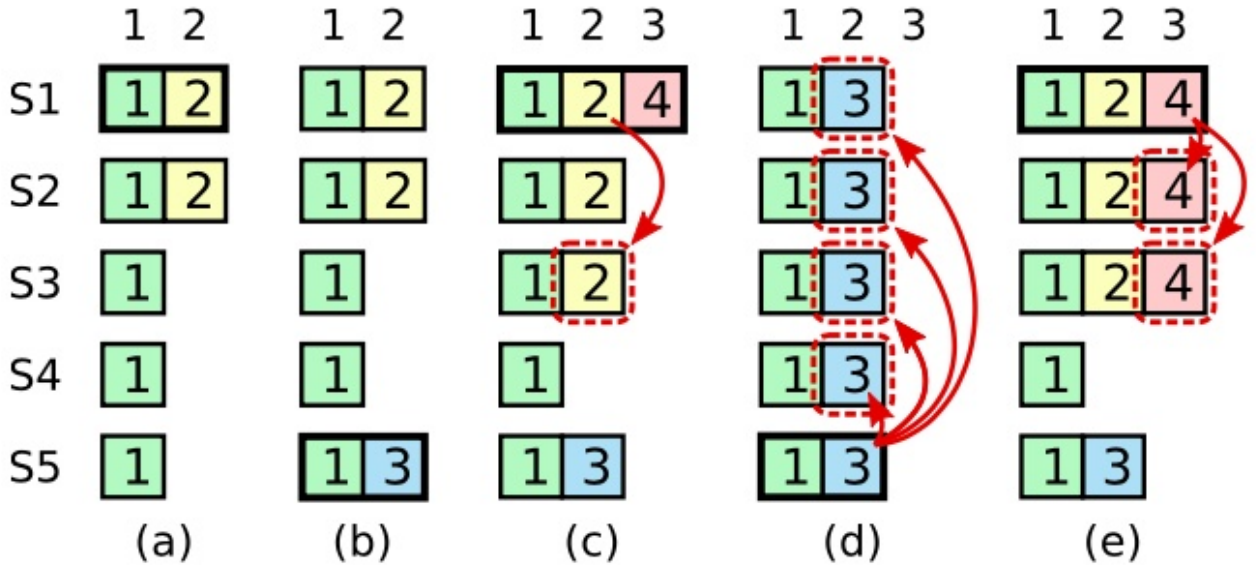


图 113: 图 8

图 8：如图的时间序列展示了为什么领导人无法决定对老任期号的日志条目进行提交。在 (a) 中，S1 是领导者，部分的复制了索引位置 2 的日志条目。在 (b) 中，S1 崩溃了，然后 S5 在任期 3 里通过 S3、S4 和自己的选票赢得选举，然后从客户端接收了一条不一样的日志条目放在了索引 2 处。然后到 (c)，S5 又崩溃了；S1 重新启动，选举成功，开始复制日志。在这时，来自任期 2 的那条日志已经被复制到了集群中的大多数机器上，但是还没有被提交。如果 S1 在 (d) 中又崩溃了，S5 可以重新被选举成功（通过来自 S2，S3 和 S4 的选票），然后覆盖了他们在索引 2 处的日志。反之，如果在崩溃之前，S1 把自己主导的新任期里产生的日志条目复制到了大多数机

> 器上，就如 (e) 中那样，那么在后面任期里面这些新的日志条目就会被提交（因为 S5 就不可能选举成功）。这样在同一时刻就同时保证了，之前的所有老的日志条目就会被提交。

为了消除图 8 里描述的情况，Raft 永远不会通过计算副本数目的方式去提交一个之前任期内的日志条目。只有领导人当前任期里的日志条目通过计算副本数目可以被提交；一旦当前任期的日志条目以这种方式被提交，那么由于日志匹配特性，之前的日志条目也都会被间接的提交。在某些情况下，领导人可以安全的知道一个老的日志条目是否已经被提交（例如，该条目是否存储到所有服务器上），但是 Raft 为了简化问题使用一种更加保守的方法。

当领导人复制之前任期里的日志时，Raft 会为所有日志保留原始的任期号，这在提交规则上产生了额外的复杂性。在其他的一致性算法中，如果一个新的领导人要重新复制之前的任期里的日志时，它必须使用当前新的任期号。Raft 使用的方法更加容易辨别出日志，因为它可以随着时间和日志的变化对日志维护着同一个任期编号。另外，和其他的算法相比，Raft 中的新领导人只需要发送更少日志条目（其他算法中必须在他们被提交之前发送更多的冗余日志条目来为他们重新编号）。

8.1.6.4.3 安全性论证

在给定了完整的 Raft 算法之后，我们现在可以更加精确的讨论领导人完整性特性（这一讨论基于 9.2 节的安全性证明）。我们假设领导人完全性特性是不存在的，然后我们推出矛盾来。假设任期 T 的领导人（领导人 T）在任期内提交了一条日志条目，但是这条日志条目没有被存储到未来某个任期的领导人的日志中。设大于 T 的最小任期 U 的领导人 U 没有这条日志条目。

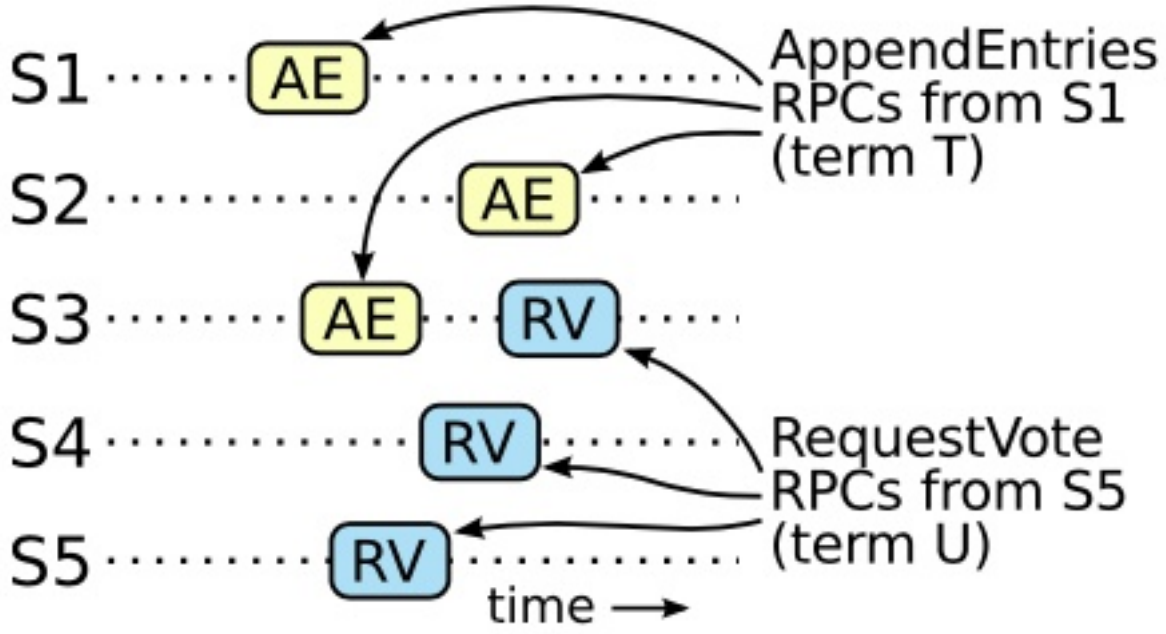


图 114: 图 9

> 图 9：如果 S1 （任期 T 的领导者）提交了一条新的日志在它的任期里，然后 S5 在之后的任期 U 里被选举为领导人，然后至少会有一个机器，如 S3，既拥有来自 S1 的日志，也给 S5 投票了。

1. 在领导人 U 选举的时候一定没有那条被提交的日志条目（领导人从不会删除或者覆盖任何条目）。
2. 领导人 T 复制这条日志条目给集群中的大多数节点，同时，领导人 U 从集群中的大多数节点赢得了选票。因此，至少有一个节点（投票者、选民）同时接受了来自领导人 T 的日志条目，并且给领导人 U 投票了，

如图 9。这个投票者是产生这个矛盾的关键。

3. 这个投票者必须在给领导人 U 投票之前先接受了从领导人 T 发来的已经被提交的日志条目；否则他就会拒绝来自领导人 T 的附加日志请求（因为此时他的任期号会比 T 大）。
4. 投票者在给领导人 U 投票时依然保存有这条日志条目，因为任何中间的领导人都包含该日志条目（根据上述的假设），领导人从不会删除条目，并且跟随者只有在和领导人冲突的时候才会删除条目。
5. 投票者把自己选票投给领导人 U 时，领导人 U 的日志必须和投票者自己一样新。这就导致了两者矛盾之一。
6. 首先，如果投票者和领导人 U 的最后一条日志的任期号相同，那么领导人 U 的日志至少和投票者一样长，所以领导人 U 的日志一定包含所有投票者的日志。这是另一处矛盾，因为投票者包含了那条已经被提交的日志条目，但是在上述的假设里，领导人 U 是不包含的。
7. 除此之外，领导人 U 的最后一条日志的任期号就必须比投票人大了。此外，他也比 T 大，因为投票人的最后一条日志的任期号至少和 T 一样大（他包含了来自任期 T 的已提交的日志）。创建了领导人 U 最后一条日志的之前领导人一定已经包含了那条被提交的日志（根据上述假设，领导人 U 是第一个不包含该日志条目的领导人）。所以，根据日志匹配特性，领导人 U 一定也包含那条被提交的日志，这里产生矛盾。
8. 这里完成了矛盾。因此，所有比 T 大的领导人一定包含了所有来自 T 的已经被提交的日志。
9. 日志匹配原则保证了未来的领导人也同时会包含被间接提交的条目，例如图 8 (d) 中的索引 2。

通过领导人完全特性，我们就能证明图 3 中的状态机安全特性，即如果服务器已经在某个给定的索引值应用了日志条目到自己的状态机里，那么其他的服务器不会应用一个不一样的日志到同一个索引值上。在一个服务器应用一条日志条目到他自己的状态机中时，他的日志必须和领导人的日志，在该条目和之前的条目上相同，并且已经被提交。现在我们来考虑在任何一个服务器应用一个指定索引位置的日志的最小任期；日志完全特性保证拥有更高任期号的领导人会存储相同的日志条目，所以之后的任期里应用某个索引位置的日志条目也会是相同的值。因此，状态机安全特性是成立的。

最后，Raft 要求服务器按照日志中索引位置顺序应用日志条目。和状态机安全特性结合起来看，这就意味着所有的服务器会应用相同的日志序列集到自己的状态机中，并且是按照相同的顺序。

8.1.6.5 跟随者和候选人崩溃

到目前为止，我们都只关注了领导人崩溃的情况。跟随者和候选人崩溃后的处理方式比领导人要简单的多，并且他们的处理方式是相同的。如果跟随者或者候选人崩溃了，那么后续发送给他们的 RPCs 都会失败。Raft 中处理这种失败就是简单的通过无限的重试；如果崩溃的机器重启了，那么这些 RPC 就会完整的成功。如果一个服务器在完成了一个 RPC，但是还没有响应的时候崩溃了，那么在他重新启动之后就会再次收到同样的请求。Raft 的 RPCs 都是幂等的，所以这样重试不会造成任何问题。例如一个跟随者如果收到附加日志请求但是他已经包含了这一日志，那么他就会直接忽略这个新的请求。

8.1.6.6 时间和可用性

Raft 的要求之一就是安全性不能依赖时间：整个系统不能因为某些事件运行的比预期快一点或者慢一点就产生了错误的结果。但是，可用性（系统可以及时的响应客户端）不可避免的要依赖于时间。例如，如果消息交换比服务器故障间隔时间长，候选人将没有足够长的时间来赢得选举；没有一个稳定的领导人，Raft 将无法工作。

领导人选举是 Raft 中对时间要求最为关键的方面。Raft 可以选举并维持一个稳定的领导人，只要系统满足下面的时间要求：

> 广播时间（broadcastTime）« 选举超时时间（electionTimeout）« 平均故障间隔时间（MTBF）

在这个不等式中，广播时间指的是从一个服务器并行的发送 RPCs 给集群中的其他服务器并接收响应的平均时间；选举超时时间就是在 5.2 节中介绍的选举的超时时间限制；然后平均故障间隔时间就是对于一台服务器而言，两次故障之间的平均时间。广播时间必须比选举超时时间小一个量级，这样领导人才能够发送稳定的心跳

消息来阻止跟随者开始进入选举状态；通过随机化选举超时时间的方法，这个不等式也使得选票瓜分的情况变得不可能。选举超时时间应该要比平均故障间隔时间小上几个数量级，这样整个系统才能稳定的运行。当领导人崩溃后，整个系统会大约相当于选举超时的时间里不可用；我们希望这种情况在整个系统的运行中很少出现。

广播时间和平均故障间隔时间是由系统决定的，但是选举超时时间是我们自己选择的。Raft 的 RPCs 需要接收方将信息持久化的保存到稳定存储中去，所以广播时间大约是 0.5 毫秒到 20 毫秒，取决于存储的技术。因此，选举超时时间可能需要在 10 毫秒到 500 毫秒之间。大多数的服务器的平均故障间隔时间都在几个月甚至更长，很容易满足时间的需求。

8.1.7 集群成员变化

到目前为止，我们都假设集群的配置（加入到一致性算法的服务器集合）是固定不变的。但是在实践中，偶尔是会改变集群的配置的，例如替换那些宕机的机器或者改变复制级别。尽管可以通过暂停整个集群，更新所有配置，然后重启整个集群的方式来实现，但是在更改的时候集群会不可用。另外，如果存在手工操作步骤，那么就会有操作失误的风险。为了避免这样的问题，我们决定自动化配置改变并且将其纳入到 Raft 一致性算法中来。

为了让配置修改机制能够安全，那么在转换的过程中不能够存在任何时间点使得两个领导人同时被选举成功在同一个任期里。不幸的是，任何服务器直接从旧的配置直接转换到新的配置的方案都是不安全的。一次性自动的转换所有服务器是不可能的，所以在转换期间整个集群存在划分成两个独立的大多数群体的可能性（见图 10）。

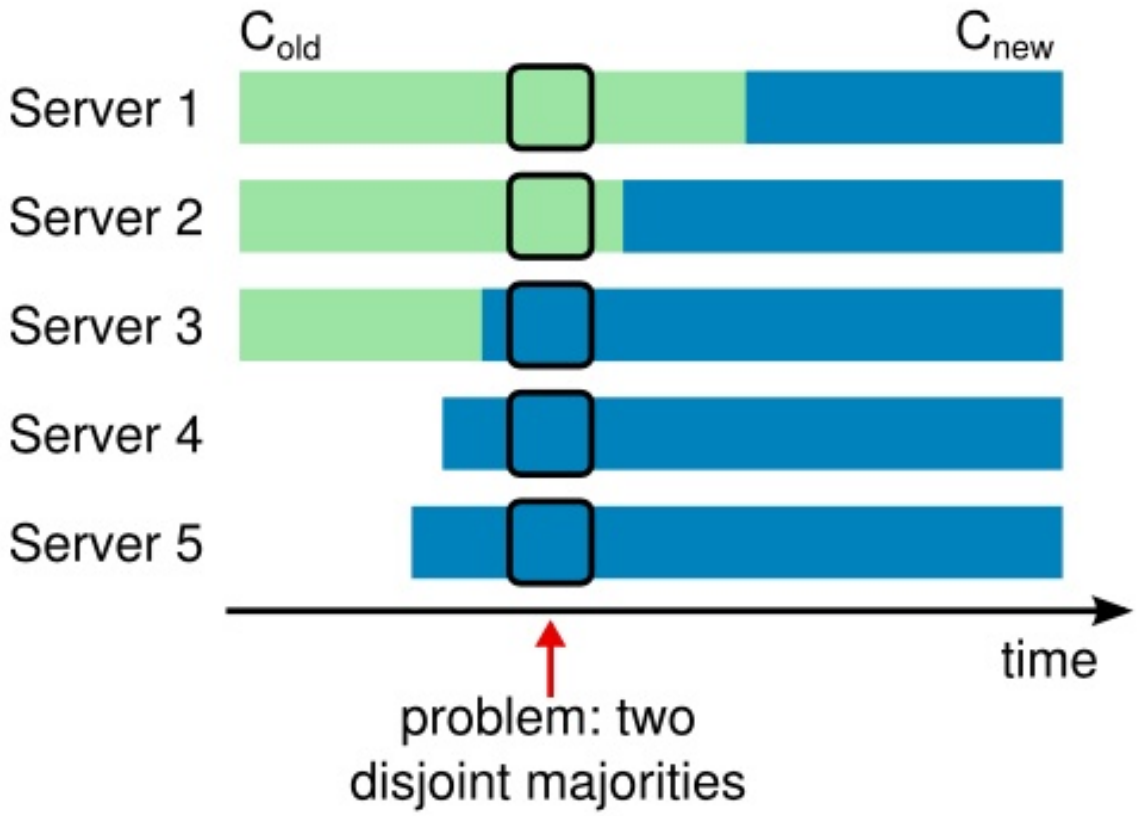


图 115: 图 10

图 10：直接从一种配置转到新的配置是十分不安全的，因为各个机器可能在任何的时候进行转换。在这个例子中，集群配额从 3 台机器变成了 5 台。不幸的是，存在这样的一个时间点，两个不同的领导人在同一个任期里都可以被选举成功。一个是通过旧的配置，一个通过新的配置。

为了保证安全性，配置更改必须使用两阶段方法。目前有很多种两阶段的实现。例如，有些系统在第一阶段停

掉旧的配置所以集群就不能处理客户端请求；然后在第二阶段在启用新的配置。在 Raft 中，集群先切换到一个过渡的配置，我们称之为共同一致；一旦共同一致已经被提交了，那么系统就切换到新的配置上。共同一致是老配置和新配置的结合：

- 日志条目被复制给集群中新、老配置的所有服务器。
- 新、旧配置的服务器都可以成为领导人。
- 达成一致（针对选举和提交）需要分别在两种配置上获得大多数的支持。

共同一致允许独立的服务器在不影响安全性的前提下，在不同的时间进行配置转换过程。此外，共同一致可以让集群在配置转换的过程人依然响应客户端的请求。

集群配置在复制日志中以特殊的日志条目来存储和通信；图 11 展示了配置转换的过程。当一个领导人接收到一个改变配置从 C-old 到 C-new 的请求，他会为了共同一致存储配置（图中的 C-old,new），以前面描述的日志条目和副本的形式。一旦一个服务器将新的配置日志条目增加到它的日志中，他就会用这个配置来做出未来所有的决定（服务器总是使用最新的配置，无论他是否已经被提交）。这意味着领导人要使用 C-old, new 的规则来决定日志条目 C-old,new 什么时候需要被提交。如果领导人崩溃了，被选出来的新领导人可能是使用 C-old 配置也可能是 C-old, new 配置，这取决于赢得选举的候选人是否已经接收到了 C-old,new 配置。在任何情况下，C-new 配置在这一时期都不会单方面的做出决定。

一旦 C-old,new 被提交，那么无论是 C-old 还是 C-new，在没有经过他人批准的情况下都不可能做出决定，并且领导人完全特性保证了只有拥有 C-old, new 日志条目的服务器才有可能被选举为领导人。这个时候，领导人创建一条关于 C-new 配置的日志条目并复制给集群就是安全的了。再者，每个服务器在见到新的配置的时候就会立即生效。当新的配置在 C-new 的规则下被提交，旧的配置就变得无关紧要，同时不使用新的配置的服务器就可以被关闭了。如图 11，C-old 和 C-new 没有任何机会同时做出单方面的决定；这保证了安全性。

