Apache ShardingSphere document

Apache ShardingSphere

2025 年 06 月 10 日

Contents

1 什么是 ShardingSphere **1**
1.1 介绍 1
1.1.1 ShardingSphere-JDBC 1
1.1.2 ShardingSphere-Proxy 1
1.2 产品功能 1
1.3 产品优势 1

2 设计哲学 **3**
2.1 连接：打造数据库上层标准 4
2.2 增强：数据库计算增强引擎 4
2.3 可插拔：构建数据库功能生态 4
2.3.1 L1 内核层 5
2.3.2 L2 功能层 5
2.3.3 L3 生态层 5

3 部署形态 **6**
3.1 ShardingSphere-JDBC 独立部署 6
3.2 ShardingSphere-Proxy 独立部署 7
3.3 混合部署架构 8

4 运行模式 **10**
4.1 单机模式 10
4.2 集群模式 10

5 线路规划 **11**

6 如何参与 **12**

7 快速入门 **13**
7.1 ShardingSphere-JDBC 13
7.1.1 应用场景 13
7.1.2 使用限制 13
7.1.3 前提条件 13

7.1.4 操作步骤 13
7.2 ShardingSphere-Proxy 15
7.2.1 应用场景 15
7.2.2 使用限制 15
7.2.3 前提条件 15
7.2.4 操作步骤 16

8 功能 **17**
8.1 数据分片 17
8.1.1 背景 17
垂直分片 18
水平分片 19
8.1.2 挑战 19
8.1.3 目标 20
8.1.4 应用场景 20
海量数据高并发的 OLTP 场景 20
海量数据实时分析 OLAP 场景 20
8.1.5 相关参考 20
8.1.6 核心概念 20
表 20
数据节点 22
分片 23
行表达式 24
分布式主键 25
8.1.7 使用限制 25
稳定支持 25
实验性支持 27
不支持 28
8.1.8 附录 29
8.2 分布式事务 29
8.2.1 背景 29
8.2.2 挑战 30
8.2.3 目标 30
8.2.4 原理介绍 30
LOCAL 事务 30
XA 事务 31
BASE 事务 31
8.2.5 应用场景 32
ShardingSphere XA 事务使用场景 32
ShardingSphere BASE 事务使用场景 32
ShardingSphere LOCAL 事务使用场景 33
8.2.6 相关参考 33
8.2.7 核心概念 33
XA 协议 33
8.2.8 使用限制 33

LOCAL 事务 . . . 33
XA 事务 . . . 33
BASE 事务 . . . 34
8.2.9 附录 . . . 34
8.3 读写分离 . . . 34
8.3.1 背景 . . . 34
8.3.2 挑战 . . . 35
8.3.3 目标 . . . 36
8.3.4 应用场景 . . . 36
复杂的主从数据库架构 . . . 36
8.3.5 相关参考 . . . 36
8.3.6 核心概念 . . . 37
主库 . . . 37
从库 . . . 37
主从同步 . . . 37
负载均衡策略 . . . 37
8.3.7 使用限制 . . . 37
8.4 数据库网关 . . . 37
8.4.1 背景 . . . 37
8.4.2 挑战 . . . 37
8.4.3 目标 . . . 38
8.4.4 应用场景 . . . 38
8.4.5 核心概念 . . . 38
SQL 方言 . . . 38
8.4.6 使用限制 . . . 38
8.5 流量治理 . . . 38
8.5.1 背景 . . . 38
8.5.2 挑战 . . . 38
8.5.3 目标 . . . 39
8.5.4 应用场景 . . . 39
计算节点过载保护 . . . 39
存储节点限流 . . . 39
8.5.5 核心概念 . . . 39
熔断 . . . 39
限流 . . . 39
8.6 数据迁移 . . . 39
8.6.1 背景 . . . 39
8.6.2 挑战 . . . 40
8.6.3 目标 . . . 40
8.6.4 应用场景 . . . 40
8.6.5 相关参考 . . . 40
8.6.6 核心概念 . . . 40
节点 . . . 40
集群 . . . 40
源端 . . . 40

目标端 41
数据迁移作业 41
存量数据 41
增量数据 41
8.6.7 使用限制 41
支持项 41
不支持项 41
8.7 数据加密 41
8.7.1 背景 41
8.7.2 挑战 42
8.7.3 目标 42
8.7.4 应用场景 42
8.7.5 相关参考 42
8.7.6 核心概念 42
逻辑列 42
密文列 43
查询辅助列 43
模糊查询列 43
8.7.7 使用限制 43
8.7.8 附录 43
8.8 数据脱敏 44
8.8.1 背景 44
8.8.2 挑战 44
8.8.3 目标 44
8.8.4 应用场景 44
8.8.5 相关参考 44
8.8.6 核心概念 44
逻辑列 44
8.8.7 使用限制 45
8.9 影子库 45
8.9.1 背景 45
8.9.2 挑战 45
8.9.3 目标 45
8.9.4 应用场景 45
8.9.5 相关参考 45
8.9.6 核心概念 46
生产库 46
影子库 46
影子算法 46
8.9.7 使用限制 46
基于 Hint 的影子算法 46
基于列的影子算法 46
8.10 可观察性 47
8.10.1 背景 47
8.10.2 挑战 49

8.10.3 目标 49
8.10.4 应用场景 49
监控仪表盘 49
应用性能监控 49
应用链路追踪 49
8.10.5 相关参考 50
8.10.6 核心概念 50
Agent 50
APM 50
Tracing 50
Metrics 50
Logging 50
8.11 联邦查询 50
8.11.1 背景 50
8.11.2 挑战 51
8.11.3 目标 51
8.11.4 应用场景 51
8.11.5 相关参考 51
8.11.6 使用限制 51

9 用户手册 52
9.1 ShardingSphere-JDBC 52
9.1.1 YAML 配置 52
简介 52
使用步骤 53
语法说明 54
模式配置 54
数据源配置 56
规则配置 57
算法配置 77
JDBC 驱动 79
9.1.2 Java API 88
简介 88
使用步骤 89
模式配置 90
数据源配置 92
规则配置 93
算法配置 118
9.1.3 特殊 API 119
数据分片 119
读写分离 122
分布式事务 124
9.1.4 可选插件 141
ClickHouse 144
Firebird 148

HiveServer2 . . . 152
Presto . . . 165
9.1.5 不支持项 . . . 169
配置 . . . 169
DataSource 接口 . . . 169
Connection 接口 . . . 169
Statement 和 PreparedStatement 接口 . . . 169
ResultSet 接口 . . . 169
JDBC 4.1 . . . 170
9.1.6 可观察性 . . . 170
Agent . . . 170
使用方式 . . . 172
Metrics . . . 174
9.1.7 GraalVM Native Image . . . 175
背景信息 . . . 175
使用限制 . . . 177
开发和测试 . . . 182
9.2 ShardingSphere-Proxy . . . 188
9.2.1 启动手册 . . . 188
使用二进制发布包 . . . 188
使用 Docker . . . 190
构建 GraalVM Native Image(Alpha) . . . 192
使用 Helm . . . 200
添加依赖 . . . 207
9.2.2 YAML 配置 . . . 208
认证和授权 . . . 208
属性配置 . . . 211
规则配置 . . . 211
数据源配置 . . . 212
9.2.3 DistSQL . . . 213
定义 . . . 213
相关概念 . . . 213
对系统的影响 . . . 214
使用限制 . . . 215
原理介绍 . . . 215
相关参考 . . . 216
语法 . . . 216
使用 . . . 413
9.2.4 数据迁移 . . . 420
简介 . . . 420
运行部署 . . . 420
使用手册 . . . 423
9.2.5 可观察性 . . . 437
Agent . . . 437
使用方式 . . . 439

Metrics . . . 440
9.2.6 可选插件 . . . 440
Seata AT 模式事务 . . . 442
9.2.7 会话管理 . . . 451
相关操作 . . . 451
9.2.8 日志配置 . . . 452
背景信息 . . . 452
操作步骤 . . . 452
9.2.9 CDC . . . 453
运行部署 . . . 453
使用手册 . . . 458
注意事项 . . . 464
9.3 通用配置 . . . 465
9.3.1 属性配置 . . . 465
背景信息 . . . 465
参数解释 . . . 465
操作步骤 . . . 465
注意事项 . . . 465
配置示例 . . . 466
9.3.2 内置算法 . . . 466
简介 . . . 466
使用方式 . . . 466
元数据持久化仓库 . . . 466
分片算法 . . . 469
分布式序列算法 . . . 475
负载均衡算法 . . . 476
加密算法 . . . 477
影子算法 . . . 479
SQL 翻译 . . . 480
分片审计算法 . . . 481
脱敏算法 . . . 481
行表达式 . . . 485
9.3.3 SQL Hint . . . 490
背景信息 . . . 490
使用规范 . . . 490
参数解释 . . . 490
SQL Hint . . . 490
9.4 错误码 . . . 492
9.4.1 SQL 错误码 . . . 492
内核异常 . . . 492
功能异常 . . . 497
其他异常 . . . 500
9.4.2 服务器错误码 . . . 500

10 开发者手册 **501**

10.1 运行模式 501
10.1.1 StandalonePersistRepository 501
全限定类名 501
定义 502
已知实现 502
10.1.2 ClusterPersistRepository 502
全限定类名 502
定义 502
已知实现 502
10.2 SQL 解析 502
10.2.1 DatabaseTypedSQLParserFacade 502
全限定类名 502
定义 502
已知实现 503
10.2.2 SQLStatementVisitorFacade 504
全限定类名 504
定义 504
已知实现 505
10.3 数据分片 506
10.3.1 ShardingAlgorithm 506
全限定类名 506
定义 506
已知实现 506
10.3.2 ShardingAuditAlgorithm 506
全限定类名 506
定义 506
已知实现 506
10.3.3 DatetimeService 506
全限定类名 506
定义 507
已知实现 507
10.3.4 InlineExpressionParser 507
全限定类名 507
定义 507
已知实现 507
10.4 基础算法 508
10.4.1 LoadBalanceAlgorithm 508
全限定类名 508
定义 508
已知实现 508
10.4.2 KeyGenerateAlgorithm 508
全限定类名 508
定义 508
已知实现 509
10.4.3 MessageDigestAlgorithm 509

全限定类名 . . . 509
定义 . . . 509
已知实现 . . . 510
10.5 SQL 审计 . . . 510
10.5.1 SQLAuditor . . . 510
全限定类名 . . . 510
定义 . . . 510
已知实现 . . . 510
10.6 数据加密 . . . 510
10.6.1 EncryptAlgorithm . . . 510
全限定类名 . . . 510
定义 . . . 510
已知实现 . . . 510
10.7 数据脱敏 . . . 510
10.7.1 MaskAlgorithm . . . 510
全限定类名 . . . 511
定义 . . . 511
已知实现 . . . 511
10.8 影子库 . . . 511
10.8.1 ShadowAlgorithm . . . 511
全限定类名 . . . 511
定义 . . . 511
已知实现 . . . 511
10.9 可观察性 . . . 511
10.9.1 PluginLifecycleService . . . 511
全限定类名 . . . 511
定义 . . . 512
已知实现 . . . 512

11 测试手册 513
11.1 整合测试 . . . 513
11.2 模块测试 . . . 513
11.3 性能测试 . . . 513
11.4 集成测试 . . . 513
11.4.1 设计 . . . 513
测试用例 . . . 514
测试环境 . . . 514
测试引擎 . . . 514
11.4.2 使用指南 . . . 515
测试用例配置 . . . 515
环境配置 . . . 516
运行测试引擎 . . . 517
11.5 性能测试 . . . 519
11.5.1 Sysbench ShardingSphere Proxy 空 Rules 性能测试 . . . 519
测试目的 . . . 519

测试环境搭建 . . . 519
测试阶段 . . . 521
11.5.2 BenchmarkSQL ShardingSphere Proxy 分片性能测试 . . . 523
测试目的 . . . 523
测试方法 . . . 523
测试工具微调 . . . 523
压测环境或参数建议 . . . 523
附录 . . . 525
BenchmarkSQL 5.0 PostgreSQL 语句列表 . . . 528
11.6 模块测试 . . . 536
11.6.1 SQL 解析测试 . . . 536
数据准备 . . . 536
11.6.2 SQL 改写测试 . . . 538
目标 . . . 538
11.7 Pipeline E2E 测试 . . . 539
11.7.1 测试目的 . . . 539
11.7.2 测试环境类型 . . . 540
11.7.3 使用指南 . . . 540
环境配置 . . . 540
测试用例 . . . 540
运行测试用例 . . . 540

12 技术参考 542
12.1 数据兼容性 . . . 542
12.2 数据库网关 . . . 543
12.3 管控 . . . 543
12.3.1 注册中心数据结构 . . . 543
/rules . . . 545
/props . . . 546
/metadata/${databaseName}/data_sources/units/ds_0/versions/0 . . . 546
/metadata/${databaseName}/data_sources/nodes/ds_0/versions/0 . . . 546
/metadata/${databaseName}/rules/sharding/tables/t_order/versions/0 . . . 547
/metadata/$databaseName/schemas/{schemaName}/tables/t_order/versions/0 . . . 547
/nodes/compute_nodes . . . 548
/nodes/qualified_data_sources . . . 548
12.4 数据分片 . . . 548
12.4.1 SQL 解析 . . . 549
12.4.2 SQL 路由 . . . 549
12.4.3 SQL 改写 . . . 549
12.4.4 SQL 执行 . . . 549
12.4.5 结果归并 . . . 549
12.4.6 查询优化 . . . 549
12.4.7 解析引擎 . . . 549
抽象语法树 . . . 549
SQL 解析引擎 . . . 550

12.4.8 路由引擎 552
分片路由 552
广播路由 554
12.4.9 改写引擎 556
正确性改写 556
优化改写 561
12.4.10 执行引擎 562
连接模式 562
自动化执行引擎 563
12.4.11 归并引擎 566
遍历归并 567
排序归并 567
分组归并 568
聚合归并 571
分页归并 571
12.5 分布式事务 572
12.5.1 导览 572
12.5.2 XA 事务 572
开启全局事务 573
执行真实分片 SQL 573
提交或回滚事务 574
12.5.3 Seata 柔性事务 574
引擎初始化 575
开启全局事务 575
执行真实分片 SQL 575
提交或回滚事务 576
12.6 数据迁移 576
12.6.1 原理说明 576
12.6.2 执行阶段说明 577
准备阶段 577
存量数据迁移阶段 577
增量数据同步阶段 577
流量切换阶段 577
12.6.3 相关参考 577
12.7 数据加密 578
12.7.1 处理流程详解 578
整体架构 578
加密规则 579
加密处理过程 580
12.7.2 解决方案详解 581
12.7.3 中间件加密服务优势 582
12.7.4 加密算法解析 583
EncryptAlgorithm 583
12.8 数据脱敏 583
12.8.1 处理流程详解 583