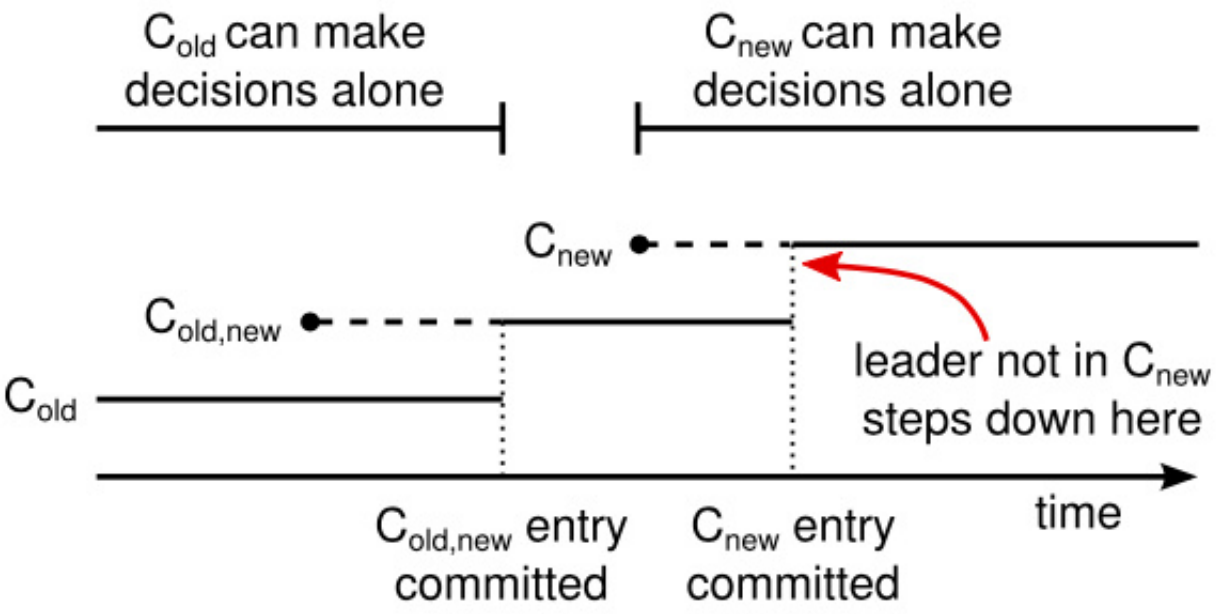


图 116: 图 11

图 11: 一个配置切换的时间线。虚线表示已经被创建但是还没有被提交的条目，实线表示最后被提交的日志条目。领导人首先创建了 C-old,new 的配置条目在自己的日志中，并提交到 C-old,new 中（C-old 的大多数和 C-new 的大多数）。然后他创建 C-new 条目并提交到 C-new 中的大多数。这样就不存在 C-new 和 C-old 可以同时做出决定的时间点。

在关于重新配置还有三个问题需要提出。第一个问题是，新的服务器可能初始化没有存储任何的日志条目。当

这些服务器以这种状态加入到集群中，那么他们需要一段时间来更新追赶，这时还不能提交新的日志条目。为了避免这种可用性的间隔时间，Raft 在配置更新的时候使用了一种额外的阶段，在这个阶段，新的服务器以没有投票权身份加入到集群中来（领导人复制日志给他们，但是不考虑他们是大多数）。一旦新的服务器追赶上了集群中的其他机器，重新配置可以像上面描述的一样处理。

第二个问题是，集群的领导人可能不是新配置的一员。在这种情况下，领导人就会在提交了 C-new 日志之后退位（回到跟随者状态）。这意味着有这样的一段时间，领导人管理着集群，但是不包括他自己；他复制日志但是不把他自己算作是大多数之一。当 C-new 被提交时，会发生领导人过渡，因为这时是最早新的配置可以独立工作的时间点（将总是能够在 C-new 配置下选出新的领导人）。在此之前，可能只能从 C-old 中选出领导人。

第三个问题是，移除不在 C-new 中的服务器可能会扰乱集群。这些服务器将不会再接收到心跳，所以当选举超时，他们就会进行新的选举过程。他们会发送拥有新的任期号的请求投票 RPCs，这样会导致当前的领导人回退成跟随者状态。新的领导人最终会被选出来，但是被移除的服务器将会再次超时，然后这个过程会再次重复，导致整体可用性大幅降低。

为了避免这个问题，当服务器确认当前领导人存在时，服务器会忽略请求投票 RPCs。特别的，当服务器在当前最小选举超时时间内收到一个请求投票 RPC，他不会更新当前的任期号或者投出选票。这不会影响正常的选举，每个服务器在开始一次选举之前，至少等待一个最小选举超时时间。然而，这有利于避免被移除的服务器扰乱：如果领导人能够发送心跳给集群，那么他就不会被更大的任期号废黜。

8.1.8 日志压缩

Raft 的日志在正常操作中不断的增长，但是在实际的系统中，日志不能无限制的增长。随着日志不断增长，他会占用越来越多的空间，花费越来越多的时间来重置。如果没有一定的机制去清除日志里积累的陈旧的信息，那么会带来可用性问题。

快照是最简单的压缩方法。在快照系统中，整个系统的状态都以快照的形式写入到稳定的持久化存储中，然后到那个时间点之前的日志全部丢弃。快照技术被使用在 Chubby 和 ZooKeeper 中，接下来的章节会介绍 Raft 中的快照技术。

增量压缩的方法，例如日志清理或者日志结构合并树，都是可行的。这些方法每次只对一小部分数据进行操作，这样就分散了压缩的负载压力。首先，他们先选择一个已经积累的大量已经被删除或者被覆盖对象的区域，然后重写那个区域还活跃的对象，之后释放那个区域。和简单操作整个数据集合的快照相比，需要增加复杂的机制来实现。状态机可以实现 LSM tree 使用和快照相同的接口，但是日志清除方法就需要修改 Raft 了。

> 图 12：一个服务器用新的快照替换了从 1 到 5 的条目，快照值存储了当前的状态。快照中包含了最后的索引位置和任期号。

图 12 展示了 Raft 中快照的基础思想。每个服务器独立的创建快照，只包括已经被提交的日志。主要的工作包括将状态机的状态写入到快照中。Raft 也包含一些少量的元数据到快照中：**最后被包含索引**指的是被快照取代的最后的条目在日志中的索引值（状态机最后应用的日志），**最后被包含的任期**指的是该条目的任期号。保留这些数据是为了支持快照后紧接着的第一个条目的附加日志请求时的一致性检查，因为这个条目需要前一日志条目的索引值和任期号。为了支持集群成员更新（第 6 节），快照中也将最后的一次配置作为最后一个条目存下来。一旦服务器完成一次快照，他就可以删除最后索引位置之前的所有日志和快照了。

尽管通常服务器都是独立的创建快照，但是领导人必须偶尔的发送快照给一些落后的跟随者。这通常发生在当领导人已经丢弃了下一条需要发送给跟随者的日志条目的时候。幸运的是这种情况不是常规操作：一个与领导人保持同步的跟随者通常都会有这个条目。然而一个运行非常缓慢的跟随者或者新加入集群的服务器（第 6 节）将不会有这个条目。这时让这个跟随者更新到最新的状态的方式就是通过网络把快照发送给他们。

安装快照 RPC：

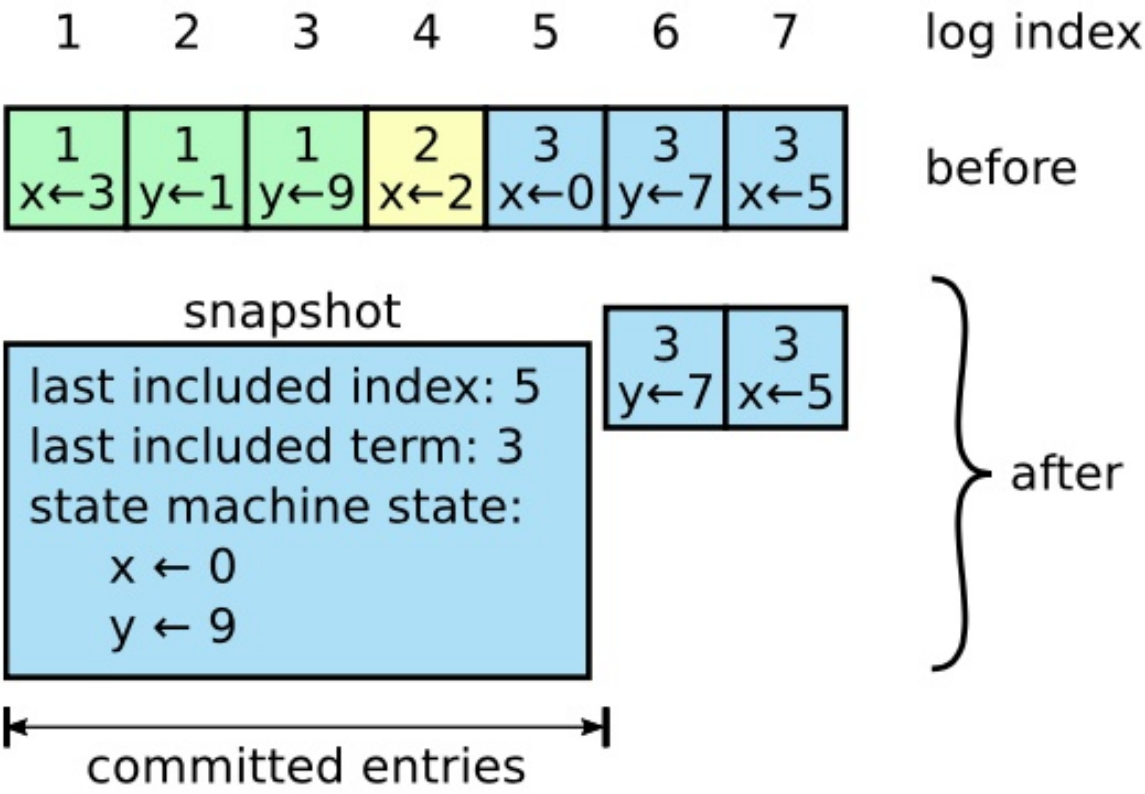


图 117: 图 12

由领导人调用以将快照的分块发送给跟随者。领导者总是按顺序发送分块。

参数	解释
term	领导人的任期号
leaderId	领导人的 Id，以便于跟随者重定向请求
lastIncludedIndex	快照中包含的最后日志条目的索引值
lastIncludedTerm	快照中包含的最后日志条目的任期号
offset	分块在快照中的字节偏移量
data[]	原始数据
done	如果这是最后一个分块则为 true

结果	解释
term	当前任期号（currentTerm），便于领导人更新自己

接收者实现:

1. 如果`term < currentTerm`就立即回复
2. 如果是第一个分块（offset 为 0）就创建一个新的快照
3. 在指定偏移量写入数据
4. 如果 done 是 false，则继续等待更多的数据
5. 保存快照文件，丢弃具有较小索引的任何现有或部分快照
6. 如果现存的日志条目与快照中最后包含的日志条目具有相同的索引值和任期号，则保留其后的日志条目并进行回复
7. 丢弃整个日志
8. 使用快照重置状态机（并加载快照的集群配置）

图 13: 一个关于安装快照的简要概述。为了便于传输，快照都是被分成分块的；每个分块都给

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower.
Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

图 118: 图 13

么就会浪费大量的磁盘带宽和其他资源；如果创建快照频率太低，他就要承受耗尽存储容量的风险，同时也增加了从日志重建的时间。一个简单的策略就是当日志大小达到一个固定大小的时候就创建一次快照。如果这个阈值设置的显著大于期望的快照的大小，那么快照对磁盘压力的影响就会很小了。

第二个影响性能的问题就是写入快照需要花费显著的一段时间，并且我们还不希望影响到正常操作。解决方案是通过写时复制的技术，这样新的更新就可以被接收而不影响到快照。例如，具有函数式数据结构的状态机天然支持这样的功能。另外，操作系统的写时复制技术的支持（如 Linux 上的 fork）可以被用来创建完整的状态机的内存快照（我们的实现就是这样的）。

8.1.9 客户端交互

这一节将介绍客户端是如何和 Raft 进行交互的，包括客户端如何发现领导人和 Raft 是如何支持线性化语义的。这些问题对于所有基于一致性的系统都存在，并且 Raft 的解决方案和其他的也差不多。

Raft 中的客户端发送所有请求给领导人。当客户端启动的时候，他会随机挑选一个服务器进行通信。如果客户端第一次挑选的服务器不是领导人，那么那个服务器会拒绝客户端的请求并且提供他最近接收到的领导人的信息（附加条目请求包含了领导人的网络地址）。如果领导人已经崩溃了，那么客户端的请求就会超时；客户端之后会再次重试随机挑选服务器的过程。

我们 Raft 的目标是要实现线性化语义（每一次操作立即执行，只执行一次，在他调用和收到回复之间）。但是，如上述，Raft 是可以执行同一条命令多次的：例如，如果领导人在提交了这条日志之后，但是在响应客户端之前崩溃了，那么客户端会和新的领导人重试这条指令，导致这条命令就被再次执行了。解决方案就是客户端对于每一条指令都赋予一个唯一的序列号。然后，状态机跟踪每条指令最新的序列号和相应的响应。如果接收到一条指令，它的序列号已经被执行了，那么就立即返回结果，而不重新执行指令。

只读的操作可以直接处理而不需要记录日志。但是，在不增加任何限制的情况下，这么做可能会冒着返回脏数据的风险，因为领导人响应客户端请求时可能已经被新的领导人作废了，但是他还不知道。线性化的读操作必须不能返回脏数据，Raft 需要使用两个额外的措施在不使用日志的情况下保证这一点。首先，领导人必须有关于被提交日志的最新信息。领导人完全特性保证了领导人一定拥有所有已经被提交的日志条目，但是在他任期开始的时候，他可能不知道那些是已经被提交的。为了知道这些信息，他需要在他的任期里提交一条日志条目。Raft 中通过领导人在任期开始的时候提交一个空白的没有任何操作的日志条目到日志中去来实现。第二，领导人在处理只读的请求之前必须检查自己是否已经被废黜了（它自己的信息已经变脏了如果一个更新的领导人被选举出来）。Raft 中通过让领导人在响应只读请求之前，先和集群中的大多数节点交换一次心跳信息来处理这个问题。可选的，领导人可以依赖心跳机制来实现一种租约的机制，但是这种方法依赖时间来保证安全性（假设时间误差是有界的）。

8.1.10 算法实现和评估

我们已经为 RAMCloud 实现了 Raft 算法作为存储配置信息的复制状态机的一部分，并且帮助 RAMCloud 协调故障转移。这个 Raft 实现包含大约 2000 行 C++ 代码，其中不包括测试、注释和空行。这些代码是开源的。同时也有大约 25 个其他独立的第三方的基于这篇论文草稿的开源实现，针对不同的开发场景。同时，很多公司已经部署了基于 Raft 的系统。

这一节会从三个方面来评估 Raft 算法：可理解性、正确性和性能。

8.1.10.1 可理解性

为了和 Paxos 比较 Raft 算法的可理解能力，我们针对高层次的本科生和研究生，在斯坦福大学的高级操作系统课程和加州大学伯克利分校的分布式计算课程上，进行了一次学习的实验。我们分别拍了针对 Raft 和 Paxos 的视频课程，并准备了相应的小测验。Raft 的视频讲课覆盖了这篇论文的所有内容除了日志压缩；Paxos 讲课包含了足够的资料来创建一个等价的复制状态机，包括单决策 Paxos，多决策 Paxos，重新配置和一些实际系统需要的性能优化（例如领导人选举）。小测验测试一些对算法的基本理解和解释一些边角的示例。每个学生都是

看完第一个视频，回答相应的测试，再看第二个视频，回答相应的测试。大约有一半的学生先进行 Paxos 部分，然后另一半先进行 Raft 部分，这是为了说明两者从第一部分的算法学习中获得的表现和经验的差异。我们计算参加人员的每一个小测验的得分来看参与者是否在 Raft 算法上更加容易理解。

我们尽可能的使得 Paxos 和 Raft 的比较更加公平。这个实验偏爱 Paxos 表现在两个方面：43 个参加者中有 15 个人在之前有一些 Paxos 的经验，并且 Paxos 的视频要长 14%。如表格 1 总结的那样，我们采取了一些措施来减轻这种潜在的偏见。我们所有的材料都可供审查。

关心	缓和偏见采取的手段	可供查看的材料
相同的讲课质量	两者使用同一个讲师。Paxos 使用的是现在很多大学里经常使用的。Paxos 会长 14%。	视频
相同的测验难度	问题以难度分组，在两个测验里成对出现。	小测验
公平评分	使用评价量规。随机顺序打分，两个测验交替进行。	评价量规（rubric）

表 1：考虑到可能会存在的偏见，对于每种情况的解决方法，和相应的材料。

参加者平均在 Raft 的测验中比 Paxos 高 4.9 分（总分 60，那么 Raft 的平均得分是 25.7，而 Paxos 是 20.8）；图 14 展示了每个参与者的得分。配置 t-检验（又称 student ‘s t-test）表明，在 95% 的可信度下，真实的 Raft 分数分布至少比 Paxos 高 2.5 分。

图 14：一个散点图表示了 43 个学生在 Paxos 和 Raft 的小测验中的成绩。在对角线之上的点表示在 Raft 获得了更高分数的学生。

我们也建立了一个线性回归模型来预测一个新的学生的测验成绩，基于以下三个因素：他们使用的是哪个小测验，之前对 Paxos 的经验，和学习算法的顺序。模型预测，对小测验的选择会产生 12.5 分的差别。这显著的高于之前的 4.9 分，因为很多学生在之前都已经有了对于 Paxos 的经验，这相当明显的帮助 Paxos，对 Raft 就没什么太大影响了。但是奇怪的是，模型预测对于先进行 Paxos 小测验的人而言，Raft 的得分低了 6.3 分；虽然我们不知道为什么，这似乎在统计上是有意义的。

我们同时也在测验之后调查了参与者，他们认为哪个算法更加容易实现和解释；这个的结果在图 15 上。压倒性的结果表明 Raft 算法更加容易实现和解释（41 人中的 33 个）。但是，这种自己报告的结果不如参与者的成绩更加可信，并且参与者可能因为我们的 Raft 更加易于理解的假说而产生偏见。

图 15：通过一个 5 分制的问题，参与者（左边）被问哪个算法他们觉得在一个高效正确的系统里更容易实现，右边被问哪个更容易向学生解释。

关于 Raft 用户学习有一个更加详细的讨论。

8.1.10.2 正确性

在第 5 节，我们已经制定了正式的规范，和对一致性机制的安全性证明。这个正式规范使用 TLA+ 规范语言使图 2 中总结的信息非常清晰。它长约 400 行，并作为证明的主题。同时对于任何想实现 Raft 的人也是十分有用的。我们通过 TLA 证明系统非常机械的证明了日志完全特性。然而，这个证明依赖的约束前提还没有被机械