脱敏规则 584
脱敏处理过程 585
12.9 影子库 586
12.9.1 原理介绍 586
DML 语句 587
DDL 语句 587
12.9.2 相关参考 588
12.10 可观察性 588
12.10.1 原理说明 588
12.11 基础架构 589

13 FAQ **590**
13.1 JDBC 590
13.1.1 引入 `shardingsphere-transaction-xa-core` 后，如何避免 spring-boot 自动加载默认的 JtaTransactionManager? 590
13.1.2 Oracle 表名、字段名配置大小写在加载 `metadata` 元数据时结果不正确? . . . 590
13.1.3 使用 MySQL XA 事务时报 `SQLException: Unable to unwrap to interface com.mysql.jdbc.Connection` 异常 591
13.2 Proxy 591
13.2.1 Windows 环境下，运行 ShardingSphere-Proxy，找不到或无法加载主类 org.apache.shardingsphere.proxy.Bootstrap，如何解决? 591
13.2.2 在使用 ShardingSphere-Proxy 的时候，如何动态在添加新的逻辑库? 592
13.2.3 在使用 ShardingSphere-Proxy 时，怎么使用合适的工具连接到 ShardingSphere-Proxy? 592
13.2.4 使用第三方数据库工具连接 ShardingSphere-Proxy 时，如果 ShardingSphere-Proxy 没有创建 Database 或者没有注册 Storage Unit，连接失败? 592
13.3 分片 593
13.3.1 Cloud not resolve placeholder …in string value …异常的解决方法? 593
13.3.2 inline 表达式返回结果为何出现浮点数? 593
13.3.3 如果只有部分数据库分库分表，是否需要将不分库分表的表也配置在分片规则中? 593
13.3.4 指定了泛型为 Long 的 `SingleKeyTableShardingAlgorithm`，遇到 `ClassCastException: Integer can not cast to Long`? 593
13.3.5 [分片、PROXY] 实现 `StandardShardingAlgorithm` 自定义算法时，指定了 `Comparable` 的具体类型为 Long，且数据库表中字段类型为 bigint，出现 `ClassCastException: Integer can not cast to Long` 异常。 593
13.3.6 ShardingSphere 提供的默认分布式自增主键策略为什么是不连续的，且尾数大多为偶数? 594
13.3.7 如何在 inline 分表策略时，允许执行范围查询操作（BETWEEN AND、>、<、>=、<=)? 594
13.3.8 为什么我实现了 `KeyGenerateAlgorithm` 接口，也配置了 Type，但是自定义的分布式主键依然不生效? 594
13.3.9 ShardingSphere 除了支持自带的分布式自增主键之外，还能否支持原生的自增主键? 594
13.4 单表 595
13.4.1 Table or view %s does not exist. 异常如何解决? 595

13.5 DistSQL 596
13.5.1 使用 DistSQL 添加数据源时，如何设置自定义的 JDBC 连接参数或连接池属性？ . 596
13.5.2 使用 `DistSQL` 删除 `storage unit` 时，出现 `Storage unit [xxx] is still used by [SingleRule]`。 596
13.5.3 使用 `DistSQL` 添加数据源时，出现 `Failed to get driver instance for jdbcURL=xxx`。 596
13.6 其他 596
13.6.1 如果 SQL 在 ShardingSphere 中执行不正确，该如何调试？ 596
13.6.2 阅读源码时为什么会出现编译错误? IDEA 不索引生成的代码？ 597
13.6.3 使用 SQLSever 和 PostgreSQL 时，聚合列不加别名会抛异常？ 597
13.6.4 Oracle 数据库使用 Timestamp 类型的 Order By 语句抛出异常提示“Order by value must implements Comparable”？ 597
13.6.5 Windows 环境下，通过 Git 克隆 ShardingSphere 源码时为什么提示文件名过长，如何解决？ 598
13.6.6 Type is required 异常的解决方法? 599
13.6.7 服务启动时如何加快 `metadata` 加载速度？ 599
13.6.8 ANTLR 插件在 src 同级目录下生成代码，容易误提交，如何避免？ 599
13.6.9 使用 `Proxool` 时分库结果不正确？ 600

14 下载 601
14.1 最新版本 601
14.1.1 Apache ShardingSphere - 版本: 5.5.2 (发布日期: January 16th, 2025) 601
14.2 全部版本 601
14.3 校验版本 601

1 什么是 ShardingSphere

1.1 介绍

Apache ShardingSphere 是一款分布式的数据库生态系统，可以将任意数据库转换为分布式数据库，并通过数据分片、弹性伸缩、加密等能力对原有数据库进行增强。

Apache ShardingSphere 设计哲学为 Database Plus，旨在构建异构数据库上层的标准和生态。它关注如何充分合理地利用数据库的计算和存储能力，而并非实现一个全新的数据库。它站在数据库的上层视角，关注它们之间的协作多于数据库自身。

1.1.1 ShardingSphere-JDBC

ShardingSphere-JDBC 定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。

1.1.2 ShardingSphere-Proxy

ShardingSphere-Proxy 定位为透明化的数据库代理端，通过实现数据库二进制协议，对异构语言提供支持。

1.2 产品功能

1.3 产品优势

- 极致性能

驱动程序端历经长年打磨，效率接近原生 JDBC，性能极致。

- 生态兼容

代理端支持任何通过 MySQL/PostgreSQL 协议的应用访问，驱动程序端可对接任意实现 JDBC 规范的数据库。

- 业务零侵入

面对数据库替换场景，ShardingSphere 可满足业务无需改造，实现平滑业务迁移。

- 运维低成本

在保留原技术栈不变前提下，对 DBA 学习、管理成本低，交互友好。

- 安全稳定

基于成熟数据库底座之上提供增量能力，兼顾安全性及稳定性。

- 弹性扩展

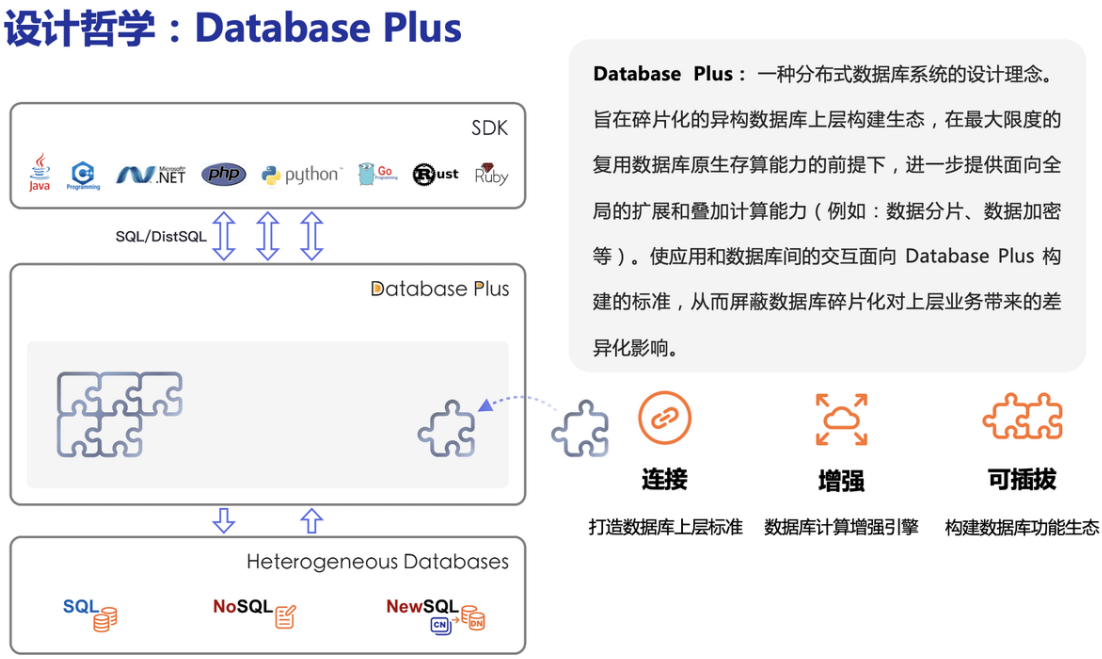
具备计算、存储平滑在线扩展能力，可满足业务多变的需求。

- 开放生态

通过多层次（内核、功能、生态）插件化能力，为用户提供可定制满足自身特殊需求的独有系统。

2 设计哲学

ShardingSphere 采用 Database Plus 设计哲学，该理念致力于构建数据库上层的标准和生态，在生态中补充数据库所缺失的能力。



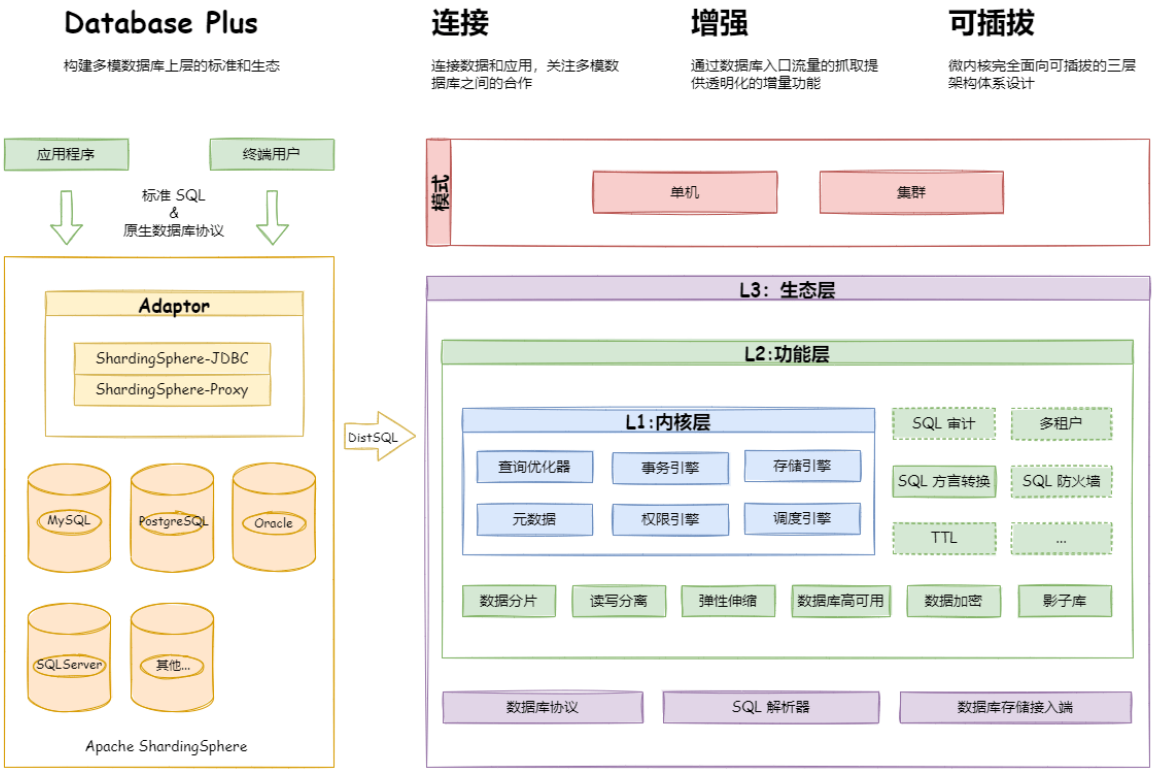
2.1 连接：打造数据库上层标准

通过对数据库协议、SQL 方言以及数据库存储的灵活适配，快速构建多模异构数据库上层的标准，同时通过内置 DistSQL 为应用提供标准化的连接方式。

2.2 增强：数据库计算增强引擎

在原生数据库基础能力之上，提供分布式及流量增强方面的能力。前者可突破底层数据库在计算与存储上的瓶颈，后者通过对流量的变形、重定向、治理、鉴权及分析能力提供更为丰富的数据应用增强能力。

2.3 可插拔：构建数据库功能生态



Apache ShardingSphere 的可插拔架构划分为 3 层，它们是：L1 内核层、L2 功能层、L3 生态层。

2.3.1 L1 内核层

是数据库基本能力的抽象，其所有组件均必须存在，但具体实现方式可通过可插拔的方式更换。主要包括查询优化器、分布式事务引擎、分布式执行引擎、权限引擎和调度引擎等。

2.3.2 L2 功能层

用于提供增量能力，其所有组件均是可选的，可以包含零至多个组件。组件之间完全隔离，互无感知，多组件可通过叠加的方式相互配合使用。主要包括数据分片、读写分离、数据加密、影子库等。用户自定义功能可完全面向 Apache ShardingSphere 定义的顶层接口进行定制化扩展，而无需改动内核代码。

2.3.3 L3 生态层

用于对接和融入现有数据库生态，包括数据库协议、SQL 解析器和存储适配器，分别对应于 Apache ShardingSphere 以数据库协议提供服务的方式、SQL 方言操作数据的方式以及对接存储节点的数据库类型。

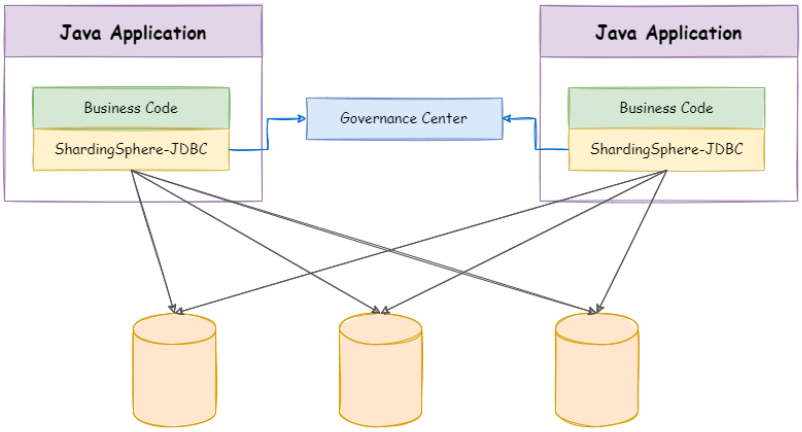
3 部署形态

Apache ShardingSphere 由 ShardingSphere-JDBC 和 ShardingSphere-Proxy 这 2 款既能够独立部署，又支持混合部署配合使用的产品组成。它们均提供标准化的基于数据库作为存储节点的增量功能，可适用于如 Java 同构、异构语言、云原生等各种多样化的应用场景。