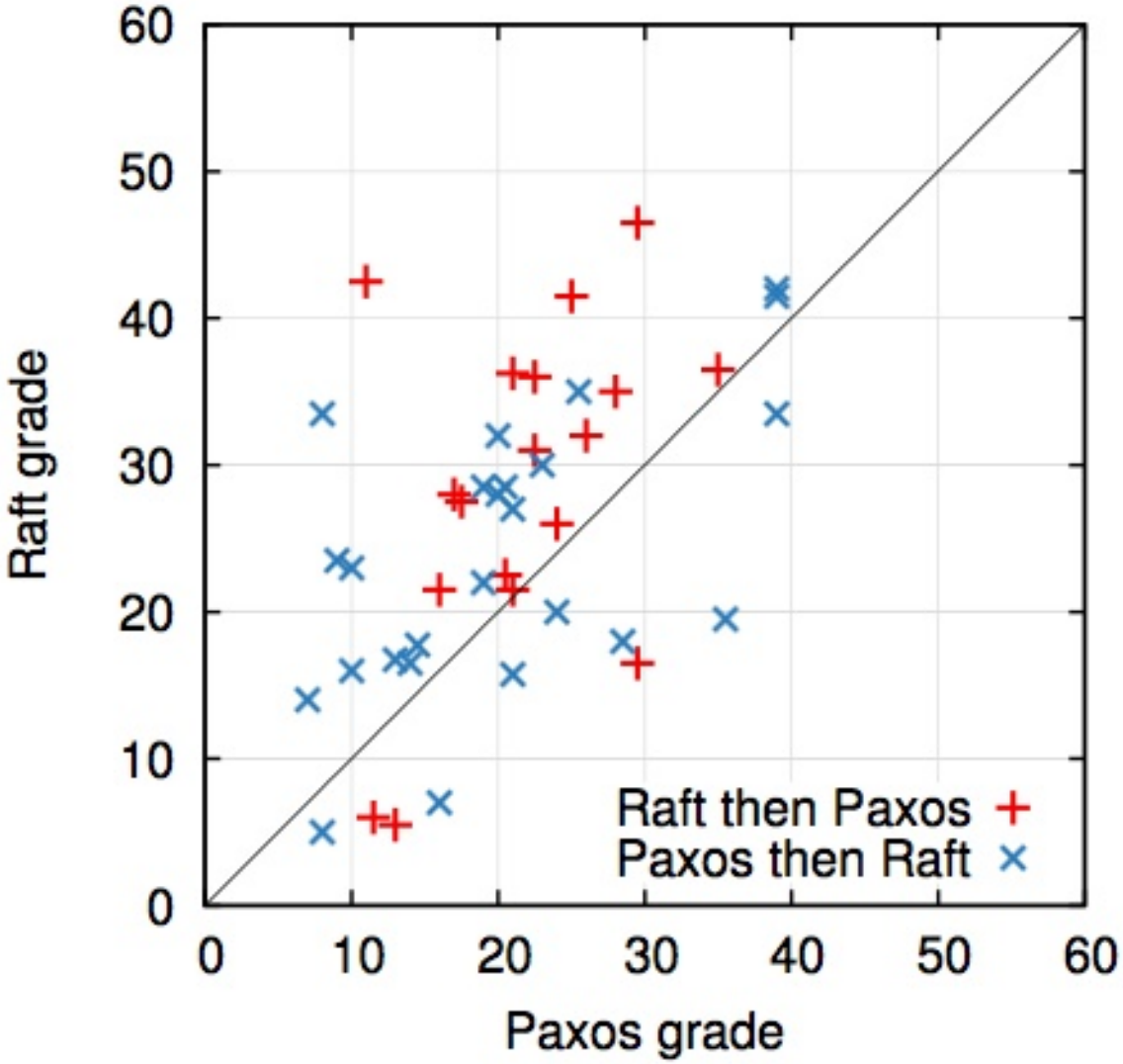


图 119: 图 14

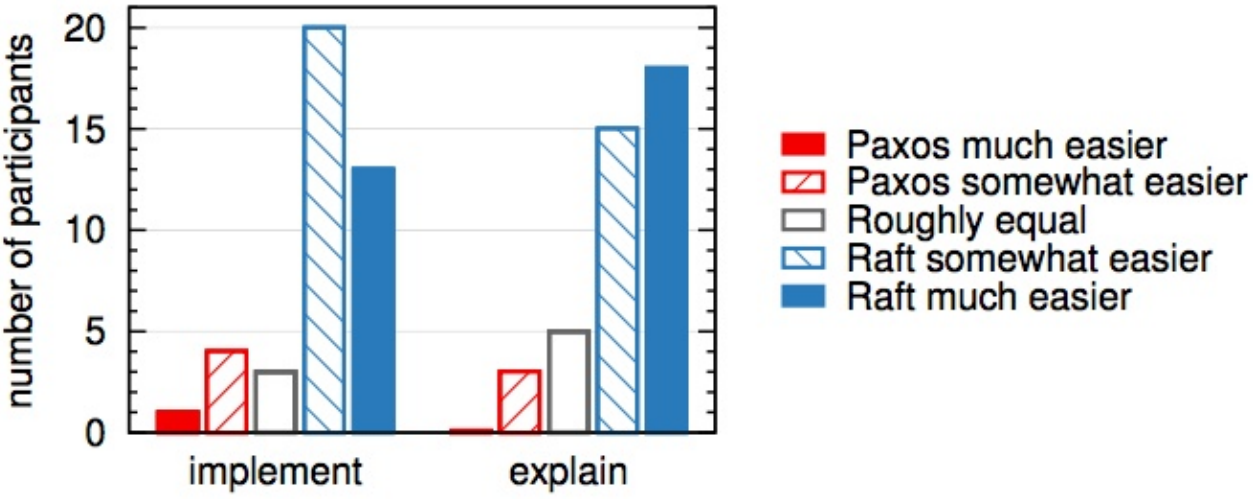


图 120: 图 15

证明（例如，我们还没有证明规范的类型安全）。而且，我们已经写了一个非正式的证明关于状态机安全性是完备的，并且是相当清晰的（大约 3500 个词）。

8.1.10.3 性能

Raft 和其他一致性算法例如 Paxos 有着差不多的性能。在性能方面，最重要的关注点是，当领导人被选举成功时，什么时候复制新的日志条目。Raft 通过很少数量的消息包（一轮从领导人到集群大多数机器的消息）就达成了这个目的。同时，进一步提升 Raft 的性能也是可行的。例如，很容易通过支持批量操作和管道操作来提高吞吐量和降低延迟。对于其他一致性算法已经提出过很多性能优化方案；其中有很多也可以应用到 Raft 中来，但是我们暂时把这个问题放到未来的工作中去。

我们使用我们自己的 Raft 实现来衡量 Raft 领导人选举的性能并且回答两个问题。首先，领导人选举的过程收敛是否快速？第二，在领导人宕机之后，最小的系统宕机时间是多久？

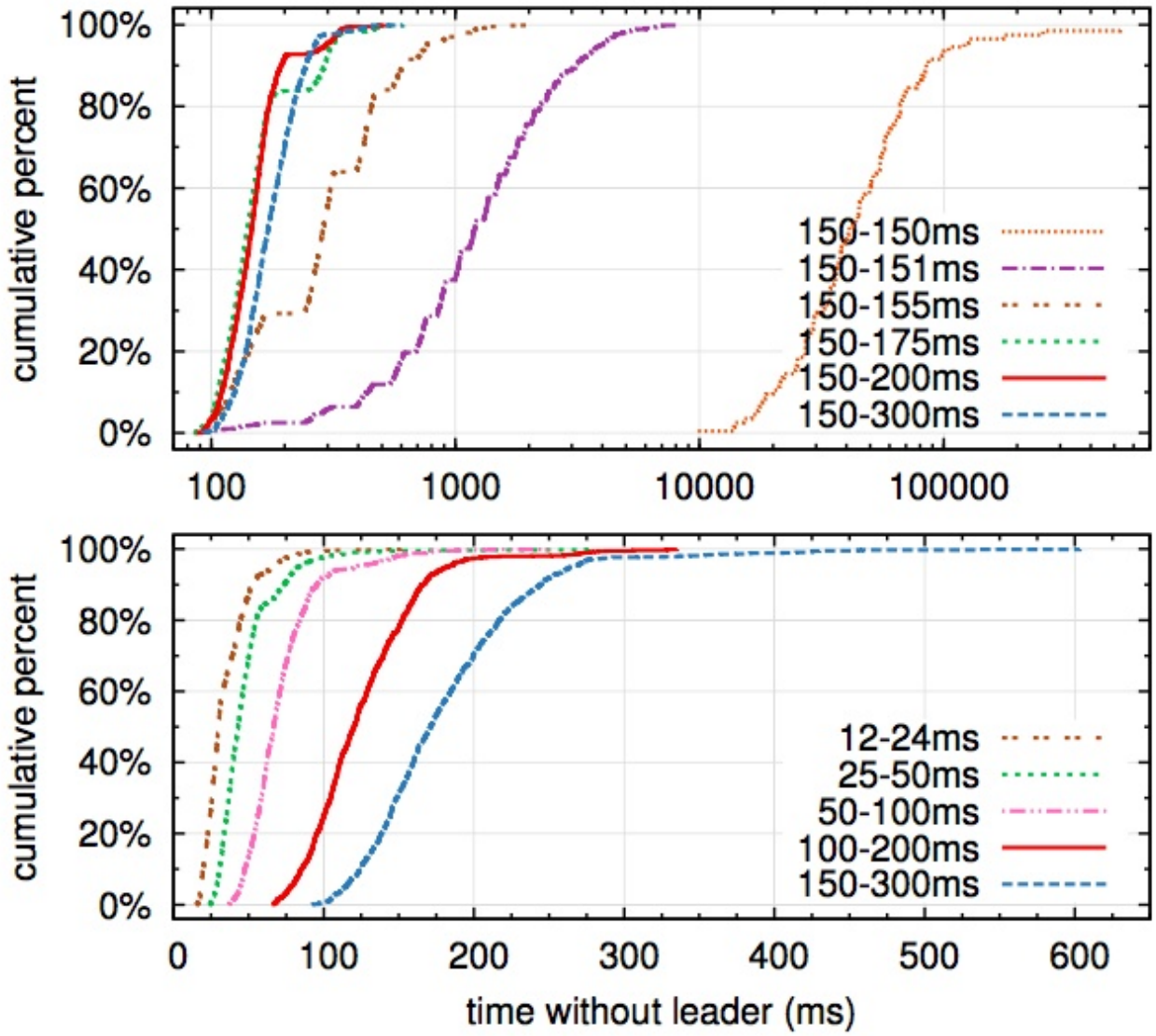


图 121: 图 16

图 16：发现并替换一个已经崩溃的领导人的时间。上面的图考察了在选举超时时间上的随机化程度，下面的图考察了最小选举超时时间。每条线代表了 1000 次实验（除了 150-150 毫秒只试了 100 次），和相应的确定的选举超时时间。例如，150-155 毫秒意思是，选举超时时间从这个

> 区间范围内随机选择并确定下来。这个实验在一个拥有 5 个节点的集群上进行，其广播时延大约是 15 毫秒。对于 9 个节点的集群，结果也差不多。

为了衡量领导人选举，我们反复的使一个拥有五个节点的服务器集群的领导人宕机，并计算需要多久才能发现领导人已经宕机并选出一个新的领导人（见图 16）。为了构建一个最坏的场景，在每一的尝试里，服务器都有不同长度的日志，意味着有些候选人是没有成为领导人的资格的。另外，为了促成选票瓜分的情况，我们的测试脚本在终止领导人之前同步的发送了一次心跳广播（这大约和领导人在崩溃前复制一个新的日志给其他机器很像）。领导人均匀的随机的在心跳间隔里宕机，也就是最小选举超时时间的一半。因此，最小宕机时间大约就是最小选举超时时间的一半。

图 16 中上面的图表明，只需要在选举超时时间上使用很少的随机化就可以大大避免选票被瓜分的情况。在没有随机化的情况下，在我们的测试里，选举过程往往都需要花费超过 10 秒钟由于太多的选票瓜分的情况。仅仅增加 5 毫秒的随机化时间，就大大的改善了选举过程，现在平均的宕机时间只有 287 毫秒。增加更多的随机化时间可以大大改善最坏情况：通过增加 50 毫秒的随机化时间，最坏的完成情况（1000 次尝试）只要 513 毫秒。

图 16 中下面的图显示，通过减少选举超时时间可以减少系统的宕机时间。在选举超时时间为 12-24 毫秒的情况下，只需要平均 35 毫秒就可以选举出新的领导人（最长的一次花费了 152 毫秒）。然而，进一步降低选举超时时间的话就会违反 Raft 的时间不等式需求：在选举新领导人之前，领导人就很难发送完心跳包。这会导致没有意义的领导人改变并降低了系统整体的可用性。我们建议使用更为保守的选举超时时间，比如 150-300 毫秒；这样的时间不大可能导致没有意义的领导人改变，而且依然提供不错的可用性。

8.1.11 相关工作

已经有很多关于一致性算法的工作被发表出来，其中很多都可以归到下面的类别中：

- Lamport 关于 Paxos 的原始描述，和尝试描述的更清晰。
- 关于 Paxos 的更详尽的描述，补充遗漏的细节并修改算法，使得可以提供更加容易的实现基础。
- 实现一致性算法的系统，例如 Chubby，ZooKeeper 和 Spanner。对于 Chubby 和 Spanner 的算法并没有公开发表其技术细节，尽管他们都声称是基于 Paxos 的。ZooKeeper 的算法细节已经发表，但是和 Paxos 着实有着很大的差别。
- Paxos 可以应用的性能优化。
- Oki 和 Liskov 的 Viewstamped Replication（VR），一种和 Paxos 差不多的替代算法。原始的算法描述和分布式传输协议耦合在了一起，但是核心的一致性算法在最近的更新里被分离了出来。VR 使用了一种基于领导人的方法，和 Raft 有很多相似之处。

Raft 和 Paxos 最大的不同之处就在于 Raft 的强领导特性：Raft 使用领导人选举作为一致性协议里必不可少的部分，并且将尽可能多的功能集中到了领导人身上。这样就可以使得算法更加容易理解。例如，在 Paxos 中，领导人选举和基本的一致性协议是正交的：领导人选举仅仅是性能优化的手段，而且不是一致性所必须要求的。但是，这样就增加了多余的机制：Paxos 同时包含了针对基本一致性要求的两阶段提交协议和针对领导人选举的独立的机制。相比较而言，Raft 就直接将领导人选举纳入到一致性算法中，并作为两阶段一致性的第一步。这样就减少了很多机制。

像 Raft 一样，VR 和 ZooKeeper 也是基于领导人的，因此他们也拥有一些 Raft 的优点。但是，Raft 比 VR 和 ZooKeeper 拥有更少的机制因为 Raft 尽可能的减少了非领导人的功能。例如，Raft 中日志条目都遵循着从领导人发送给其他人这一个方向：附加条目 RPC 是向外发送的。在 VR 中，日志条目的流动是双向的（领导人可以在选举过程中接收日志）；这就导致了额外的机制和复杂性。根据 ZooKeeper 公开的资料看，它的日志条目也是双向传输的，但是它的实现更像 Raft。

和上述我们提及的其他基于一致性的日志复制算法中，Raft 的消息类型更少。例如，我们数了一下 VR 和 ZooKeeper 使用的用来基本一致性需要和成员改变的消息数（排除了日志压缩和客户端交互，因为这些都比较

独立且和算法关系不大)。VR 和 ZooKeeper 都分别定义了 10 中不同的消息类型，相对的，Raft 只有 4 中消息类型（两种 RPC 请求和对应的响应)。Raft 的消息都稍微比其他算法的要信息量大，但是都很简单。另外，VR 和 ZooKeeper 都在领导人改变时传输了整个日志；所以为了能够实践中使用，额外的消息类型就很必要了。

Raft 的强领导人模型简化了整个算法，但是同时也排斥了一些性能优化的方法。例如，平等主义 Paxos（EPaxos）在某些没有领导人的情况下可以达到很高的性能。平等主义 Paxos 充分发挥了在状态机指令中的交换性。任何服务器都可以在一轮通信下就提交指令，除非其他指令同时被提出了。然而，如果指令都是并发的被提出，并且互相之间不通信沟通，那么 EPaxos 就需要额外的一轮通信。因为任何服务器都可以提交指令，所以 EPaxos 在服务器之间的负载均衡做的很好，并且很容易在 WAN 网络环境下获得很低的延迟。但是，他在 Paxos 上增加了非常明显的复杂性。

一些集群成员变换的方法已经被提出或者在其他的工作中被实现，包括 Lamport 的原始的讨论，VR 和 SMART。我们选择使用共同一致的方法因为他对一致性协议的其他部分影响很小，这样我们只需要很少的一些机制就可以实现成员变换。Lamport 的基于 的方法之所以没有被 Raft 选择是因为它假设在没有领导人的情况下也可以达到一致性。和 VR 和 SMART 相比较，Raft 的重新配置算法可以在不限制正常请求处理的情况下进行；相比较的，VR 需要停止所有的处理过程，SMART 引入了一个和 类似的方法，限制了请求处理的数量。Raft 的方法同时也需要更少的额外机制来实现，和 VR、SMART 比较而言。

8.1.12 结论

算法的设计通常会把正确性，效率或者简洁作为主要的目标。尽管这些都是很有意义的目标，但是我们相信，可理解性也是一样的重要。在开发者把算法应用到实际的系统中之前，这些目标没有一个会被实现，这些都会必然的偏离发表时的形式。除非开发人员对这个算法有着很深的理解并且有着直观的感觉，否则将会对他们而言很难在实现的时候保持原有期望的特性。

在这篇论文中，我们尝试解决分布式一致性问题，但是一个广为接受但是十分令人费解的算法 Paxos 已经困扰了无数学生和开发者很多年了。我们创造了一种新的算法 Raft，显而易见的比 Paxos 要容易理解。我们同时也相信，Raft 也可以为实际的实现提供坚实的基础。把可理解性作为设计的目标改变了我们设计 Raft 的方式；随着设计的进展，我们发现自己重复使用了一些技术，比如分解问题和简化状态空间。这些技术不仅提升了 Raft 的可理解性，同时也使我们坚信其正确性。

8.1.13 感谢

这项研究必须感谢以下人员的支持：Ali Ghodsi，David Mazie‘res，和伯克利 CS 294-91 课程、斯坦福 CS 240 课程的学生。Scott Klemmer 帮我们设计了用户调查，Nelson Ray 建议我们进行统计学的分析。在用户调查时使用的关于 Paxos 的幻灯片很大一部分是从 Lorenzo Alvisi 的幻灯片上借鉴过来的。特别的，非常感谢 DavidMazieres 和 Ezra Hoch，他们找到了 Raft 中一些难以发现的漏洞。许多人提供了关于这篇论文十分有用的反馈和用户调查材料，包括 Ed Bugnion，Michael Chan，Hugues Evrard，Daniel Giffin，Arjun Gopalan，Jon Howell，Vimalkumar Jeyakumar，Ankita Kejriwal，Aleksandar Kracun，Amit Levy，Joel Martin，Satoshi Matsushita，Oleg Pesok，David Ramos，Robbert van Renesse，Mendel Rosenblum，Nicolas Schiper，Deian Stefan，Andrew Stone，Ryan Stutsman，David Terei，Stephen Yang，Matei Zaharia 以及 24 位匿名的会议审查人员（可能有重复），并且特别感谢我们的领导人 Eddie Kohler。Werner Vogels 发了一条早期草稿链接的推特，给 Raft 带来了极大的关注。我们的工作由 Gigascale 系统研究中心和 Multiscale 系统研究中心给予支持，这两个研究中心由关注中心研究程序资金支持，一个是半导体研究公司的程序，由 STARnet 支持，一个半导体研究公司的程序由 MARCO 和 DARPA 支持，在国家科学基金会的 0963859 号批准，并且获得了来自 Facebook，Google，Mellanox，NEC，NetApp，SAP 和 Samsung 的支持。Diego Ongaro 由 Junglee 公司，斯坦福的毕业团体支持。