3.1 ShardingSphere-JDBC 独立部署

ShardingSphere-JDBC 定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。它使用客户端直连数据库，以 jar 包形式提供服务，无需额外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架。

- 适用于任何基于 JDBC 的 ORM 框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template 或直接使用 JDBC；
- 支持任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, HikariCP 等；
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL，PostgreSQL，Oracle，SQLServer 以及任何可使用 JDBC 访问的数据库。

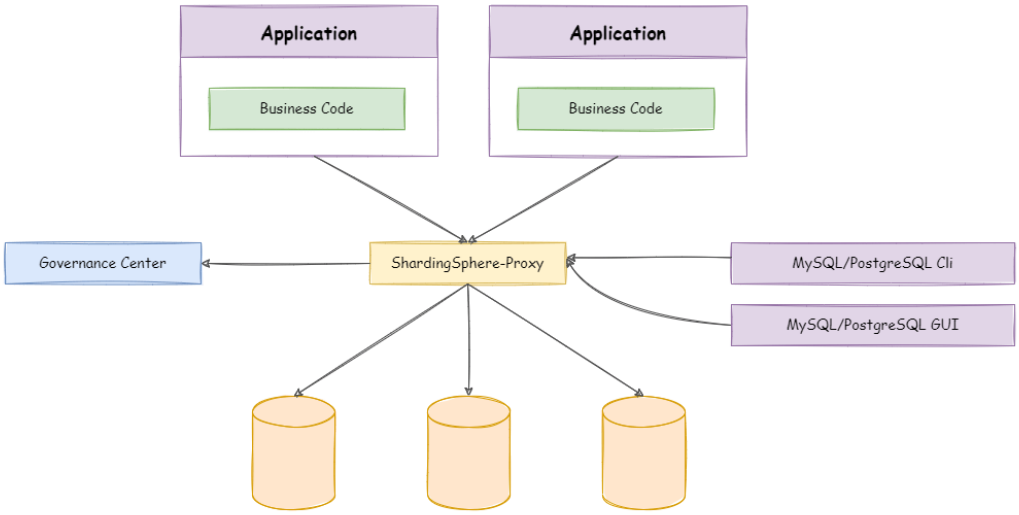


	ShardingSphere-JDBC	ShardingSphere-Proxy
数据库	任意	MySQL/PostgreSQL
连接消耗数	高	低
异构语言	仅 Java	任意
性能	损耗低	损耗略高
无中心化	是	否
静态入口	无	有

3.2 ShardingSphere-Proxy 独立部署

ShardingSphere-Proxy 定位为透明化的数据库代理端，通过实现数据库二进制协议，对异构语言提供支持。目前提供 MySQL 和 PostgreSQL 协议，透明化数据库操作，对 DBA 更加友好。

- 向应用程序完全透明，可直接当做 MySQL/PostgreSQL 使用；
- 兼容 MariaDB 等基于 MySQL 协议的数据库，以及 openGauss 等基于 PostgreSQL 协议的数据库；
- 适用于任何兼容 MySQL/PostgreSQL 协议的的客户端，如：MySQL Command Client, MySQL Workbench, Navicat 等。

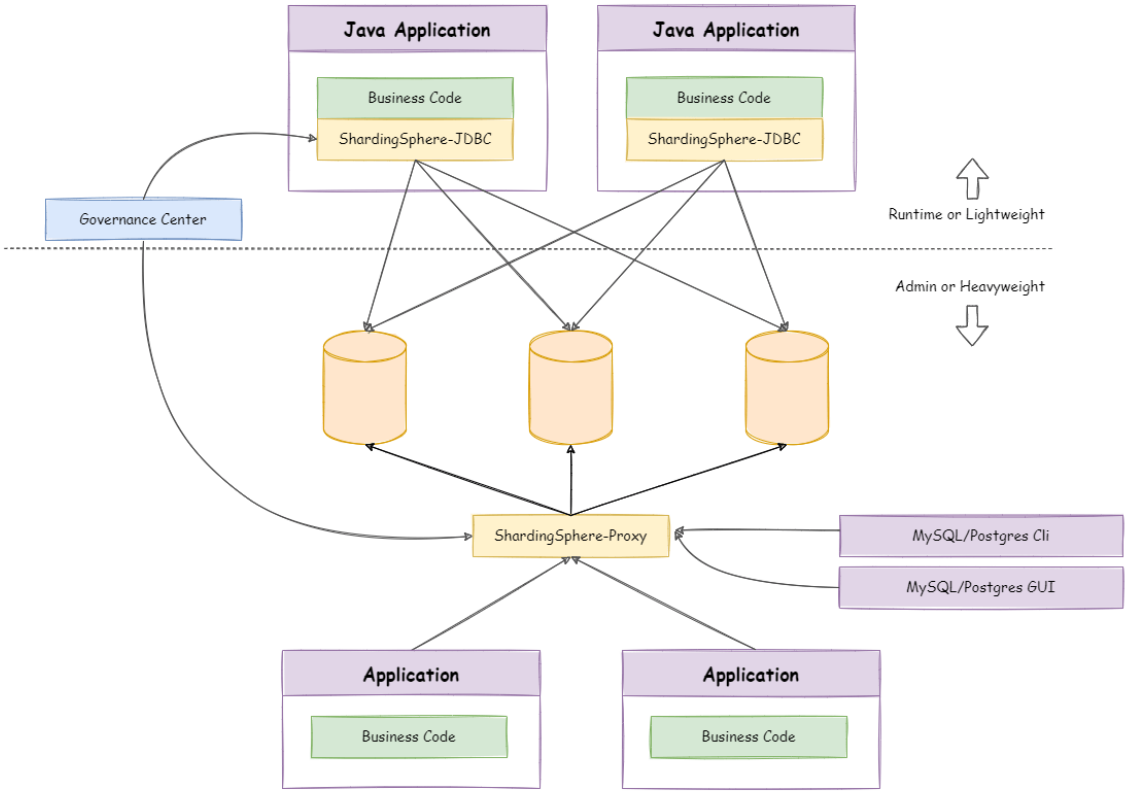


	ShardingSphere-JDBC	ShardingSphere-Proxy
数据库	任意	MySQL/PostgreSQL
连接消耗数	高	低
异构语言	仅 Java	任意
性能	损耗低	损耗略高
无中心化	是	否
静态入口	无	有

3.3 混合部署架构

ShardingSphere-JDBC 采用无中心化架构，与应用程序共享资源，适用于 Java 开发的高性能的轻量级 OLTP 应用；ShardingSphere-Proxy 提供静态入口以及异构语言的支持，独立于应用程序部署，适用于 OLAP 应用以及对分片数据库进行管理和运维的场景。

Apache ShardingSphere 是多接入端共同组成的生态圈。通过混合使用 ShardingSphere-JDBC 和 ShardingSphere-Proxy，并采用同一注册中心统一配置分片策略，能够灵活的搭建适用于各种场景的应用系统，使得架构师更加自由地调整适合于当前业务的最佳系统架构。


Java Application
Business Code
ShardingSphere-JDBC
Java Application
Business Code
ShardingSphere-JDBC
Governance Center
Runtime or Lightweight
Admin or Heavyweight
ShardingSphere-Proxy
MySQL/Postgres Cli
MySQL/Postgres GUI
Application
Business Code
Application
Business Code

4 运行模式

Apache ShardingSphere 提供了两种运行模式，分别是单机模式和集群模式。

4.1 单机模式

能够将数据源和规则等元数据信息持久化，但无法将元数据同步至多个 Apache ShardingSphere 实例，无法在集群环境中相互感知。通过某一实例更新元数据之后，会导致其他实例由于获取不到最新的元数据而产生不一致的错误。

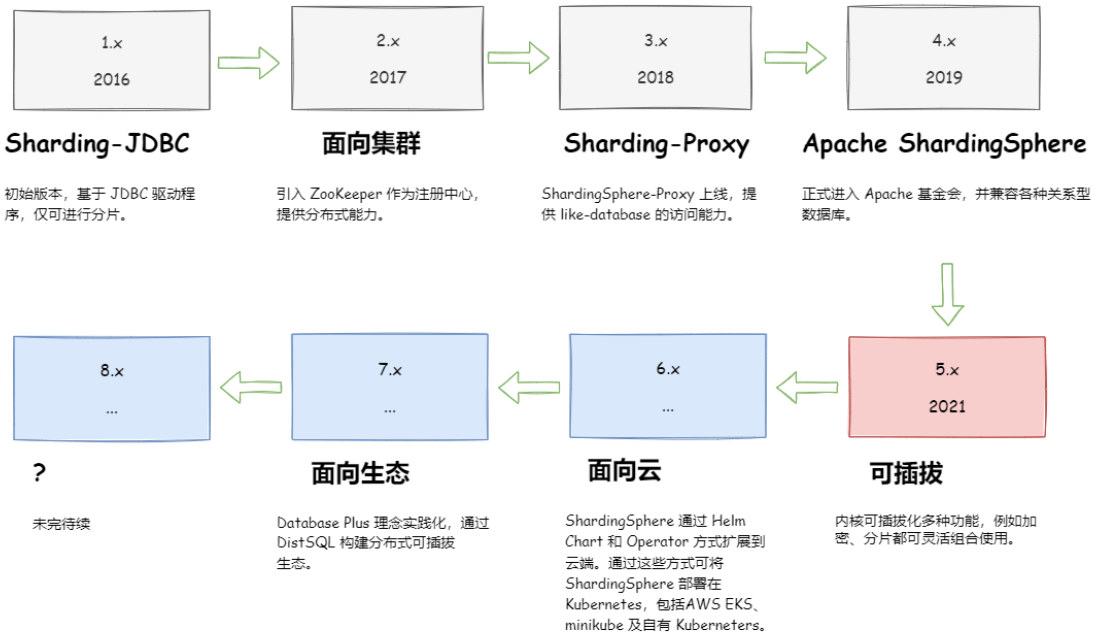
适用于工程师在本地搭建 Apache ShardingSphere 环境。

4.2 集群模式

提供了多个 Apache ShardingSphere 实例之间的元数据共享和分布式场景下状态协调的能力。它能够提供计算能力水平扩展和高可用等分布式系统必备的能力，集群环境需要通过独立部署的注册中心来存储元数据和协调节点状态。

在生产环境建议使用集群模式。

5 线路规划


1.x
2016
Sharding-JDBC
初始版本，基于 JDBC 驱动程序，仅可进行分片。
2.x
2017
面向集群
引入 ZooKeeper 作为注册中心，提供分布式能力。
3.x
2018
Sharding-Proxy
ShardingSphere-Proxy 上线，提供 like-database 的访问能力。
4.x
2019
Apache ShardingSphere
正式进入 Apache 基金会，并兼容各种关系型数据库。
5.x
2021
可插拔
内核可插拔化多种功能，例如加密、分片都可灵活组合使用。
6.x
...
面向云
ShardingSphere 通过 Helm Chart 和 Operator 方式扩展到云端。通过这些方式可将 ShardingSphere 部署在 Kubernetes，包括AWS EKS、minikube 及自有 Kuberneters。
7.x
...
面向生态
Database Plus 理念实践化，通过 DistSQL 构建分布式可插拔生态。
8.x
...
?
未完待续

6 如何参与

ShardingSphere 已于 2020 年 4 月 16 日成为 Apache 软件基金会的顶级项目。欢迎通过邮件列表参与讨论。

7 快速入门

本章节以尽量短的时间，为使用者提供最简单的 Apache ShardingSphere 的快速入门。

示例代码：**https://github.com/apache/shardingsphere/tree/master/examples**

7.1 ShardingSphere-JDBC

7.1.1 应用场景

Apache ShardingSphere-JDBC 可以通过 `Java` 和 `YAML` 这 2 种方式进行配置，开发者可根据场景选择适合的配置方式。

7.1.2 使用限制

目前仅支持 JAVA 语言

7.1.3 前提条件

开发环境需要具备 Java JRE 8 或更高版本。

7.1.4 操作步骤

1. 规则配置。

详情请参见用户手册。

2. 引入 maven 依赖。

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

注意：请将 ${latest.release.version} 更改为实际的版本号。

3. 创建 YAML 配置文件

```
# JDBC 逻辑库名称。在集群模式中，使用该参数来联通 ShardingSphere-JDBC 与 ShardingSphere-Proxy。
# 默认值: logic_db
databaseName (?):

mode:

dataSources:

rules:
- !FOO_XXX
    ...
- !BAR_XXX
    ...

props:
  key_1: value_1
  key_2: value_2
```

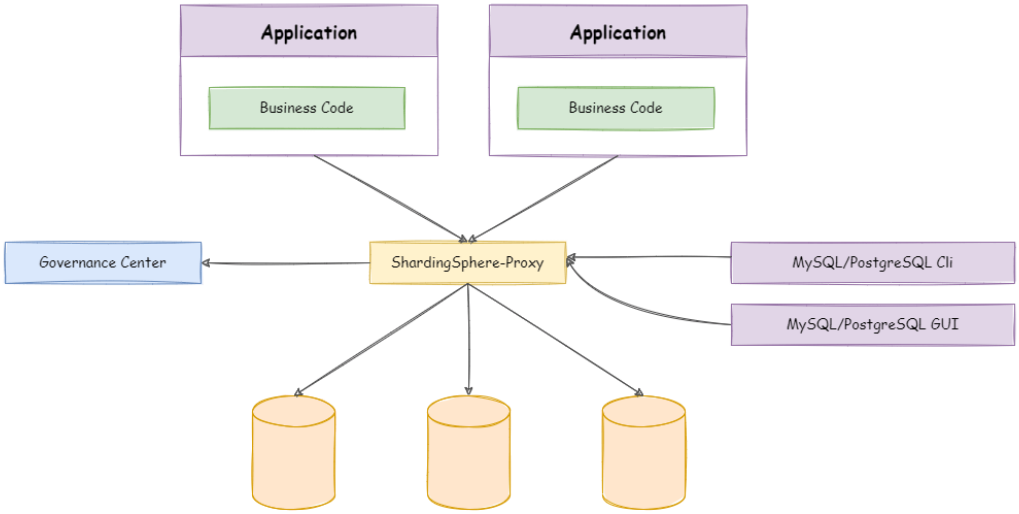
4. 以 spring boot 为例，编辑 application.properties。

```
# 配置 DataSource Driver
spring.datasource.driver-class-name=org.apache.shardingsphere.driver.ShardingSphereDriver
# 指定 YAML 配置文件
spring.datasource.url=jdbc:shardingsphere:classpath:xxx.yaml
```