8.1.14 参考

略

8.2 calvin

卡尔文：快速分布式事务的分区数据库系统

8.2.1 摘要

许多分布式存储系统通过分区和复制实现了高吞吐量的数据访问，每个系统都有自己的优势和权衡。但是，为了实现高可扩展性，当今的系统通常会减少事务支持，不允许单个事务跨越多个分区。Calvin 是一个实用的事务调度和数据复制层，它使用确定性排序保证显著降低与分布式事务一起处理通常令人望而却步的争用成本。与以前的确定性数据库系统原型不同，Calvin 支持基于磁盘的存储，在商品计算机群集上近乎线性地扩展，并且没有单点故障。通过复制事务输入而不是效果，Calvin 还能够支持多个一致性级别（包括跨地理距离副本的 Paxos 基础强一致性），而无需事务吞吐量成本。

8.2.2 数据和定义

C.2.4[Distributed Systems]: 分布式数据库

H.2.4[Database Management]: 系统-并发，分布式数据库，事务处理

8.2.3 关键字

确定性、分布式数据库系统、复制、事务处理

8.2.4 背景和简介

分布式数据库系统设计的几种当前趋势之一是放弃支持传统的 ACID 数据库事务。一些系统，如亚马逊的 Dynamo[13]，MongoDB[24]，CouchDB[6] 和 Cassandra[17] 不提供任何事务支持。其他事务仅提供有限的事务性，例如单行事务更新（例如 Bigtable[11]）或访问仅限于数据库小子集（例如 Azure[9]、Megastore[7] 和 Oracle NoSQL Database[26]）的事务。这些系统不支持完全 ACID 事务的主要原因是提供线性外向可扩展性。其他系统（例如 VoltDB [27，16]）支持完整的 ACID，但在处理访问跨多个分区数据的事务时停止（或限制）并发事务执行。

减少事务支持大大简化了构建线性可扩展分布式存储解决方案的任务，这些解决方案旨在为”令人尴尬的可分区”应用服务。但是，对于不容易分区的应用程序，确保原子性和隔离性的负担通常由应用程序程序员承担，从而导致代码复杂性增加、应用程序开发速度变慢以及客户端事务调度性能低。

Calvin 旨在与非事务存储系统一起运行，将其转换为提供高可用性和完整 ACID 事务的共享无（近）线性可扩展的数据库系统。这些事务可能跨跨共享无群集的多个分区。Calvin 通过提供存储系统上方的层来完成此目的，该层用于处理分布式事务的调度以及系统中的复制和网络通信。在面对分布式事务时允许可伸缩性的关键技术特性是能够消除分布式提交协议的确定性锁定机制。

8.2.4.1 分布式事务的成本

分布式事务一直以来都是由数据库社区按照 1980 年代 System R*[22]* 的架构所倡导的方式实现的。*System R* 式分布式事务妨碍吞吐量和延长延迟的主要机制是要求所有参与的机器在提交时达成协议，以确保原子性和持久性。为了确保隔离，在此协议协议的整个期间，必须持有事务的所有锁，这通常是两阶段提交。

协议协议期间持有锁的问题在于，两阶段提交需要所有参与计算机之间的多个网络往返，因此运行协议所需的时间通常比执行所有本地事务逻辑所需的时间大得多。如果一些常用的记录经常涉及分布式事务，那么在这些记录上持有锁的额外时间会对总体事务吞吐量产生极其有害的影响。我们指的是事务保留其锁的总持续时间，我们将事务持有其锁的总持续时间（包括所需提交协议的持续时间）称为事务的争用空间。尽管本文中的大多数讨论都假定了悲观并发控制机制，但扩展事务的争用占用空间的成本在乐观模式中同样适用——由于级联中止的可能性，通常更糟。

对两阶段提交的某些优化（如将多个并发事务提交决策组合为单轮协议）可以降低两阶段提交中的 CPU 和网络开销，但不会降低其争用成本。虽然检测和纠正死锁通常不会带来令人望而却步的系统开销，但它会导致事务中止和重新启动，从而在某种程度上增加延迟并降低吞吐量。

8.2.4.2 副本一致性

分布式数据库系统设计的第二个趋势是减少复制的一致性保证。Dynamo、SimpleDB、Cassandra、Voldemort、Riak 和 PNUTS 等系统都降低了复制数据的一致性保证 [13，1，17，2，3，12]。降低这些系统的副本一致性的典型原因是 CAP 定理 [5，14]，为了使系统实现 24/7 全局可用性。即使在一个网络分区时仍然可用，系统必须提供较低的一致性保证。然而，去年，这种趋势开始逆转——部分原因可能是全球信息基础设施的不断完善，使得部分网络分区越来越罕见——有几个新系统支持强一致的复制。例如，谷歌的 Megastore[7] 和 IBM 的 Spinnaker[25], 通过 Paxos[18，19] 实现副本同步。

同步更新具有一个低延迟成本的一致性协议，这依赖于副本之间的网络延迟。这种成本可能很大，因为副本在地理上通常分开，以减少相关的故障可能。但是，这本质上是一种延迟成本，不一定影响争用或吞吐量。

8.2.4.3 在不增加争用的情况下达成协议

Calvin 的低成本分布式事务和同步副本方法如下：当多台计算机需要就如何处理特定事务达成一致时，它们在事务边界之外（即获取锁并开始执行事务之前）执行。

达成如何处理事务的协议后，必须根据计划节点失败执行到完成，相关问题不会导致事务中止。如果节点发生故障，它将从并行执行同一计划的副本中恢复，或者，它可以重播该节点的计划活动的历史记录。并行计划执行和计划历史记录重播都需要活动计划来确定性的，否则副本可能会出现分歧或历史记录可能会错误地重复。

为了支持这种确定性保证，同时在事务执行中最大化并发性，Calvin 使用基于我们在以前工作中介绍的确定性锁协议 [28]。

由于所有 Calvin 节点都就要尝试事务按什么顺序达成协议，因此它能够完全避开分布式提交协议，从而减少分布式事务的争用占用空间，尽管存在多分区事务，但几乎以线性方式进行扩展。我们的实验表明，在高争用工作负载下，Calvin 明显优于传统的分布式数据库设计。我们发现，在亚马逊云中的一个商用集群上每秒可以运行 50 万次 TPC-C 交易，这可以与目前在 TPC-C 网站上发布的在更高端的硬件上获得的世界记录竞争。

这篇论文的要点如下：

- 事务调度和数据复制层的设计，将非事务存储系统转换为近线性扩展的 shared-nothing 的数据库系统，该系统可提供高可用性、强一致性和完整的 ACID 事务。
- 确定性并发控制协议的具体实现，它比以前的方法更具可扩展性，并且不会引发潜在的单点故障。
- 一种数据预提取机制，利用事务执行之前执行的计划阶段，允许事务对驻留在磁盘上的数据进行操作，而无需在整个磁盘查找期间扩展事务的争用空间。
- 快速检查点方案，加上 Calvin 的决定论保证，完全消除了对物理重做日志及其相关开销的需要。

下面将继续讨论确定性数据库的背景。在第 3 节中，将介绍 calvin 的架构体系。在第 4 节中，将介绍 calvin 如何处理访问磁盘数据的事务。在第 5 节，将介绍 calvin 定期实现完整数据库快照的机制。在第 6 节，我们设计了一系列的实验，用来衡量 calvin 在不同负载下的吞吐量和延迟。我们在第 7 节介绍相关的工作，在第 8 节讨论未来的工作，并在第 9 节结束。

8.2.5 确定性数据库系统

在传统的（System R* 类型）分布式数据库系统中，提交分布式事务时需要共识协议，主要原因是确保事务的所有影响都以原子方式成功实现持久存储 - 无论是涉及事务的所有节点都同意” 提交” 其本地更改，还是没有一个节点同意。阻止节点提交其本地更改（因此导致整个事务中止）的事件分为两类：非确定性事件（如节点故障）和确定性事件（例如，如果库存物料低于零，则强制中止的事务逻辑）。

事务因任何非确定性事件而必须中止，没有任何根本原因; 当系统选择由于外部事件而中止事务时，则是出于实际考虑。毕竟，强制系统中的所有其他节点等待出现非确定性事件（如硬件故障）的节点恢复，可能会使系统长期保持等待状态。

但是，如果存在副本，没有与失败节点并行执行完全相同的操作，则依赖于与受事件影响节点通信以执行事务的其他节点无需等待失败的节点恢复到其原始状态，而是可以请求副本节点处理当前或将来事务所需的任何数据。此外，事务可以提交，因为副本节点能够完成事务，并且失败的节点最终将能够在恢复时共同复制事务。

因此，如果存在一个副本，该副本正在与遇到非确定性故障的节点并行处理相同的事务，则无需在出现此类故障时中止事务。惟一的问题是，副本需要经过相同的数据库状态序列，以便在事务的中间立即替换失败的节点。同步复制每个数据库状态更改的开销太大，无法实现。相反，确定性数据库系统同步复制批量事务请求在传统的数据库实现中, 简单的复制事务输入通常不足以确保副本不偏离, 因为数据库保证他们将处理事务的方式在逻辑上等同于一些串行事务的顺序输入, 而且这两个副本可能会选择过程相当于不同的串行输入订单, 例如由于不同的线程调度, 网络延迟或其他硬件约束。但是，如果修改数据库的并发控制层，按商定的事务输入（并且对数据库进行其他几个次要修改 [28]）的顺序获取锁，则可以制作所有副本来模拟相同的串行执行顺序，并且可以保证数据库状态不会偏离。

这种确定性数据库允许两个副本通过复制数据库输入来保持一致，如上所述，这些主动复制节点的存在使分布式事务能够在存在非确定性故障时提交其工作（这可能发生在事务中间）。这就消除了在分布式事务结束时使用共识协议的主要理由（需要检查可能导致事务中止的节点故障）。上述中止的另一个潜在原因, 事务中的确定性逻辑（例如，如果库存为零，应中止事务）, 不一定必须作为在事务结束时执行共识协议的一部分。相反，事务中涉及的每个节点都等待来自每个节点的单向消息，该消息可能确定地中止事务，并且只有在收到这些消息后才提交。

8.2.6 系统架构

Calvin 设计为作为可扩展的事务层，高于实现基本 CRUD 接口（创建/插入、读取、更新和删除）的任何存储系统之上。尽管可以运行在分布式非事务存储系统（如 SimpleDB 或 Cassandra）上运行 Calvin，如果存储系统不是开箱即用的，那么解释 Calvin 的架构就更直接了。例如，存储系统可能是安装在多台独立计算机上的单节点键值存储。在此配置中，Calvin 将数据分区组织到每个节点上的存储系统中，并协调事务执行过程中节点之间必须发生的所有网络通信。

在图 1 中介绍了 calvin 的整体架构，calvin 本质上将系统分为了三个独立的处理层：

- 排序层（sequencer）拦截事务性输入，并将它们放入全局事务性输入序列中——这个序列将是所有副本在执行期间确保串行等价的事务的顺序。因此，序列器也处理这个输入序列的复制和记录。
- 调度层（scheduler）使用确定性锁方案编排事务执行，以保证与序列层指定的串行顺序等价，同时允许事务由一个事务执行线程池并发执行。(尽管它们显示在图 1 中的调度器组件下面，但这些执行线程在概念上属于调度层。)
- 存储层处理所有物理数据存储布局。Calvin 事务使用简单的 CRUD 接口访问数据; 任何支持类似接口的存储引擎都可以很容易地插入到 Calvin 中。

Figure 1: System Architecture of Calvin

这三个层都是水平伸缩的，它们的功能在一个无共享节点集群中分区。部署中的每个节点通常运行每个层的一个分区。(图 1 中的灰框表示群集中的物理计算机)。将在下边章节讨论这三层的实现。

通过将复制机制、事务功能和并发控制 (在排序层和调度层中) 与存储层分离，calvin 的设计与高度单一的传统数据库设计明显不同，传统数据库设计高度单一，物理访问方法、缓冲区管理器、锁管理器和日志管理器高度集成并相互依赖。这种解耦使得很难实现某些流行的恢复和并发技术，例如基于物理日志的 ARIES 算法和 next-key locking 技术来处理幻像（handle phantoms）（即在并发控制内的逻辑属性上使用物理代理）。Calvin 并不是试图分离数据库中的事务组件与其他组件的唯一尝试。由于云计算的出现和它带来的高度模块化的服务，

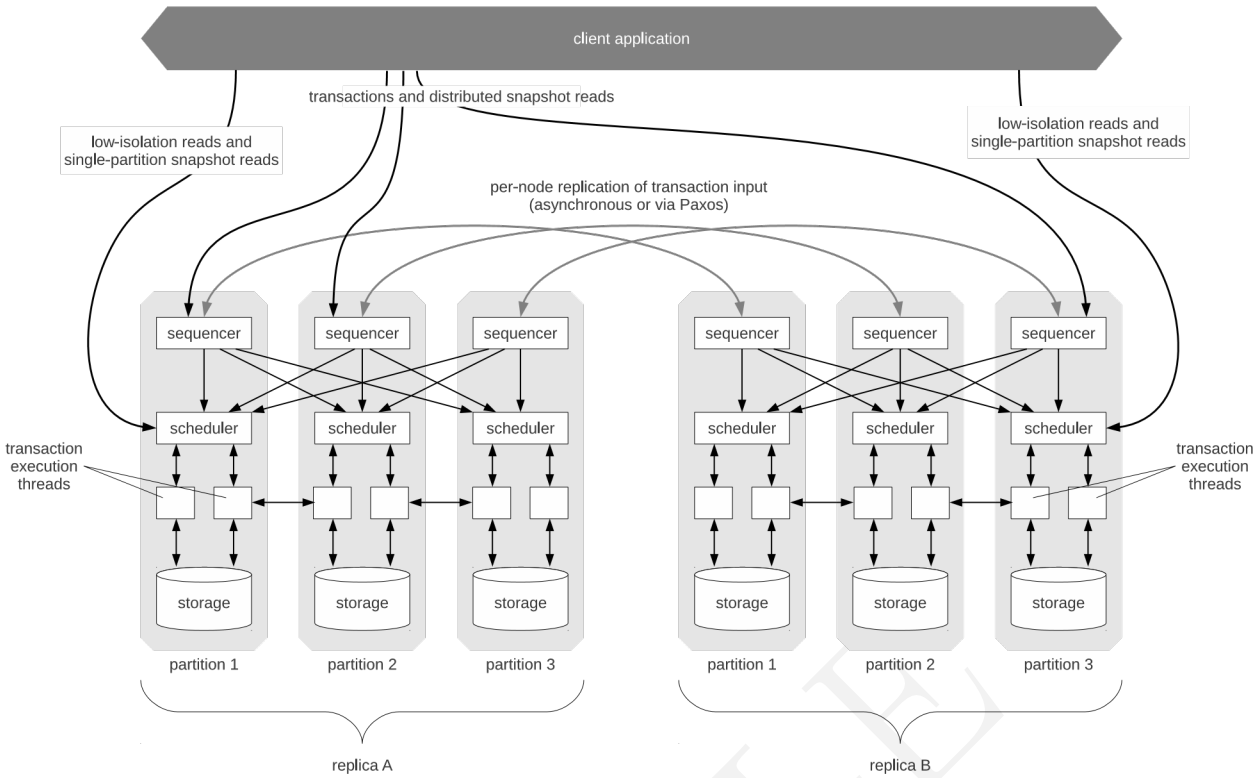


图 122: Figure1_System_Architecture_of_Calvin

数据库社区对将这些功能分离成不同的模块化系统组件重新产生了兴趣 [21]。

8.2.6.1 序列和复制

在以前与确定性数据库系统的工作中，我们实现了排序层功能作为一个简单的回声（echo）服务-一个单独的节点接受事务请求，记录到磁盘，并按时间戳顺序转发到适当数据库节点的每个副本中。单节点排序器的问题有:

(a) 它们表示潜在的单点故障

(b) 随着系统的增长，单节点排序器的恒定吞吐量绑定使整个系统可伸缩性快读降低。

Calvin 的排序层分布在整个系统副本中，并且在每个副本中的每台计算机上分区。

Calvin 将时间划分为以 10 毫秒为单位的时间段，在此期间，每个机器的序列器组件会收集来自客户端的事务请求。在这个时间段结束时，到达序列器节点的所有请求都被编译为批处理。这是事务输入副本发生的点（下面讨论）。

成功复制序列器的批处理后，它会向其副本中每个分区上的调度程序发送一条消息，其中包含 (1) 序列器的唯一节点 ID (2) 纪元编号 epoch number(10ms 同步，在整个系统中递增)。(3) 接收者需要参与的所有事务输入。这允许每个调度器通过（确定性 deterministic、循环的方式 round-robin）这个纪元（epoch）的所有排序的批来拼凑自己的全局事务顺序视图。

8.2.6.1.1 同步和异步复制

Calvin 目前支持两种复制事务的输入模式：异步复制和 Paxos-based 的同步复制。在这两种模式下，节点被组织成复制组。每个复制组都包含特定分区的所有副本。例如，在图 1 的部署中，副本 A(replica A) 中的分区 1 (partition 1) 和副本 B(replica B) 中的分区 1（partition 1）共同组成一个复制组。

在异步复制模式下，一个副本被指定为主副本，所有事务请求将立即转发到位于此副本节点的排序器。编译每个批处理后，每个主副本排序器组件将批处理转发到其复制组中的所有其他从序列器上。这样做的优点是，在

主副本事务开始执行之前，延迟非常低，但故障转移的复杂性会很大。在主排序器发生故障时，必须在同一副本中的所有节点与故障节点复制组上的所有成员之间达成一致，涉及（a）哪个批处理是失败的排序器发送的最后一个有效批处理，（b）批包含的事务有哪些，因为每个调度器仅发送它实际需要执行的每个批处理的部分视图。

Calvin 还支持基于 Paxos 的事务输入同步复制。在这个模式下，复制组内的所有序列器都使用 Paxos 就每个纪元事务的组合批处理达成一致。Calvin 目前实现使用 Zookeeper，这是一种高可靠的分布式协调服务，通常为分布式数据库系统用于检测信号、配置同步和命名服务 [15]。Zookeeper 未针对存储高数据量进行优化，并且可能会产生比最有效的 Paxos 实现更高的延迟。但是，对于本文运行的所有试验，Zookeeper 会处理必要的吞吐量来复制 Calvin 的事务性输入，并且由于这个同步步骤不会扩展争用，事务性吞吐量完全不受这个预处理步骤的影响。同时实施 Calvin 序列器之间的更简化 Paxos 协议来改进 Calvin 代码库，而不是使用 Zookeeper 开箱即用的协议，对于延迟敏感的应用程序可能很有用，但不会提高事务吞吐量。

Figure 2: Average transaction latency under Calvin' s different replication modes.

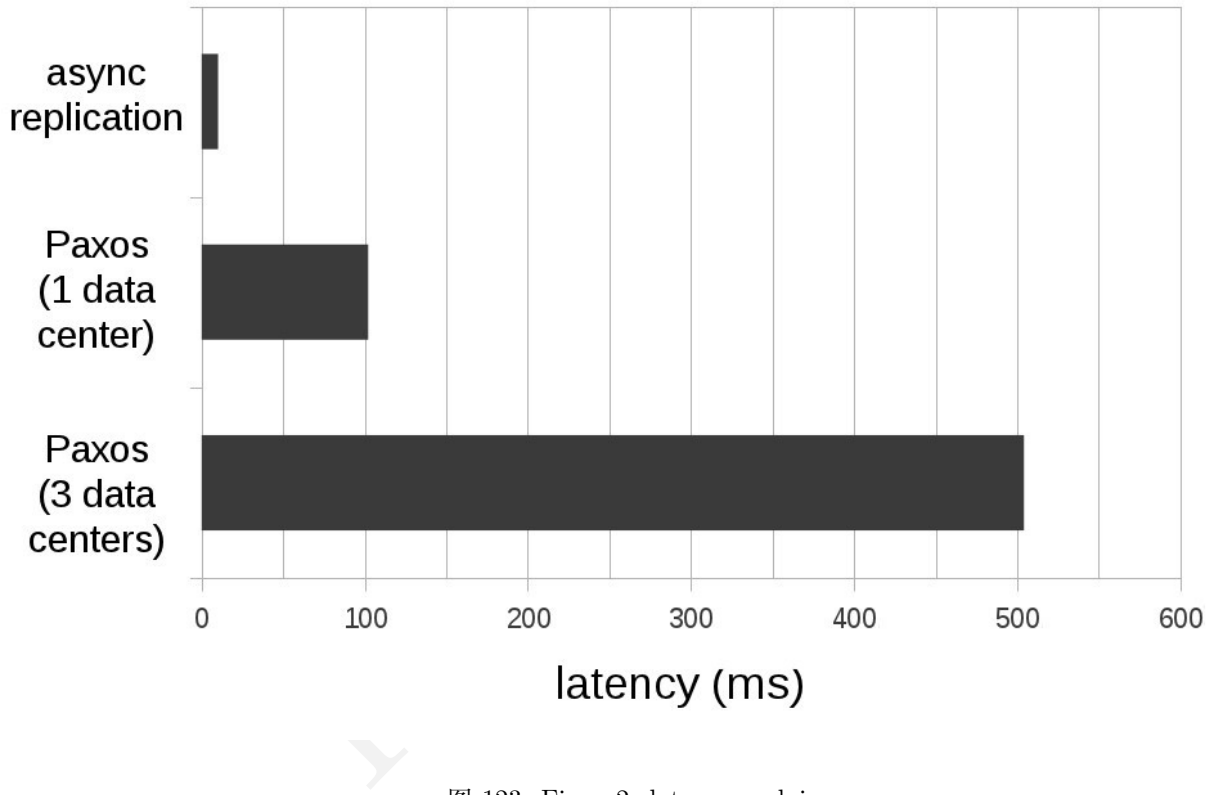


图 123: Figure2_latency_calvin

图 2 可以看到在不同复制模式下，Calvin 的平均事务延迟。上述数据是使用 4 台 EC2 High-CPU 机器每个副本，运行 40000 microbenchmark 事务每秒（10000 每节点），其中 10% 是多分区（关于实验附加内容请参考第 6 章）。报告的两个 Paxos 延迟都使用了三副本（共 12 个节点）。当所有副本在一个数据中心运行时，副本之间的 ping 时间约为 1ms。跨数据中心复制时，一个在亚马逊云的美国东数据中心，一个在亚马逊云的美国西数据中心，一个亚马逊的 EU 数据中心。它们之间的 ping 值在 100 到 170ms 之间。更改 Calvin 的复制模式不会影响总事务吞吐量。

8.2.6.2 调度和并发控制

当数据库系统的事务组件从存储组件中拆解时，它不能再对数据层的物理更新进行任何假设，也不能引用物理数据结构（如页面和索引），也不能知道事务对数据库中数据的物理布局的副作用。日志记录和并发协议必须具有逻辑性，仅指记录键，而不是物理数据结构。幸运的是，在确定性数据库系统中，无法执行物理日志记录并不是一个问题; 由于数据库的状态可以从数据库输入中确定，因此逻辑日志记录非常简单（输入由序列层记录，检查点由存储层记录 - 请参阅第 5 章，以进一步讨论 Calvin 中的检查点）。

尽管对于并发控制来说，只访问逻辑记录，出现问题的概率略高，因为锁定键的范围和对幻像更新的健壮性通常需要对数据进行物理访问。为了处理这个问题，Calvin 可以使用最近提出的另一种用于非绑定数据库系统的方法，即创建可以在事务层 [20] 中逻辑锁定虚拟资源，不过这个特定的实现仍需要未来的工作。

Calvin 的确定性锁管理器是整个调度层的一部分，每个节点的调度程序只负责锁定存储在该节点存储层中的记录，即使存在访问其他节点记录的事务。锁协议类似于严格的两阶段锁，但是增加了两个变量：

- 对于任何一对事务 A 和事务 B 都请求某个本地记录 R 上的排它锁，如果事务 A 按照排序层提供的顺序出现在 B 之前，那么 A 必须在 B 之前请求对 R 的锁。在实践中，Calvin 通过在单个线程中序列化所有锁请求来实现这一点。线程扫描序列层发送的串行事务顺序; 对于每个条目，它请求事务在其生命周期中需要的所有锁。(因此，所有事务都必须提前声明其完整的读/写集; 第 3.2.1 节讨论了其中的局限性。)
- 锁管理器必须严格按照事务请求锁的顺序向请求事务授予每个锁。因此在上面的例子中，B 在 R 上的锁不能被授予，直到 A 获得了 R 上的锁，执行到完成并释放了锁。

客户端指定事务逻辑使用基本的 CRUD 接口访问数据的 C++ 接口函数。事务代码不需要完全了解分区（尽管用户可以指定在计算机之间应如何对键进行分区），因为 Calvin 截获事务代码中出现的所有数据访问，并自动执行所有远程读取结果转发。

一旦一个事务在这个协议下获得了它的所有锁（因此可以完全安全地执行），它就会移交给一个工作线程执行。每个工作线程实际执行的事务分为五个阶段：

1. **读/写集分析**。事务执行读取在提交事务请求时执行的第一件事是分析事务的读写集，注意（a）本地存储的读写集的元素（即线程正在执行的节点），以及（b）存储写入元素集参与的节点集，这些节点在事务中称为活动参与者；仅存储读取集元素的参与者节点称为被动参与者。
2. **执行本地读取**。接下来，工作线程将访问本地存储中读取集中所记录的值。根据存储接口的不同，这可能意味着将记录的副本复制到本地缓存层，或者只是保存指向记录可以找到内存位置的指针。
3. **提供远程读取服务**。本地读取阶段的所有结果都转发到每个积极参与节点上对应的工作线程。由于被动参与者不修改任何数据，他们不需要执行实际的事务代码，并且不需要收集任何远程读取结果。如果辅助线程在被动参与节点上执行，则在此阶段之后完成。
4. **收集远程读取结果**。如果辅助线程在积极参与的节点上执行，则它必须执行事务代码，因此它必须首先获取所有读取结果，本地读取的结果（在第二阶段获得）和远程读取结果（由每个参与节点在第三阶段适时转发）。在此阶段中，工作线程收集后一组读取结果。
5. **事务逻辑执行和应用写入**。工作线程收集所有读取结果后，继续执行所有事务逻辑，应用任何本地写。可以忽略非本地写入，因为它们将被相应的节点上对等事务执行线程视为本地写入，并该节点执行。

假设分布式事务在每个参与节点上几乎同时开始执行（并不总是这样-第 6 章将对此进行更详细的论述），则所有读取都并行执行，并且所有远程读取结果也并行执行，不同节点的工作线程无需在事务执行时彼此请求数据。

8.2.6.2.1 依赖事务

必须执行读取才能确定其完整读/写集（我们术语中，dependent transcations 称为依赖事务）事务在 Calvin 中不受本机支持，因为 Calvin 的确定性锁协议要求在事务开始执行之前预先了解所有事务的读写集。相反，Calvin 支持一种称为乐观锁位置预测（OLLP）的方案，它可以通过修改客户端事务代码本身 [28] 以极低的开销成本实现。其理念是，依赖事务之前应有一个廉价、低隔离、不重复、只读侦查查询，该查询执行所有必要的读操作以发现事务的完整读/写集，然后发送实际事务到全局序列中执行，使用侦查查询的结果读/写。因为它是侦查查询的记录读（实际事务的读/写集）之间发生了变化的执行侦查查询和实际执行事务，读取结果必须重新核对，如果侦查读写集不再有效，进程必须（确定地）重新启动。

在此类事务中特别常见的是那些必须执行二级索引查找以确定其完整读/写集的事务。由于二级索引的修改成本通常相对较高，因此他们很少保留在值更新频率极高的字段中。例如，关于“存货项目名称”或“纽约证券交易所股票代号”的二级索引将是常见的，而在诸如“存货项目数量”或“纽约证券交易所股票价格”等不稳定的字

段上创建二级索引将是不常见的。因此，在大多数常见的现实工作负载下，我们很少期望 OLLP 方案导致重复的事务重新启动。

TPCC 基准测试中的“付款”事务类型是此类事务的示例。由于 TPCC 工作负载从未修改支付转换操作的读写集可能依赖的索引，因此，使用 OOLP 时不必重新启动付款事务。

8.2.7 基于磁盘存储的 Calvin

之前关于确定性数据库系统的工作附带了一个警告，即确定性执行仅适用于完全驻留主内存中的数据库 [28]。其理由是，确定性数据库系统相对于传统非确定性数据库系统的一个主要缺点是，非确定性数据库系统能够保证与任何串行排序等价，因此可以任意重新排序事务，而像 Calvin 这样的系统则受制于排序器选择的顺序。

> For example, if a transaction (let’ s call it A) is stalled waiting for a disk access, a traditional system would be able to run other transactions (B and C, say) that do not conflict with the locks already held by A. If B and C’ s write sets overlapped with A’ s on keys that A has not yet locked, then execution can proceed in manner equivalent to the serial order B—C—A rather than A—B—C. In a deterministic system, however, B and C would have to block until A completed. Worse yet, other transactions that conflicted with B and C—but not with A—would also get stuck behind A. On-the-fly reordering is therefore highly effective at maximizing resource utilization in systems where disk stalls upwards of 10 ms may occur frequently during transaction execution.

例如，如果一个事务（事务 A）在等待磁盘访问时停滞，则传统数据库能运行与已持有锁不冲突的其他事务（事务 B 和事务 C），如果 B 和 C 的写入集和 A 尚未锁定的键重叠，则执行可以以等效于排序 A-B-C 的 B-C-A 的方式执行。但是，在确定性系统中，B 和 C 必须阻塞，直到 A 完成。更糟的是，其他与 B 和 C 有冲突但和 A 没有冲突的事务也会落后于 A。因此，在事务执行过程中磁盘暂停时间经常超过 10 毫秒的系统中，动态重新排序对于最大限度地利用资源非常有效。

Calvin 遵循这个指导设计原则，避免了确定性在基于磁盘数据库上下文中的这种缺点：在获得锁之前，尽可能将繁重的任务移动到事务处理管道的较早位置。

当序列器组件收到出现磁盘停滞的事务请求时，它会在事务请求转发到调度层之前引入人为的延迟转发，同时向相关存储组件“warm up”的磁盘上访问驻留记录。如果人工延迟大于或等于在执行期间执行磁盘驻留记录放入内存所花费的时间，那么当事务实际执行时，它将只访问内存缓存记录。注意，这个方案的事务总延迟不应该大于在传统系统的磁盘 IO 进行执行期间 (因为在这两种情况下都发生完全相同的磁盘操作集)，但任何磁盘延迟都没有增加事务的争用。

为了清楚地演示此技术的适用性（和缺陷），我们为 Calvin 实现了一个简单的基于磁盘的存储系统，其中冷数据被写入本地文件系统，并且仅在事务需要时读取到内存驻留键表。当运行每台 10000microbenchmark 事务时（有关实验的详细信息请参考第 6 章）,Calvin 的总事务吞吐量不受访问磁盘存储的事务存在的影响，只要磁盘的事务不超过 0.9 （10000 个事务中的 90 个）。但是，这个数字取决于使用的服务器的特定硬件配置。我们运行了低端硬件的实验，发现可支持的磁盘访问事务的数量受本地磁盘最大吞吐量（不是争用占用空间）的有效限制。由于 microbenchmark 工作负载涉及随机访问需要不同的文件，因此每台机器每秒 90 个磁盘访问事务足以将磁盘随机访问吞吐量变成瓶颈。使用高端磁盘 (或闪存)，可以支持更多的基于磁盘的事务，而不会影响总的吞吐量。

为了更好的理解 Calvin 与其他磁盘配置、闪存、网络存储等对接的潜力，我们还实现了一个存储引擎，其中冷数据存储在单独计算机的内存中，该计算机只能在预先指定的延迟（模拟网络或存储访问延迟）后才为数据请求服务。使用这个设置，我们发现每台计算机都能支持相同的负载，即每秒 10000 个事务，无论有多少这些事务访问过冷数据，及时在极高的争用项（争用指数 contention index0.01)。

我们发现在协调确定性执行与基于磁盘的存储方面存在两个主要挑战。首先，必须准确预测磁盘延迟，以便事务延迟到适当的时间量。其次，序列器层必须准确追踪所有存储节点中哪些键存在内存中，以便确定合适需要预取。

8.2.7.1 磁盘 I/O 延迟预测

准确预测从磁盘到内存获取记录所需要的的时间并非易事。读取磁盘驻留服务器所占用的时间可能因多原因而有很大差异：

- 头部和主轴移动的可变物理距离
- 之前队列磁盘的 IO 操作
- 远程读取的网络延迟
- 故障转移
- 由于遍历基于磁盘的数据结构（例如树）需要执行多个 IO 操作

因此，无法完美地预测延迟，任何启发式的使用有时会导致低估，有时还会导致高估。当调整 Calvin 在高争用条件下对磁盘驻留数据执行良好性能时，磁盘 IO 延迟被证明是一个特别有趣和关键的参数。我们发现，如果序列器选择保守的高估计值，并且延迟转发事务的时间超过可能必要的时间，则磁盘访问导致的争用成本将最小化（因为提取几乎总是在事务需要读取记录之前完成），但代价是总体事务延迟。过高的估计值还可能导致存储系统的内存超载，等待请求安排的事务变成”冷”记录。

但是，如果序列器低估了磁盘 I/O 延迟，并且没有将事务延迟足够长的时间，则它将被安排得太快并在执行期间停止，直到所有读取完成。由于锁的保持时间较长，为了保持高吞吐量，可能需要很高的争用占用空间成本。

所以，在评估磁盘 IO 延迟时，总事务延迟和争用之间存在根本的权衡。在上述两项实验中，我们调整了延迟预测，使至少 99% 的磁盘访问事务在其相应的预取请求完成后被调度。使用基于文件系统的简单存储引擎，这意味着引入 40ms 的人工延迟，但即使在非常高的争用（争用指数 0.01）下，这足以维持吞吐量。在较低争用（争用指数 0.001）下，我们发现，除了将事务请求收集成批 (平均 5 毫秒) 所导致的默认延迟之外，没有必要延迟。对这个特殊的延迟争用折衷进行更详尽的探索将是未来研究的一个有趣方向，特别是当我们进一步试验将 Calvin 连接到各种商业可用的存储引擎时。

8.2.7.2 全局热点记录追踪

为了使序列器在预热读取集时准确确定哪些事务要延迟调度，每个节点的序列器组件必须跟组系统中当前内存中的数据，而不仅仅是由序列器节点上的存储组件管理的数据。虽然这对于本文中的实验是可行的，但这不是一个可扩展的解决方案。如果每个排序都未追踪热键的全局列表，一个解决方案是将所有事务从计划时间延迟到允许允许有足够时间进行预取。这可防止磁盘寻求扩展争用占用空间，但会在每个事务中产生延迟。另一个解决方案（仅适用于单分区事务）是让计划程序跨所有副本同步追踪其本地热数据，然后允许调度程度确定地决定延迟请求单分区事务（尝试读取冷数据）的锁。更全面的探索这一战略，包括研究如何为多部门事务实施该战略，仍是今后的工作。

8.2.8 检查点

确定性数据库系统有两个属性，可以简化确保容错的任务。首先，主动复制允许客户端在发生崩溃时立即将故障转移到另一个副本。

其次，只记录事务性输入—不需要支付物理重做日志的开销。重放事务性输入的历史记录足以将数据库系统恢复到当前状态。但是，在每次失败时都重放数据库从一开始就有的整个历史是低效的 (而且荒谬的)。相反，Calvin 定期采用完整数据库状态的检查点，以提供恢复期间开始重播的起点。

Calvin 支持三种检查点模式: 朴素同步检查点，an asynchronous variation of Cao et al.’ s Zig-Zag algorithm [10]，以及只有在存储层支持完全多版本化时才支持的异步快照模式。

第一种模式使用活动复制系统中固有的冗余来创建系统检查点。系统可以定期冻结整个副本，并生成系统的完整版本快照。由于这一次只在一个快照上发生，因此客户端看不到副本不可用的时间段。这种方法的一个问题是，接受检查点的副本可能明显落后于其他副本，如果由于另一个副本的硬件故障而调用该副本，则可能出现问题。此外，副本可能会花费大量时间来赶上其他副本，特别是在负载沉重的系统中。

Calvin 的第二检查点模式是紧密基于 Cao 等人的 Zig-Zag 算法 [10]。Zig-Zag 在给定的数据存储中存储每条记录的两个副本，分别为 AS[K]0 和 AS[K]1，加上每条记录另外两个比特，MR[K] 和 MWK。MR[K] 指定从数据库中读取记录 K 时应该使用哪个记录版本，而 MW[K] 指定更新记录 K 时要覆盖哪个版本。所以记录 K 的新值总是写为 AS[K]MW[K]，每次更新 K 时 MR[K] 等于 MW[K]。

Zig-Zag 的每个检查点周期开始于在数据库完全停顿的物理一致性点上，将数据库中所有密钥 K 的 MW[K] 设置为 ¬MR[K]。因此，AS[K]MW[K] 总是存储记录的最新版本，而 AS[K]MW[K] 总是存储最近检查点开始之前写入的最后一个值。因此，异步检查点线程可以通过每个键 K，以 AS[K]]¬MW[K] 的形式登录到磁盘，而不必担心记录被打破了。

利用 Calvin 的全局串行顺序，我们实现了一个曲折的变体，它不需要静默数据库来创建物理一致性点。相反，Calvin 捕获关于虚拟一致性点的快照，虚拟一致性点只是全局串行顺序中的一个预先指定的点。当一个虚拟点的一致性方法,Calvin 的存储层开始保持两个版本的每个记录存储系统中的“之前”的版本，这只能通过交易之前更新虚拟点的一致性，和一个“后”版本，由事务写入后出现虚拟点的一致性。一旦一致性虚拟点之前的所有事务都完成执行，每个记录的“前”版本就有效地不可变了，一个异步检查点线程就可以开始将它们检查点指向磁盘。一旦检查点完成，任何复制版本都将被垃圾收集: 所有同时拥有“之前”版本和“之后”版本的记录将丢弃它们的“之前”版本，这样在下一次检查点开始之前，每个版本只保留一条记录。

上面描述的 Calvin 的第一个检查点模式涉及到在检查点期间完全停止事务执行，而在异步检查点线程处于活动状态时，该模式只会带来适度的开销。图 3 显示了 Calvin 在一个典型的检查点捕获期间的最大吞吐量。这个测量是在运行微基准测试的单机 Calvin 部署上进行的，该部署处于低争用状态 (有关我们的实验设置的更多信息，请参阅第 6 节)。虽然由于 (a) 获取检查点的 CPU 成本和 (b) 访问记录时少量的锁存争用，总吞吐量有所降低，但异步地向存储写入稳定值不会增加锁争用或事务延迟。

Calvin 还能够利用存储引擎，这些引擎显式地跟踪除了当前版本之外的每条记录的所有最新版本。多版本存储引擎允许在不获取任何锁的情况下执行只读查询，以增加内存使用为代价，减少总体争用和总的并发控制开销。在这种模式下运行时，Calvin 的检查点方案采用普通的“SELECT *”查询形式，对所有记录进行查询，查询结果被记录到磁盘上的一个文件中，而不是返回给客户端。

Figure 3: Throughput over time during a typical checkpointing period using Calvin’ s modified Zig-Zag scheme.