详情请参见 Spring Boot。

7.2 ShardingSphere-Proxy

7.2.1 应用场景



ShardingSphere-Proxy 的定位为透明化的数据库代理，理论上支持任何使用 MySQL、PostgreSQL、openGauss 协议的客户端操作数据，对异构语言、运维场景更友好。

7.2.2 使用限制

ShardingSphere-Proxy 对系统库/表（如 information_schema、pg_catalog）支持有限，通过部分图形化数据库客户端连接 Proxy 时，可能客户端或 Proxy 会有错误提示。可以使用命令行客户端（`mysql`、`psql`、`gsql` 等）连接 Proxy 验证功能。

7.2.3 前提条件

使用 Docker 启动 ShardingSphere-Proxy 无须额外依赖。使用二进制分发包启动 Proxy，需要环境具备 Java JRE 8 或更高版本。

7.2.4 操作步骤

1. 获取 ShardingSphere-Proxy

目前 ShardingSphere-Proxy 可以通过以下方式：- 二进制发布包 - Docker - Helm

2. 规则配置

编辑 `%SHARDINGSPHERE_PROXY_HOME%/conf/global.yaml`。

编辑 `%SHARDINGSPHERE_PROXY_HOME%/conf/database-xxx.yaml`。

> `%SHARDINGSPHERE_PROXY_HOME%` 为 Proxy 解压后的路径，例：`/opt/shardingsphere-proxy-bin/`

详情请参见 配置手册。

3. 引入依赖

如果后端连接 PostgreSQL 或 openGauss 数据库，不需要引入额外依赖。

如果后端连接 MySQL 数据库，请下载 mysql-connector-java-5.1.49.jar 或者 mysql-connector-java-8.0.11.jar，并将其放入 `%SHARDINGSPHERE_PROXY_HOME%/ext-lib` 目录。

4. 启动服务

- 使用默认配置项

```
sh %SHARDINGSPHERE_PROXY_HOME%/bin/start.sh
```

默认启动端口为 `3307`，默认配置文件目录为：`%SHARDINGSPHERE_PROXY_HOME%/conf/`。

- 自定义端口和配置文件目录

```
sh %SHARDINGSPHERE_PROXY_HOME%/bin/start.sh ${proxy_port} ${proxy_conf_directory}
```

5. 使用 ShardingSphere-Proxy

执行 MySQL / PostgreSQL / openGauss 的客户端命令直接操作 ShardingSphere-Proxy 即可。

使用 MySQL 客户端连接 ShardingSphere-Proxy：

```
mysql -h${proxy_host} -P${proxy_port} -u${proxy_username} -p${proxy_password}
```

使用 PostgreSQL 客户端连接 ShardingSphere-Proxy：

```
psql -h ${proxy_host} -p ${proxy_port} -U ${proxy_username}
```

使用 openGauss 客户端连接 ShardingSphere-Proxy：

```
gsql -r -h ${proxy_host} -p ${proxy_port} -U ${proxy_username} -W ${proxy_password}
```

8 功能

Apache ShardingSphere 提供了多样化的功能，涵盖范围从数据库内核、数据库分布式到贴近数据库上层的应用，为用户提供了大量的功能池。

功能并无边界，只要满足数据库服务和生态的共性需求即可，期待更多的开源工程师参与 Apache ShardingSphere 社区，提供新颖思路和令人兴奋的功能。

8.1 数据分片

8.1.1 背景

传统的将数据集中存储至单一节点的解决方案，在性能、可用性和运维成本这三方面已经难于满足海量数据的场景。

从性能方面来说，由于关系型数据库大多采用 B+ 树类型的索引，在数据量超过阈值的情况下，索引深度的增加也将使得磁盘访问的 IO 次数增加，进而导致查询性能的下降；同时，高并发访问请求也使得集中式数据库成为系统的最大瓶颈。

从可用性的方面来讲，服务化的无状态性，能够达到较小成本的随意扩容，这必然导致系统的最终压力都落在数据库之上。而单一的数据节点，或者简单的主从架构，已经越来越难以承担。数据库的可用性，已成为整个系统的关键。

从运维成本方面考虑，当一个数据库实例中的数据达到阈值以上，对于 DBA 的运维压力就会增大。数据备份和恢复的时间成本都将随着数据量的大小而愈发不可控。一般来讲，单一数据库实例的数据的阈值在 1TB 之内，是比较合理的范围。

在传统的关系型数据库无法满足互联网场景需要的情况下，将数据存储至原生支持分布式的 NoSQL 的尝试越来越多。但 NoSQL 对 SQL 的不兼容性以及生态圈的不完善，使得它们在与关系型数据库的博弈中始终无法完成致命一击，而关系型数据库的地位却依然不可撼动。

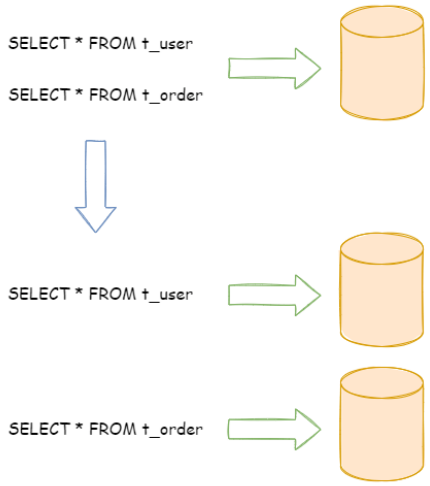
数据分片指按照某个维度将存放在单一数据库中的数据分散地存放至多个数据库或表中以达到提升性能瓶颈以及可用性的效果。数据分片的有效手段是对关系型数据库进行分库和分表。分库和分表均可以有效的避免由数据量超过可承受阈值而产生的查询瓶颈。除此之外，分库还能够用于有效的分散对数据库单点的访问量；分表虽然无法缓解数据库压力，但却能够提供尽量将分布式事务转化为本地事务的可能，

一旦涉及到跨库的更新操作，分布式事务往往会使问题变得复杂。使用多主多从的分片方式，可以有效的避免数据单点，从而提升数据架构的可用性。

通过分库和分表进行数据的拆分来使得各个表的数据量保持在阈值以下，以及对流量进行疏导应对高访问量，是应对高并发和海量数据系统的有效手段。数据分片的拆分方式又分为垂直分片和水平分片。

垂直分片

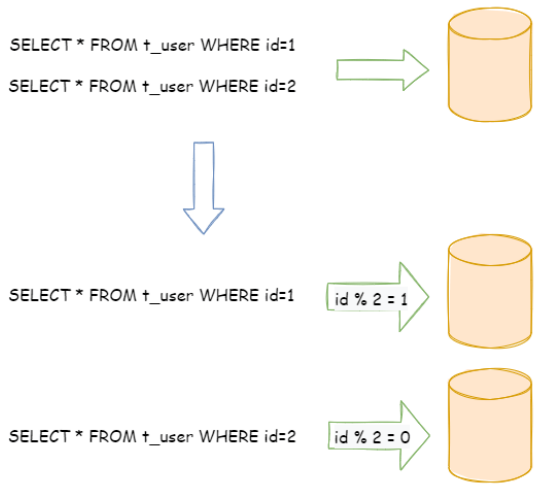
按照业务拆分的方式称为垂直分片，又称为纵向拆分，它的核心理念是专库专用。在拆分之前，一个数据库由多个数据表构成，每个表对应着不同的业务。而拆分之后，则是按照业务将表进行归类，分布到不同的数据库中，从而将压力分散至不同的数据库。下图展示了根据业务需要，将用户表和订单表垂直分片到不同的数据库的方案。