8.2.9 性能和可扩展性

为了研究 Calvin 在各种共同参数下的性能和可伸缩性特征，我们使用两个基准进行了许多实验：TPC-C benchmark 和 Microbenchmark，以便更好地控制基准参数的多样化。除非另有说明，否则所有实验均使用高 CPU/超大型实例在 Amazon EC2 上运行，这些实例承诺 7GB 内存和 20 个 EC2 计算单元—8 个虚拟内核，每个实例具有 2.5EC2 计算单元。

8.2.9.1 TPC-C benchmark

Figure 4: Total and per-node TPC-C (100% New Order) throughput, varying deployment size.

TPC-C 基准由几类事务组成，但大部分工作负载（包括几乎所有需要高度隔离的分布式事务）由”New Order”事务组成，该事务模拟客户在电子商务应用程序上下订单。由于实验的重点是分布式事务，因此我们的 TPCC 实现仅针对新订单事务进行。如果我们运行完整的 TPC-C 基准，我们期望获得类似的性能和可伸缩效果。

图 4 显示了总吞吐量和每台机器吞吐量（TPCC New Order 事务执行每秒）作为节点数关系的图，每个节点存

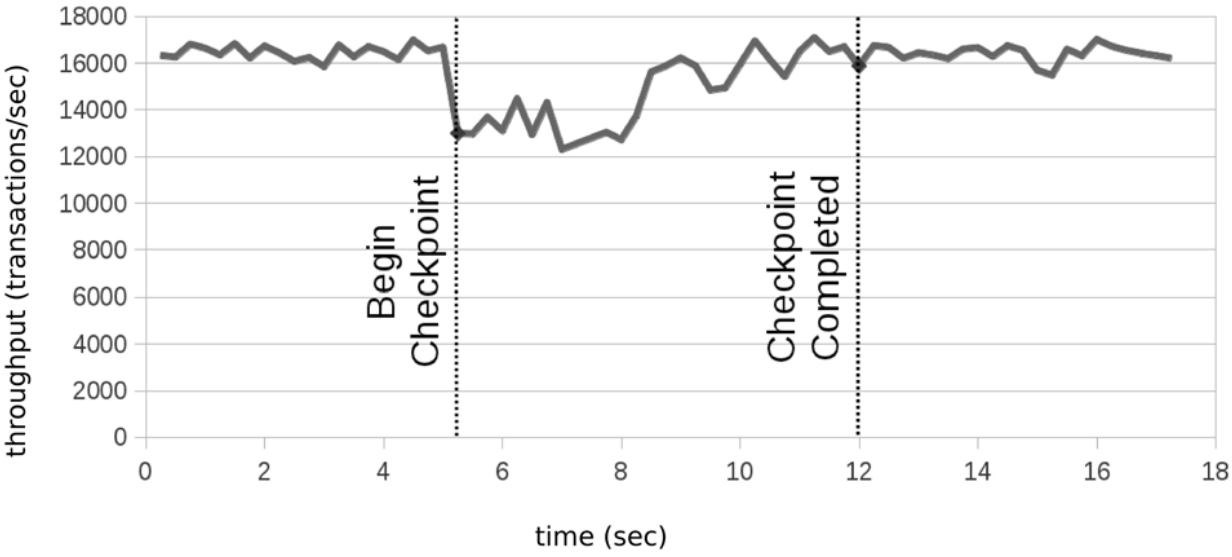


图 124: zig_zag

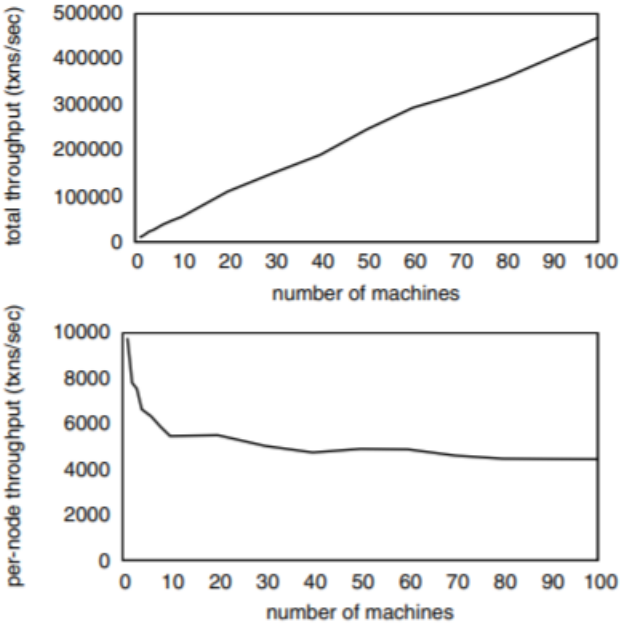


图 125: tpcc

储包括 10 个 TPCC 仓库的数据分区，为了充分体现 Calvin 对分布式事务的处理情况，多仓库新订单事务（约占新订单交易总数的 10%）始终访问第二个仓库，该仓库和第一个不在一台机器上。

由于每个分区包含 10 个 warehouse，并且 New Order 更新了某些仓库的 10 “districts”，因此最多可以在任何节点上同时执行 100 个新订单事务（因为每个分区的惟一区域不超过 100 个，而且每个新订单事务都需要一个区域上的独占锁）。因此，将锁的持有时间降至最低至关重要，因为系统的吞吐量受这个并发事务完成（释放锁）的速度限制，以便新事务可以获取该区域上的独占锁并开始启动。

如果 Calvin 在共识协议（如分布式新订单事务的两阶段提交）期间持有锁，则吞吐量将严重受限。（第 6.3 节提供了与实现两阶段提交的传统系统的详细比较。）如果没有共识协议，Calvin 能够在大于 10 个节点的集群中实现每个节点 5000 个事务每秒，并线性扩展。（下一节将描述 Calvin 在较小集群上每节点实现更多事务的原因。）因此，我们的 Calvin 实现能够在 100 个节点集群上实现近 50 万次 TPCC 事务/秒。指的注意的是，目前 TPCC 世界纪录的保持着，Oracle 运行 504161 新订单事务每秒。尽管运行在比我们用于实验的机器高得多的高端硬件 [4]。

6.2 Microbenchmark experiments

Figure 5: Total and per-node microbenchmark throughput, varying deployment size.

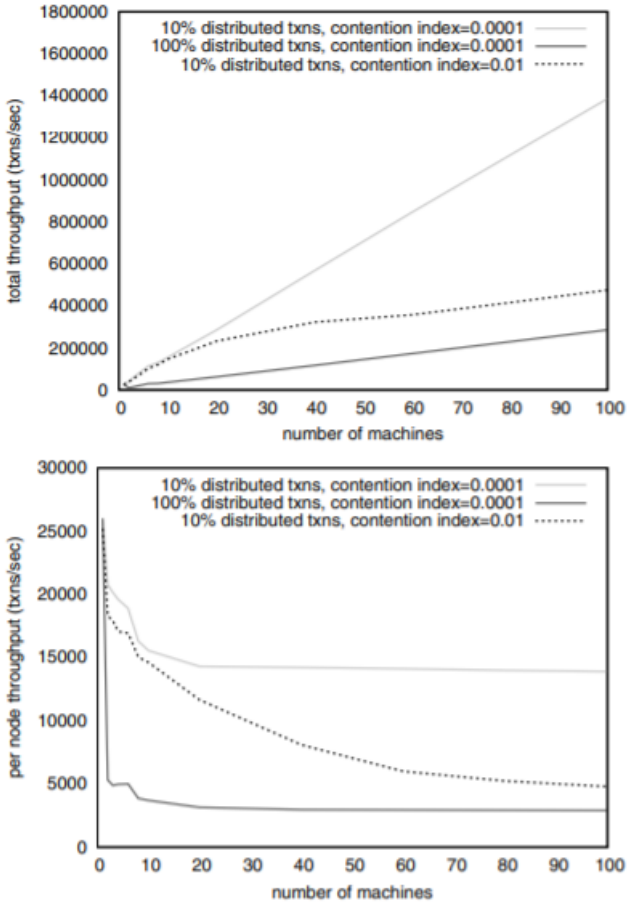


图 126: mircobenchmark

为了更准确地检查合并分布式事务和高争用时产生的成本，我们实现了一个微基准测试，该基准与 TPCC 的新订单事务共享某些特征，同时减少了总体开销并允许对工作负载进行更精细的调整。基准测试的每个事务读取 10 条记录，并对结果进行约束检查，仅在通过约束检查时更新每个记录上的计数器。在微基准事务所记录的 10 条记录中，一条是从一组热记录中选择的，其余记录是从一组大得多的记录中选择的，除非微基准事务跨越两

台机器，在这种情况下，它会在参与事务的每台机器上访问一个热记录。通过更改热记录的数量，我们可以微调争用。在随后的讨论中，我们使用术语争用指数（contention index）来引用在特定节点上执行事务时更新的热记录分数。例如，争用指数为 0.001 意味着每个事务从 1000 条热记录中选择要在每个参与节点上更新一条（即最多可以同时执行 1000 个事务），而争用指数 1 表示每个事务都涉及到热记录（即事务必须完全串行执行）。

图 5 显示我们将微基准缩放为 100 个 Calvin 节点的不同争用设置的实验，并且具有不同数量的分布式事务。在非常低争用下添加节点时，每个节点的吞吐量将下降到稳定量大约 10 台节点，然后保持不变，以线性方式对许多节点进行分析。在较高争用（争用指数 0.01 类似于 TPCC 争用级别），随着机器的增加，我们会看到更长的、更渐进的每节点吞吐量下降，更缓慢地接近一个稳定的量。

多种因素促成了 calvin 这种可伸缩曲线的形状。在所有情况下，一台节点和两台节点之间的急剧下降是每个多部分事务必须执行的额外工作的 CPU 成本的结果：

- 序列化和反序列化远程读取结果
- 等待获取远程读取结果的事务之间的上下文切换
- 在所有参与的节点上设置、执行和清理事务后，即使它只计算一次总吞吐量

在丢弃掉这类初始化后，随着节点的加入而进一步下降的原因（即使争用和参与分布式事务的节点数都保持不变）也相当微秒。假设，在高争用负载下，节点 A 开始执行需要从节点 B 远程读取分布式事务，但 B 尚未访问该事务（B 可能仍在处理序列中的早期事务，并且在序列中所有以前的事务获得锁之前，它无法处理事务）。节点 A 也许能够开始执行一些其他非冲突事务，但很快它只需要等待 B 赶上，就可以提交挂起的分布式事务并执行后续冲突事务。通过这种机制，任何特定节点的领先或落后是有限的，争用越高，它的极限越紧。随着机器的增加，会发生两件事：

- **Slow machines**。并不是所有 EC2 实例都等于高性能，有时一个 EC2 用户卡住。如图 5 中的实验结果，使用相同的 EC2 实例，所有三行都显示了 6 到 8 台机器之间突然下降，很明显缓慢机器的添加是在 6 到 8 节点的时候。
- **Execution progress skew**。由于许多因素，例如 OS 线程调度、可变的网络延迟和事务序列之间争用的随机变化，这些机器偶尔会略微领先或落后于其他机器。机器越多，就越有可能在任何时候至少有一台因为某种原因稍微落后。

系统总吞吐量对执行进度倾斜的敏感性在很大程度上取决于两个因素：

- **机器数量**。机器越少，增加节点越容易发生倾斜。例如，假设 n 台机器中的每台机器都花费了 k 时间执行进度倾斜（即落后包）。那么在每一时刻都有 $1-(1-k)^n$ 概率至少有一台节点在减慢系统的速度。随着 n 的增长，这个概率趋近于 1，每增加一台机器，倾斜效应就越小。
- **争用级别**。争用率越高，每台机器的随机减速就越有可能导致其他机器也降低它们的执行速度。在低争用（争用指数 = 0.0001）情况下，我们看到每个节点的吞吐量只有在添加前几台机器时急剧下降，然后在大约 10 个节点时趋于平稳，因为执行进度倾斜的逐渐减小的增长对总吞吐量的影响相对较小。在较高的争用（争用指数 = 0.01）情况下，我们看到了更剧烈的初始下降，在曲线开始变平之前需要添加更多的机器，因为即使是执行进度倾斜水平上的微小增量增加也会对吞吐量产生显著影响。

8.2.9.2 处理高争用

大多数现实世界的工作负载在大多数时候都有低争用，但是出现少量的极热数据项并不罕见。因此，我们在这些工作负载下对 Calvin 进行了实验，我们认为这种工作负载是很少实际系统尝试支持分布式事务的主要原因：结合许多高争用的多分区事务。因此，在这个实验中，我们并不关注实际工作负载的全部，而是只考虑由高争用多分区事务组成的工作负载的子集。其他事务仍然可能与这些高度冲突的事务（除了那些非常热的事务之外）发生冲突，因此工作负载的这个子集的吞吐量可能与整个系统吞吐量紧密耦合。

Figure 6: Slowdown for 100% multipartition workloads, varying contention index.

图 6 显示了 4 节点和 8 节点 Calvin 系统在运行 100% 多分区事务时（与运行完全可分割、低争用版本的相同工

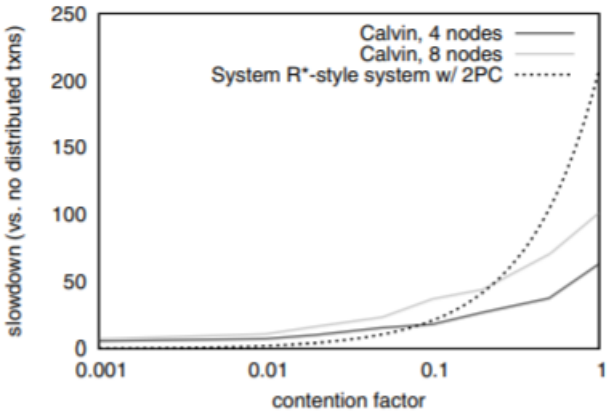


图 127: slowdown

作负载相比) 变慢的因素，这取决于争用。回想一下，争用索引是每个事务锁定的热记录总数的一部分，因此争用指数为 0.01 意味着最多可以并发执行 100 个事务，而争用指数为 1 则强制事务完全串行地运行。

因为现代分布式系统的实现没有实现系统 R 风格的两阶段提交分布式事务，与任何早期系统不会成为同类对比，我们包括比较 contention-based 放缓的一个简单的模型的系统。我们假设在非多分区、低争用的情况下，该系统将获得与 Calvin 类似的吞吐量 (每台机器大约每秒 27000 个微基准事务)。为了计算多分区事务引起的速度下降，我们考虑了两阶段提交引起的争用占用空间的扩展。由于给定一个争用索引 C，最多可以并发执行 1/C 事务，因此在提交时运行 2PC 的系统每秒执行的事务总数永远不会超过 1/C、D2PC，其中 D2PC 是两阶段提交协议的持续时间。

典型的 EC2 数据中心节点之间的往返 ping 延迟约为 1 毫秒，但包括消息多路复用、序列化/反序列化和线程调度的延迟，在我们的系统中，事务执行线程之间的单向延迟几乎从不小于 2 毫秒，而且通常更长。因此，在我们的系统模型中，开销类似于 Calvin，我们期望在每个分布式事务上持有大约 8ms 的锁。注意，这个模型有点理想化，因为假定事务的争用占用空间只包括两阶段提交的延迟。其他导致 Calvin 实际减速的因素在这个模型中被完全忽略，包括:

- 多分区事务的 CPU 成本
- 启动 2PC 之前到达本地提交/中止决策的延迟 (在实际系统中可能需要额外的远程读操作)
- 执行进度倾斜 (假设所有节点都开始执行每个事务，然后以完美的同步方式执行 2PC)

执行进度倾斜 (假设所有节点都开始执行每个事务，然后以完美的步调执行 2PC)，在高争用分布式事务的上下文中，会比该模型预测的速度慢得多。

图 6 所示的结果中有两个非常显著的特性。首先，在低争用的情况下，Calvin 得到了大约 5 到 7 倍的速度降低——从 27000 减少到大约 5000(4 个节点) 或 4000(8 个节点)/秒的事务——正如我们在前面的实验中看到的，从 1 台机器减少到 4 或 8 台。对于本实验中检查的所有争用级别，4 个节点和 8 个节点情况下吞吐量的差异是不同节点之间工作负载执行进度倾斜的结果; 正如可以预测的那样，这种倾斜对吞吐量的不利影响在较高的争用级别上要严重得多。

第二，正如预期的那样，在非常高的争论中，尽管我们忽略了一些预期的成本，运行两阶段提交的系统模型比 Calvin 产生的速度明显要慢得多。这就证明了 (a) 分布式提交协议是大多数现代分布式系统决定不支持 ACID 事务背后的主要因素，(b) Calvin 缓解了这个问题。

8.2.10 相关工作

Calvin 体系结构的一个关键贡献是，它以主动复制为特色，其中相同的事务性输入被发送到多个副本，每个副本以确定性的方式处理事务性输入，以避免发散。已经有几个相关的尝试以这种方式主动复制数据库系统。Pacitti

et al.[23],Whitney et al.[29],Stonebraker et al.[27],Jones et al.[16] 所有提出执行事务处理在一个没有并发控制的分布式数据库通过连续执行事务，因此相当于一个已知的串行顺序在每个节点上单个线程 (在某些情况下，一个节点可以是单一 CPU 核心多核服务器 [27])。通过串行执行事务，消除了并发事务的线程调度导致的不确定性，更容易实现主动复制。然而，序列化事务可能会限制事务吞吐量，因为如果一个事务停止 (例如网络读取)，其他事务将无法接管。Calvin 支持并发事务，同时仍然确保与给定串行顺序的逻辑等价。此外，尽管这些系统在执行前选择了一个串行顺序，但对这个顺序的遵守并不像 Calvin 中那样严格 (例如，事务可能会由于硬件故障而中止)，因此分布式事务仍然需要两阶段提交。

上述每一项工作都实现了一个类似于 Calvin 序列层的系统组件，该组件选择序列顺序。Calvin 的排序器设计最类似于 H-Store 设计 [27]，其中客户端可以向集群中的任何节点提交事务。输入之间的同步副本不同，然而，在 Calvin 可以使用异步 (日志传送) 复制或 Paxos-based 强烈一致的同步复制，而 H-Store 复制输入通过拖延交易的预期网络延迟发送一个事务复制，然后使用一个确定的方案为事务顺序假设所有事务到达这个时间窗口内的所有副本。

Bernstein 等人的 Hyder[8] 在概念上与 Calvin 相似，尽管实现高可伸缩性的建筑设计和方法截然不同。在 Hyder 中，事务根据从最近的快照中获得的数据库视图执行后提交它们的 “意图” ——缓冲写。意图是由全据有序，处理一个确定的 “融合” 函数，决定交易承诺什么，必须中止交易 (例如由于数据更新后失效数据库的事务的观点执行事务，但在融合函数验证事务)。Hyder 的全局有序的事情到尝试的日志确定地由事务的后续影响组成，而 Calvin 中的类似日志包含未执行的事务请求。然而，Hyder 的乐观方案在概念上与 3.2.1 节中讨论的乐观锁位置预测方案 (OLLP) 非常相似。OLLP 的 “侦察” 查询确定事务输入，在 “实际” 事务执行时，以与 Hyder 的 meld 函数确定验证事务结果相同的乐观方式确定验证事务输入。

Lomet 等人提出在云设置中 “拆分” 事务处理系统组件，其方式类似于 Calvin 将管道的不同阶段分离到不同的子系统 [21]。尽管 Lomet 等人的并发控制和复制机制与 Calvin 的不同，两个系统都将 “事务组件” (调度层) 与 “数据组件” (存储层) 分离开来，从而允许任意存储后端根据应用程序的需要为事务处理系统提供服务。Calvin 还将解捆绑更进一步，将处理数据复制的测序层分离出来。

谷歌的 Megastore[7] 和 IBM 的 Spinnaker[25] 最近率先使用了 Paxos 算法 [18,19]，用于现代的大容量事务数据库中的强一致性数据复制 (尽管 Paxos 及其变异体被广泛用于在无数其他应用程序中达成同步协议)。像 Calvin 一样，Spinnaker 使用 ZooKeeper[15] 来实现 Paxos。由于它们不是确定性系统，所以 Megastore 和 Spinnaker 都必须使用 Paxos 来复制事务性效果，而 Calvin 只需要使用 Paxos 来复制事务性输入。

8.2.11 未来工作

在当前实现中，Calvin 通过从最近的完整快照中恢复崩溃的机器，然后重放所有最近的事务来处理硬件故障。但是，由于同一副本中的其他节点可能依赖于从受影响的机器进行的远程读取，因此，在恢复完成之前，副本的其余部分的吞吐量很容易变慢或停止。

在未来，我们打算开发一个更加无缝的故障转移系统。例如，使用简单的技术，故障可以完全不可见。所有副本的集合可以分为复制子组——通常在同一局域网络上的彼此相邻的副本对或三副本。在一个副本中的数据库节点 A 与多部分事务执行值相关的传出消息不仅发送到同一副本中的预期节点 B，还发送到复制子组中节点 B 的每个副本，以防子组节点 A 副本发生故障。这种冗余技术伴随着各种权衡，如果分区间网络通信有可能成为瓶颈 (特别是因为确定性系统中的活动复制已经提供了高可用性)，则无法实现这种权衡，但它说明了在发生故障时实现高度” 无问题” 系统。

这两种方法之间的一个很好的折衷办法可能是集成一个组件，该组件可监视每个节点的状态，可以检测故障，并通过指导故障计算机的其他副本适当地转发远程读取消息，为具有故障节点的副本精心安排更快的故障转移。这样的组件还可以很好地监视只读查询、动态数据迁移和重新分区以及负载监控的负载平衡。

8.2.12 结论

本文介绍了 Calvin，这是一个事务处理和复制层，旨在将通用的、非事务性的、未复制的数据存储转换为完全 ACID 的、一致复制的分布式数据库系统。Calvin 支持数据库的水平可伸缩性和不受约束的 acid 兼容分布式事务，同时支持异步和基于 paxos 的同步复制，既可以在单个数据中心内进行，也可以跨地理上分离的数据中心进行。通过使用确定性框架，Calvin 能够消除分布式提交协议，这是现代分布式系统最大的可伸缩性障碍。因此，Calvin 在简化的 TPC-C 基准上实现了接近世界纪录的事务吞吐量。

8.2.13 致谢

这项工作是由 NSF 在 IIS-0845643 和 IIS-0844480 赠款下赞助的。Kun Ren 获得中国国家自然科学基金 61033007、国家 973 基金 2012CB316203 资助。本材料中所表达的任何意见、发现、结论或建议均为作者个人观点，并不一定反映国家科学基金会 NSF 或中国国家自然科学基金会的观点。

8.2.14 引用

[1] Amazon simpledb. <http://aws.amazon.com/simpledb/>.

[2] Project voldemort. <http://project-voldemort.com/>.

[3] Riak. <http://wiki.basho.com/riak.html>.

[4] Transaction processing performance council.<http://www.tpc.org/tpcc/>.

[5] D. Abadi. Replication and the latency-consistency tradeoff. <http://dbmsmusings.blogspot.com
↪ /2011/12/replication-andlatency-consistency.html>.

[6] J. C. Anderson, J. Lehnardt, and N. Slater. CouchDB: The Definitive Guide. 2010.

[7] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In CIDR, 2011.

[8] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In CIDR, 2011.

[9] D. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In SIGMOD, 2010.

[10] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In SIGMOD, 2011.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In OSDI, 2006.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!' s hosted data serving platform. VLDB, 2008.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon' s highly available key-value store. SIGOPS, 2007.

[14] S. Gilbert and N. Lynch. Brewer' s conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 2002.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In In USENIX Annual Technical Conference.

[16] E. P. C. Jones, D. J. Abadi, and S. R. Madden. Concurrency control for partitioned databases. In SIGMOD, 2010.

HUBBLE

[17] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In PODC, 2009.

[18] L. Lamport. The part-time parliament. ACM Trans. Comput. Syst., 1998.

[19] L. Lamport. Paxos made simple. ACM SIGACT News, 2001.

[20] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. VLDB, 2009.

[21] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In CIDR, 2009.

[22] C. Mohan, B. G. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. ACM Trans. Database Syst., 1986.

[23] E. Pacitti, M. T. Ozsu, and C. Coulon. Preventive multi-master replication in a cluster of autonomous databases. In Euro-Par, 2003.

[24] E. Plugge, T. Hawkins, and P. Membrey. The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing. 2010.

[25] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. VLDB, 2011.

[26] M. Seltzer. Oracle nosql database. In Oracle White Paper, 2011.

[27] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it' s time for a complete rewrite). In VLDB, 2007.

[28] A. Thomson and D. J. Abadi. The case for determinism in database systems. VLDB, 2010.

[29] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In HPTS, 1997.

8.3 缓存和并发控制

原文：Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System

8.3.1 摘要

由于两个主要的限制因素，分布式事务的性能很差。首先，分布式事务具有较高的延迟，因为它们对远程数据的每一次访问都会导致较长的网络延迟。第二，这种高延迟增加了分布式事务之间发生争用的可能性，导致高终止率和低性能。

我们提出了 Sundial，一种内存中的分布式乐观并发控制协议，它解决了这两个限制。首先，为了降低事务终止率，Sundial 根据事务的数据访问模式在运行时动态地确定事务之间的逻辑顺序。Sundial 通过将逻辑契约应用于每个数据元素来实现这一点，这允许数据库动态计算事务的逻辑提交时间戳。第二，为了减少远程数据访问的开销，Sundial 允许数据库在服务器的本地主存储器中缓存远程数据并保持缓存一致性。使用逻辑租赁，Sundial 将并发控制和缓存一致性集成到一个简单的统一协议中。我们对最先进的分布式并发控制协议进行 Sundial 评估。在高争用情况下，Sundial 的性能优于下一个最佳协议高达 57%。Sundial 的缓存方案在高访问偏差的工作负载中提高了高达 4.6× 的性能。

8.3.2 介绍

当联机事务处理 (OLTP) 应用程序的计算和存储需求超过单个服务器的资源时，企业通常将目光转向分布式数据库管理系统 (DBMS)。这样的系统将数据库分割成不相交的子集，称为分区，这些子集存储在多个无共享服务器上。如果事务只在单个服务器上访问数据，那么这些系统就实现了很高的性能 [2,3]。但是，当事务访问分

HUBBLE

布在多个服务器 [38] 的数据时，性能会降低。这有两个原因。首先最主要的原因，过长的网络延迟导致分布式事务的执行时间长，使得它们比单分区事务慢。第二，长的执行时间增加了事务之间发生争用的可能性，导致了更多的终止和性能下降。

最近改进分布式并发控制的工作集中在协议级和硬件级的改进上。改进的协议可以减少事务 [22,35,36,46] 之间的同步开销，但由于高协调开销（例如锁阻塞），仍然可以限制性能）。硬件级的改进包括使用特殊的硬件，如优化的网络，使低延迟远程内存访问 [20,48,55]，这可以增加系统的总体成本。理想情况下，并发控制协议应该在低成本的商品硬件上实现良好的性能。

为了避免长时间的网络延迟，先前的工作大多建议在多个数据库服务器 [16] 中复制频繁访问的共享数据，从而使一些分布式事务成为本地事务。这种复制需要分析数据库工作负载或手动识别热数据，这为用户增加了复杂性。数据库最好通过缓存机制自动复制数据，而无需用户干预。

为了解决这些问题，我们引入了 Sundial，一种内存中的分布式并发控制协议，它优于现有的方法。为了解决长网络延迟问题，Sundial 原生支持服务器本地主存中的数据缓存，而不需要附加协议或牺牲序列化性，并且只在有益的情况下使用缓存的数据。为了解决协调分布式事务的高开销，Sundial 采用了一种混合的悲观/乐观并发控制协议，该协议通过动态重新排序经历读写冲突的事务来减少中止。

Sundial 后面的启用技术是一个称为逻辑租约的概念，它允许 DBMS 在执行序列化的同时动态地确定事务之间的逻辑顺序。逻辑租约包括两个逻辑时间戳，指示元组有效的逻辑时间范围。DBMS 动态计算事务的提交时间戳，它必须与事务访问的所有元组的租约重叠。先前的工作表明，逻辑租约在提高硬件缓存一致性 [52] 和多核并发控制协议 [53] 的性能和并发性方面是有效的。据我们所知，本文是第一个将逻辑租约应用于分布式系统，也是第一次无缝地将并发控制和缓存结合在一个无共享数据库中。

为了评估 Sundial，我们在分布式 DBMS 测试床中实现了它，并将其与三种最先进的协议进行了比较：MaaT[35]、Google F1[41]，带 Wait-Die 的两阶段提交。我们使用两个具有不同争用设置和集群配置的 OLTP 工作负载。Sundial 相比于其他最好的协议实现了高达 57% 的高吞吐量和 41% 的低延迟。对于具有倾斜读的密集型工作负载，Sundial 的缓存机制提高了最高 4.6x 的吞吐量。

8.3.3 背景和相关工作

并发控制为数据库事务提供了两个关键属性：原子性和隔离性。原子性保证了事务的所有更改或任何更改都应用于数据库。隔离指定事务何时可以看到其他事务的写入。我们在本文中考虑了可串行化：并发事务产生的结果与顺序执行的结果相同。

虽然序列化是一个健壮且易于理解的隔离级别，但是提供序列化性的并发控制协议在 DBMS 中会产生大量同步开销。这已经在多核 [51] 和分布式 [25] 设置中并发控制的可伸缩性研究中得到了证明。在分布式 DBMS 中，由于数据库服务器之间的高网络延迟，问题更加严重。

常用的两类并发控制协议 [9]：两阶段锁 (2PL) 和时间戳排序 (T/O)。2PL[11，21] 协议是悲观的：事务只有在事务获得具有适当权限的锁（例如读或写）之后才访问元组）。相反，T/O 协议使用逻辑时间戳来确定事务的逻辑提交顺序。T/O 协议的变体包括多版本并发控制 (MVCC)，它维护每个元组的多个版本以减少冲突；乐观并发控制 (OCC)，其中只有在事务执行后才会检查冲突。与 2PL[51] 相比，OCC 可能会招致更多的中止，但具有非阻塞执行的优点。Sundial 将 2PL 和 OCC 的好处分别用于写-写和读-写冲突。Sundial 通过逻辑租赁（参见第 3 节）进一步减少了读写冲突导致的中止。

8.3.3.1 分布式并发控制

最近的研究提出了许多分布式并发控制协议 [15,20,35,41,48]，这些协议都是基于 2PL 的变体 (例如 Spanner[15] 和 F1[41]) 或 T/O(例如 MaaT[35] 和 Lomet 等人 [34])。在这些协议中，MaaT[35] 是最接近 Sundial 的，它们都使用逻辑时间戳来动态调整事务之间的提交顺序，但使用了不同的技术。MaaT 要求所有冲突事务显式地相互协调以调整它们的时间戳间隔，这将会损害性能并导致更多的中止。相反，冲突的事务在 Sundial 中不进行协调。此外，Sundial 使用缓存来减少网络延迟，而 MaaT 不支持缓存。

Lomet 等人 [34] 提出了一种多版本并发控制协议，该协议使用时间戳范围懒惰地确定事务的提交时间戳。该协议在多核和分布式设置中都有效。在冲突时，协议要求所有涉及的事务缩小其时间戳范围-这是 Sundial 通过逻辑租约避免的昂贵操作。此外，他们的协议需要一个多版本数据库，而 Sundial 工作在一个单一版本数据库。第 6.8 节比较了 Lomet 等人的协议和 Sundial 的性能。

8.3.3.2 多核并发控制

快速事务处理在单服务器多核心系统中进行了深入的研究 [28,32,37,47,49,50,53,54]。其中一些技术也可以应用于分布式环境：我们将其中一些合并到我们的分布式并发控制协议中。为多核处理器开发的其他技术由于不同的系统特性，如高网络延迟，不能很容易地在分布式设置中使用。

在本论文中，我们关注的是分布式并发控制协议的性能，因此不与单服务器多核系统设计的协议进行比较。

8.3.3.3 数据复制和缓存一致性

前边提到数据复制 [16,38] 可以减小分布式事务的开销，复制对于热（经常访问的）只读数据特别有效。当跨服务器复制时，热数据可以在本地服务于读请求，大大减少了远程请求的数量。

数据复制有几个缺点。首先，数据库的用户需要配置工作负载，或者手动指定要复制的数据，这是一个快速发展的数据库中的艰巨任务。其次，当 DBMS 更新被复制的数据时，会通知所有持有副本的服务器以保持一致性，这很复杂，增加了性能开销。第三，全表复制会增加内存占用，如果复制的表很大，这是有问题的。

缓存是利用热点数据的更灵活的解决方案。具体来说，DBMS 决定在没有用户干预的情况下在服务器的本地内存中缓存远程数据。从本地缓存读取数据的查询不会与远程服务器联系，从而减少网络延迟和流量。缓存协议中的关键挑战是保持缓存一致性（即避免机器读取陈旧的缓存数据）。缓存一致性是并行计算中的一个经典问题，许多先前的工作存在于不同的研究领域，包括多核和多套接字处理器 [6,42,56]、分布式共享内存系统 [27,31]、分布式文件系统 [24] 和数据库 [5,23,39]。在传统的 DBMS 中，缓存和并发控制是两个独立的层；它们都是众所周知的难以设计和验证 [43]。在第 4 节中，我们演示了如何借助租赁契约将并发控制和缓存一致性合并到单个轻量级协议中。

8.3.3.4 整合并发控制和缓存一致性

并发控制和缓存一致性的集成也是在数据共享系统 [40] 的背景下进行的研究。例如包括 IBM DB2[26]、Oracle RAC[13]、Oracle Rdb[33] 以及最近的 Microsoft Hyder[10]，在这些系统中，集群中的服务器共享存储 (即共享磁盘架构)，每个服务器都可以访问数据库中的所有数据，并在本地缓冲区中缓存最近访问的数据。缓存一致性协议确保缓存中的数据新鲜度。

Sundial 在两个方面不同于数据共享系统。首先，Sundial 是建立在无共享架构之上的，因此比使用共享磁盘的数据共享系统更有规模。第二，现有的数据共享系统通常使用 2PL 进行并发控制；一致性协议也基于锁。相比之下，Sundial 对并发控制和缓存一致性都使用逻辑租赁，这降低了协议的复杂性，增加了并发（第 4 节）。

另一个相关的系统是 G-Store[17]，它支持通过键值存储进行事务多键访问。在事务启动之前，服务器启动分组协议，以收集事务将访问的密钥的独占所有权；事务完成之后，所有权被收回。分组协议有两个缺点：(1) 它要求 DBMS 在执行事务之前知道事务将访问哪些键，(2) 不允许多个事务同时读取同一 key。这些问题在 Sundial 中不存在，它支持动态工作集和并发读取。

8.3.4 SUNDAIL 并发控制

本节详细介绍了 Sundial 分布式并发控制协议。讨论假设通过局域网 (LAN) 连接的同质服务器集群。数据跨服务器进行分区，每个元组映射到一个主服务器，该主服务器可以代表外部用户启动事务，以及代表对等服务器处理远程查询。发起事务的服务器称为事务的协调器；参与事务的其他服务器称为参与者。

8.3.4.1 逻辑租约

逻辑租约基于逻辑时间，并指定并发操作之间的部分逻辑顺序。每个数据元素都附带一个逻辑租约，使用两个64 位时间戳wts和rts表示。wts是最后一次写入元组的逻辑时间；rts是租约的结束，这意味着元组可以在逻辑时间ts读取，从而使wts ts rts。事务只在当前租约到期后才写入元组，即在不少于rts+1的时间戳下写入元组。由于租约在 Sundial 中是合乎逻辑的，所以写入事务不会在物理时间wait租约到期-它只是在逻辑时间提前跳转。

为了实现序列化，DBMS 计算事务T的单个提交时间戳，这是一个逻辑时间，属于事务访问的所有元组的租约范围。也就是说，对于T访问的所有元组，tuple.wts T.commit_ts tuple.rts。从逻辑时间的角度来看，T 的操作是在提交时间戳处原子执行的。DBMS 只使用事务访问的元组的租约来计算提交时间戳，因此不需要跨事务协调。

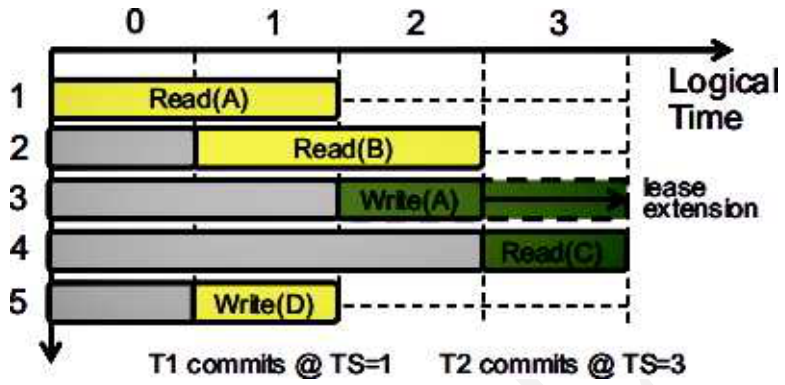


图 128: 图 1

图 1：逻辑租约示例两个事务调度，T1 和 T2，访问元组 A、B、C 和 D。T1 和 T2 的操作分别用黄和绿来表示。由于逻辑时间戳是离散的，所以每个时间戳表示为图中的间隔。

图 1，说明 Sundial 并发控制的高级思想。DBMS 执行两个事务，T1 和 T2。T1 读取具有逻辑租约 [0,1] 和 [1,2] 的元组 A 和 B；T1 还写入 D 并为新数据创建新的租约 [1,1]。T1 在时间戳 1 处提交，因为它与 T1 访问的所有租约重叠。

T2 写入 A，并创建新的 [2,2] 租约，它同时在 [3,3] 读取 C。然而，在这个情况下，这两项租约并不重叠。在这种情况下，DBMS 将 A 上租约的结束从 2 扩展到 3，这样 T2 执行的两个操作都在时间戳 3 处有效。如果租约扩展成功，即系统中没有其他事务在时间戳 3 写入 A，则 DBMS 在时间戳 3 提交 T2。

请注意，在本例中，当 T1 提交时，T2 已经修改了 A，这是由 T1 读取的。然而，这并不像传统的 OCC 协议那样导致 T1 中止，因为 Sundial 以逻辑顺序而不是物理时间顺序序列化事务。物理和逻辑提交顺序可能不同，如示例所示-虽然 T1 在 T2 之后按物理时间顺序提交，但 T1 提交的逻辑时间戳较小。这个 Sundial 的时间租约特性允许 DBMS 动态地确定事务的提交时间戳，这避免了由于读写冲突而导致的一些不必要的中止，并导致较低的中止率。

8.3.5 冲突处理

Sundial 使用不同的方法来解决事务之间的不同类型的冲突。在数据库 [18,44] 和软件事务内存 [19] 中也提出了这种混合方案。

Sundial 使用悲观 2PL 处理写-写冲突。我们之所以做出这种选择，是因为许多写都是读-修改-写，而对同一个元组的两个这样的更新总是冲突的 (除非 DBMS 利用语义信息，如交换性)；使用 OCC 处理这种冲突会导致除一个冲突事务之外的所有事务中止。相反，Sundial 使用 OCC 而不是 2PL 处理读写冲突，以实现更高的并发。这防止事务在执行过程中等待锁。更重要的是，这允许 DBMS 使用 3.1 节中讨论的逻辑租约技术动态调整事务的提交顺序以减少中止。

图 2：读写冲突例子在 2PL，OCC 和 Sundial 中两个存在读写冲突的事务处理

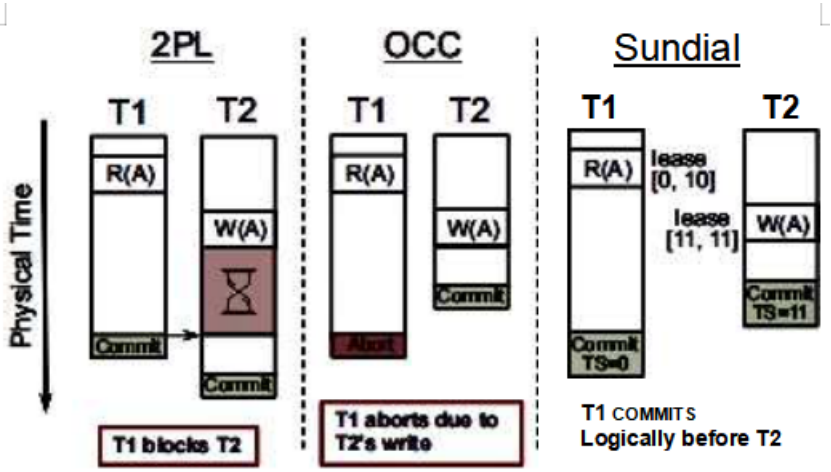


图 129: 图 2

图 2 是一个示例，说明 2PL、OCC 和 Sundial 在处理读写冲突方面的区别。在 2PL 中，DBMS 在事务 T1 读取之前锁定的一个元组，此锁阻止 T2 写入到相同的元组。这增加了 T2 的执行时间。在传统的 OCC 协议中，事务在正常执行期间不会持有锁。因此，T2 在 T1 之前提交，并在数据库中更新元组 A，当 T1 尝试提交时，自上次 T1 读取它以来，A 已经更改。然后，DBMS 由于这种数据冲突终止了 T1。

在本例中，终止 T1 是不必要的，因为在保持序列化性的两个事务之间存在逻辑提交顺序，即 T1 在 T2 之前提交。Sundial 能够使用逻辑租约识别此顺序。具体来说，T1 读取带有租约 [0,10] 的 A 版本，并在该租约中的时间戳上提交 (例如。时间戳 0)。T2 以新的 [11,11] 租约写入 A，因此在时间戳 11 提交。虽然 T1 按照物理时间顺序在 T2 之后作出提交决定，但这两个事务都能够按照逻辑时间顺序在 T2 之前提交 T1。

8.3.6 协议阶段

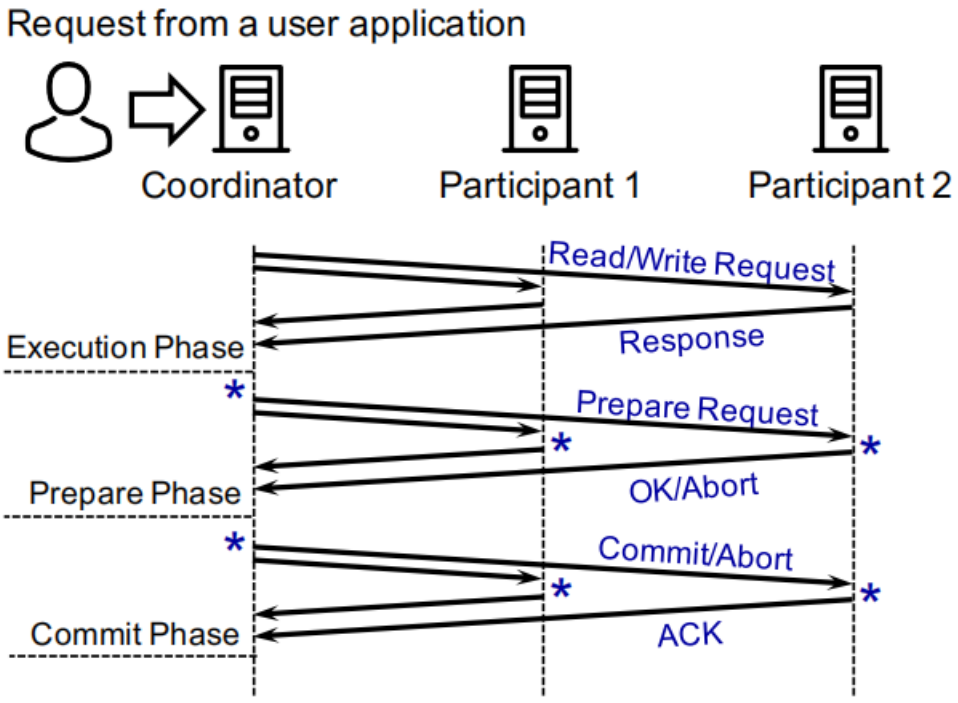


图 130: 图 3

图 3：一个分布式事务的生命周期一个分布式事务经过执行阶段和两阶段提交阶段-其中包含准备阶段和提交阶段

Sundial 中分布式事务的生命周期如图 3，包含三个阶段：(1) 执行 (2) 准备 (3) 提交。协调器在收到用户应用程序请求后启动新事务。在第一阶段执行阶段中，协调器执行事务的逻辑并向事务参与者发送请求。事务完成执行阶段时，协调器开始两阶段提交。在两阶段提交的第一阶段准备阶段，协调器发送一个准备提交请求到参与事务的每个参与者（即在执行阶段访问的服务器）。每个参与者响应一个 OK 或者 ABORT。如果所有参与者包括协调者都同意事务可以提交，则事务进入到最终阶段提交阶段。协调者向每个参与者发送消息以完成事务。否则，DBMS 会终止事务并回滚其修改。为了确保事务的修改在服务器/网络故障中持续存在，DBMS 在 2PC 协议期间在每个服务器的不同点设置日志记录（图 3 中的*表示）。

在 Sundial 中，只有在事务提交阶段更新元组时，租约的`wts`才会发生变化；租约的`rts`只在准备阶段或提交阶段发生变化。元组的`wts`和`rts`只能增加，但永远不会减少。在这三个阶段中，DBMS 可以协调参与者计算事务提交时间戳或扩展逻辑租约。DBMS 在其正常运行时消息中捎带与时间戳相关的信息。我们现在更详细地描述每个阶段的逻辑。

8.3.6.1 执行阶段

在执行阶段，事务在数据库中读取和写入元组并执行计算。每个元组包含一个逻辑租约 (即`wts`和`rts`)。Sundial 处理写-写冲突使用 2PL 等待-删除算法 [9]。因此，DBMS 还维护当前锁持有者和等待锁的事务的等待列表。每个元组都有以下格式，其中 DB 是数据库，DB[key] 表示以 key 为主键的元组：

```
DB[key] = {wts, rts, owner, waitlist, data}
```

Sundial 使用 OCC 处理读写冲突，因此 DBMS 维护每个事务的读集 (RS) 和写集 (WS)。两组数据结构如下所示。数据字段包含事务读取 (RS) 或写入 (WS) 的数据库元组的本地副本。

```
RS[key] = {wts, rts, data}, WS[key] = {data}
```

算法 1：事务 T 的执行阶段

当 T 开始时 T.commit_ts 初始化为 0。缓存相关代码以 * 开头（详见第 4 节）

```
Function read(T, key)
    if key   T.WS:
        return WS[key].data
    elif key   T.RS:
        return RS[key].data
    else:
        *if key   Cache and Cache.decide_read():
        *    T.RS[key].{wts, rts, data} = Cache[key].{wts, rts, data}
        else:
            n = get_home_node(key)
            # read_data() atomically reads wts, rts, and data and
            # return them back
            T.RS[key].{wts, rts, data} = RPCn::read_data(key)
        *   Cache[key].{wts, rts, data} = T.RS[key].{wts, rts, data}
        T.commit_ts = Max(T.commit_ts, T.RS[key].wts)
        return RS[key].data

Function write(T, key, data)
    if key   T.WS:
```

HUBBLE

```
        n = get_home_node(key)
        # lock() tries to lock the tuple DB[key]; if locking is successful,
        # it returns wts and rts of the locked tuple
        {success, wts, rts} = RPCn::lock(key)
        if not success or (key   T.RS and wts   T.RS[key].wts):
            Abort(T)
        else:
            T.commit_ts = Max(T.commit_ts, rts + 1)
    T.WS[key].data = data
```

对于读取请求，如果请求的元组已经在事务 T 的读或写集中，DBMS 只需返回数据（第 2-5 行）。否则，DBMS 将远程过程调用 (RPCn) 发送到元组的主服务器 n，后者原子地读取元组的`wts`、`rts`和数据，并将它们返回到协调器的读取集（第 10-12 行）。然后，协调器将 T.commit_ts 更新为至少是元组的`wts`（第 14 行），这反映了 T 的逻辑提交时间必须不早于上一次写入元组的逻辑时间。最后返回数据（第 15 行）。

对于写操作，如果元组已经在 T 的写集中，DBMS 只需更新写集中的本地数据（第 17、25 行）。否则，DBMS 通过向其主服务器发出 RPC 来锁定元组（第 18-20 行）。RPC 返回锁获取是否成功，如果成功，则返回锁定元组的`wts`和`rts`。如果锁失败，或者如果锁定的密钥存在于读取集中，但返回的`wts`与读取集中的`wts`不匹配，则事务中止（第 22 行）。否则，DBMS 将事务的`commit_ts`提前到`rts+1`（第 24 行），并更新写集中的数据 (第 25 行)。请注意，在 T 的执行阶段，其他事务不能读取 T 写集中的元组-它们只有在 T 提交后才可见（3.3.3 节）。

Sundial 的一个重要特征是，读取元组的事务不会阻止写入相同元组的另一个事务（反之亦然）。如果元组被锁定，读取事务仍然读取元组的数据，并与其相关的逻辑租约。读取事务的提交时间戳将小于持有锁的事务的提交时间戳。但是，只要两个事务都找到满足其租约的提交时间戳，它们就能够提交。

8.3.6.2 准备阶段

算法 2：事务 T 的准备阶段协调器执行 validate_read_set()，参与者执行 renew_lease()

缓存相关代码以 * 开头

```
Function validate_read_set(T)
    for key   T.RS.keys() \ T.WS.keys():
        if commit_ts > T.RS[key].rts:
            n = get_home_node(key)
            # Extend rts at the home server
            resp = RPCn::renew_lease(key, wts, commit_ts)
            if resp == ABORT:
            *   Cache.remove(key)
                return ABORT
    return COMMIT

# renew_lease() must be executed atomically
Function renew_lease(key, wts, commit_ts)
    if wts = DB[key].wts or (commit_ts > DB[key].rts and
      DB[key].is_locked()):
        return ABORT
```

```
    else:
        DB[key].rts = Max(DB[key].rts, commit_ts)
        return OK
```

准备阶段的目标是由 DBMS 确定事务的所有读写是否在计算 commit_ts 有效。算法 2 显示了 DBMS 在准备阶段执行的逻辑；validate_read_set() 在协调器上执行，协调器在参与者中调用 renew_lease()。

由于写集中的所有元组都已锁定，DBMS 只验证已读但未写的元组。对于事务 T 的读集中的每个键，而不是写集（第 2 行），如果 T.commit_ts 在元组的租约范围内 (即`tuple.wts` `T.commit_ts` `tuple.rts1`)，则读取已经有效，不需要做任何事情（第 3 行）。否则，RPC 将被发送到元组的主服务器以更新其租约（第 4-6 行）。如果续租失败，交易中止（第 9 行）。

参与者执行 renew_lease() 以执行租约续签。如果元组的当前 wts 与事务在执行阶段观察到的 wts 不同，或者如果需要扩展但元组被不同的事务锁定，则 DBMS 不能扩展租约，ABORT 返回给协调器（第 13-14 行）。否则，DBMS 通过将元组的 RTS 更新到至少 commit_ts （行 16) 来扩展租约)。在 Sundial 中，租约扩展是更改元组 RTS 的唯一方法。每个元组的 wts 和 rts 都只能增加，但永远不会减少。

8.3.6.3 提交阶段

在提交阶段，数据库要么通过将写入集的所有写入复制到数据库并释放锁来提交事务，要么通过简单地释放锁来中止事务。算法 3 显示协调器执行的函数。如果锁获取在执行阶段失败，或者 validate_read_set() 返回 ABORT，则 abort()，否则执行 commit()。

算法 3：事务 T 的提交阶段

数据库提交或终止事务

缓存相关代码以 * 开头

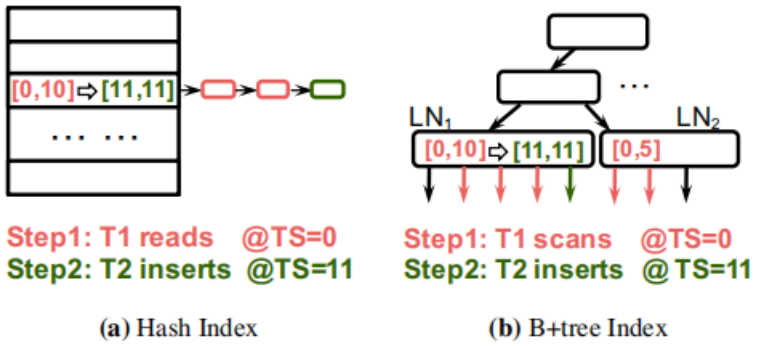
```
Function commit(T)
    for key   T.WS.keys():
        n = get_home_node(key)
        RPCn::update_and_unlock(key, T.WS[key].data, T.commit_ts)
    *   Cache[key].wts = Cache[key].rts = T.commit_ts
    *   Cache[key].data = T.WS[key].data

Function abort(T)
    for key   T.WS.keys():
        n = get_home_node(key)
        # unlock() releases the lock on DB[key]
        RPCn::unlock(key)

Function update_and_unlock(key, data, commit_ts)
    DB[key].data = data
    DB[key].wts = DB[key].rts = commit_ts
    unlock(DB[key])
```

DBMS 在提交阶段不会对事务的读集执行任何附加操作。因此，如果事务没有写入参与者的任何元组，DBMS 就会跳过该参与者的提交阶段。Sundial 应用这个小优化来减少网络流量。

8.3.7 索引和幻读

除了常规的读和写之外，索引必须支持插入和删除。因此，索引需要额外的并发控制机制来保证正确性。具体来说，如果事务使用索引两次读取一组元组，但由于另一个事务对该集合的插入或删除而获得不同的结果集，则会发生幻读。序列化不允许幻读。本节讨论如何避免幻读。

通过将插入和删除视为对索引节点的写入，并将查找或扫描视为对索引节点的读取，可以扩展基本的 Sundial 协议来处理索引访问。这样，每个索引节点 (例如，B 树索引中的叶节点或散列索引中的桶) 都可以被视为规则元组，3.3 节中讨论的协议可以应用于索引。逻辑租约也可以保持在更细的粒度（例如，叶节点中的每个指针或桶中的每个块），以避免由于错误共享而不必要的中止；在 Sundial 中，我们将逻辑租约附加到每个索引节点。为了确保对同一索引节点的多个索引查找/扫描返回一致的结果，以后的访问必须验证索引节点的版本没有更改。



(a) Hash Index (b) B+tree Index

图 131: 图 4

图 4：并行索引读取/扫描和插入-逻辑租赁嵌入到 B 树索引的叶节点，以使索引高并发的访问。

图 4 显示了两个并发访问哈希索引 (图 4a) 和 B 树索引 (图 4b) 的事务示例。在散列索引示例中，T1 读取映射到具有 [0,10] 租约的给定桶的所有元组。然后 T2 将元组插入同一个桶中，并将桶上的租约更新为 [11,11]。T1 和 T2 有读写冲突，但不会相互阻塞；如果每个事务都找到满足所有访问租约的提交时间戳，则两个事务都可以提交。

在 B 树索引示例中，T1 执行扫描，该扫描涉及两个叶节点 LN1 和 LN2，并记录它们的逻辑租约 ([0,10] and [0,5])。扫描完成后，T2 向 LN1 插入一个新的键。当 T2 提交时，它更新 LN1 的内容，以及它对 [11,11] 的逻辑租约。类似于散列索引示例，T1 和 T2 都可以按逻辑时间顺序在 T2 之前提交和 T1 提交。因此，并发扫描和插入不需要引起中止。

8.3.8 容错性

Sundial 通过两阶段提交 (2PC) 协议容忍单服务器和多服务器故障，其中协调器和参与者在发送某些消息之前登录到持久存储（图 3）。在 2PC 期间，Sundial 中的逻辑租约需要特殊处理：如果未能地记录逻辑租约，可能会导致不可序列化的调度，如清单 1 中的示例所示。

```
清单1: 当逻辑租约未被记录时，序列化冲突-元组A和B分别存储在服务器1和2中

T1 R(A)                  lease [0,9]
T2 W(A), W(B),commit     @ TS =10
Server 2 crashes
Server 2 recovers
T1 R(B)                  lease [0,0]
```

该示例包含两个事务 (T1 和 T2)，它们分别访问存储在服务器 1 和 2 中的两个元组 A 和 B。当服务器 2 崩溃后恢复时，映射到服务器 2 的元组 (即元组 B) 的逻辑租约将重置为 [0,0]。因此，DBMS 在时间戳 0 时提交 T1，

因为 T1 访问的元组的租约 (即 [0,9] 的 A 和 [0,0] 的 B) 重叠在 0。然而，这种执行违反了序列化性，因为 T1 观察到 T2 对 B 的写入，而不是对 A 的写入。发生此违规是因为在服务器 2 从崩溃中恢复后，B 上的逻辑租约丢失并重置为 [0,0]。如果服务器 2 没有崩溃，T1 对 B 的读取将返回从时间戳 10 开始的租约，导致 T1 由于不重叠的租约和失败的租约扩展而中止。

解决上述问题的一个简单解决方案是，每当逻辑租约发生变化时，就记录它们，并在恢复后恢复它们。然而，这会导致对持久存储的大量写入，因为即使是读操作也可能会扩展租约，从而导致对日志的写入。

我们观察到，DBMS 不会记录每次租赁变更，而只能记录一个大于服务器上所有租赁结束时间的上限时间戳 UT。恢复后，服务器上的所有租约都设置为 [UT，UT]。这保证将来读取到恢复的元组发生在元组的 wts 之后的时间戳上，时间戳不大于 UT。在清单 1 所示的示例中，T1 对 B 的读取返回 [UT，UT] 的租约，其中 UT 大于或等于 10。这导致 T1 由于不重叠租赁而中止。请注意，UT 可以是租赁的松散上限。这减少了 UT 的存储和日志记录开销，因为只有当最大租约超过最后记录的 UT 时才记录它。

©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

HUBBLE