垂直分片往往需要对架构和设计进行调整。通常来讲，是来不及应对互联网业务需求快速变化的；而且，它也并无法真正的解决单点瓶颈。垂直拆分可以缓解数据量和访问量带来的问题，但无法根治。如果垂直拆分之后，表中的数据量依然超过单节点所能承载的阈值，则需要水平分片来进一步处理。

水平分片

水平分片又称为横向拆分。相对于垂直分片，它不再将数据根据业务逻辑分类，而是通过某个字段（或某几个字段），根据某种规则将数据分散至多个库或表中，每个分片仅包含数据的一部分。例如：根据主键分片，偶数主键的记录放入 0 库（或表），奇数主键的记录放入 1 库（或表），如下图所示。



水平分片从理论上突破了单机数据量处理的瓶颈，并且扩展相对自由，是数据分片的标准解决方案。

8.1.2 挑战

虽然数据分片解决了性能、可用性以及单点备份恢复等问题，但分布式的架构在获得了收益的同时，也引入了新的问题。

面对如此散乱的分片之后的数据，应用开发工程师和数据库管理员对数据库的操作变得异常繁重就是其中的重要挑战之一。他们需要知道数据需要从哪个具体的数据库的子表中获取。

另一个挑战则是，能够正确的运行在单节点数据库中的 SQL，在分片之后的数据库中并不一定能够正确运行。例如，分表导致表名称的修改，或者分页、排序、聚合分组等操作的不正确处理。

跨库事务也是分布式的数据库集群要面对的棘手事情。合理采用分表，可以在降低单表数据量的情况下，尽量使用本地事务，善于使用同库不同表可有效避免分布式事务带来的麻烦。在不能避免跨库事务的场景，有些业务仍然需要保持事务的一致性。而基于 XA 的分布式事务由于在并发度高的场景中性能无法满足需要，并未被互联网巨头大规模使用，他们大多采用最终一致性的柔性事务代替强一致事务。

8.1.3 目标

尽量透明化分库分表所带来的影响，让使用方尽量像使用一个数据库一样使用水平分片之后的数据库集群，是 Apache ShardingSphere 数据分片模块的主要设计目标。

8.1.4 应用场景

海量数据高并发的 OLTP 场景

由于关系型数据库大多采用 B+ 树类型的索引，在数据量超过阈值的情况下，索引深度的增加也将使得磁盘访问的 IO 次数增加，进而导致查询性能的下降。通过 ShardingSphere 数据分片，按照某个业务维度，将存放在单一数据库中的数据分散地存放至多个数据库或表中，可以达到提升性能的效果。通过使用 ShardingSphere-JDBC 接入端，可以满足高并发的 OLTP 场景下的性能要求。

海量数据实时分析 OLAP 场景

在传统的数据库架构中，如果用户想要进行数据分析，需要先使用 ETL 工具，将数据同步至数据平台中，然后再进行数据分析，使用 ETL 工具会导致数据分析的实效性大打折扣。ShardingSphere-Proxy 提供静态入口以及异构语言的支持，独立于应用程序部署，适用于实时分析的 OLAP 场景。

8.1.5 相关参考

- 数据分片的配置
- 数据分片的开发者指南

8.1.6 核心概念

表

表是透明化数据分片的关键概念。Apache ShardingSphere 通过提供多样化的表类型，适配不同场景下的数据分片需求。

逻辑表

相同结构的水平拆分数据库（表）的逻辑名称，是 SQL 中表的逻辑标识。例：订单数据根据主键尾数拆分为 10 张表，分别是 `t_order_0` 到 `t_order_9`，他们的逻辑表名为 `t_order`。

真实表

在水平拆分的数据库中真实存在的物理表。即上个示例中的 `t_order_0` 到 `t_order_9`。

绑定表

指分片规则一致的一组分片表。使用绑定表进行多表关联查询时，必须使用分片键进行关联，否则会出现笛卡尔积关联或跨库关联，从而影响查询效率。例如：`t_order` 表和 `t_order_item` 表，均按照 `order_id` 分片，并且使用 `order_id` 进行关联，则此两张表互为绑定表关系。绑定表之间的多表关联查询不会出现笛卡尔积关联，关联查询效率将大大提升。举例说明，如果 SQL 为：

```
SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);
```

在不配置绑定表关系时，假设分片键 order_id 将数值 10 路由至第 0 片，将数值 11 路由至第 1 片，那么路由后的 SQL 应该为 4 条，它们呈现为笛卡尔积：

```
SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_0 o JOIN t_order_item_1 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_0 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);
```

在配置绑定表关系，并且使用 `order_id` 进行关联后，路由的 SQL 应该为 2 条：

```
SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);
```

其中 `t_order` 表由于指定了分片条件，ShardingSphere 将会以它作为整个绑定表的主表。所有路由计算将会只使用主表的策略，那么 `t_order_item` 表的分片计算将会使用 `t_order` 的条件。

注意：绑定表中的多个分片规则，需要按照逻辑表前缀组合分片后缀的方式进行配置，例如：

```
rules:
- !SHARDING
  tables:
    t_order:
      actualDataNodes: ds_${0..1}.t_order_${0..1}
    t_order_item:
```

```
    actualDataNodes: ds_${0..1}.t_order_item_${0..1}
  bindingTables:
    - t_order, t_order_item
```

广播表

指所有的数据源中都存在的表，表结构及其数据在每个数据库中均完全一致。适用于数据量不大且需要与海量数据的表进行关联查询的场景，例如：字典表。

单表

指所有的分片数据源中仅唯一存在的表。适用于数据量不大且无需分片的表。

注意：符合以下条件的单表会被自动加载：- 数据加密、数据脱敏等规则中显示配置的单表 - 用户通过 ShardingSphere 执行 DDL 语句创建的单表

其余不符合上述条件的单表，ShardingSphere 不会自动加载，用户可根据需要配置单表规则进行管理。

数据节点

数据分片的最小单元，由数据源名称和真实表组成。例：ds_0.t_order_0。逻辑表与真实表的映射关系，可分为均匀分布和自定义分布两种形式。

均匀分布

指数据表在每个数据源内呈现均匀分布的态势，例如：

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order0
  └── t_order1
```

数据节点的配置如下：

```
db0.t_order0, db0.t_order1, db1.t_order0, db1.t_order1
```

自定义分布

指数据表呈现有特定规则的分布，例如：

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order2
  ├── t_order3
  └── t_order4
```

数据节点的配置如下：

```
db0.t_order0, db0.t_order1, db1.t_order2, db1.t_order3, db1.t_order4
```

分片

分片键

用于将数据库（表）水平拆分的数据库字段。例：将订单表中的订单主键的尾数取模分片，则订单主键为分片字段。SQL 中如果无分片字段，将执行全路由，性能较差。除了对单分片字段的支持，Apache ShardingSphere 也支持根据多个字段进行分片。

分片算法

用于将数据分片的算法，支持 =、>=、<=、>、<、BETWEEN 和 IN 进行分片。分片算法可由开发者自行实现，也可使用 Apache ShardingSphere 内置的分片算法语法糖，灵活度非常高。

自动化分片算法

分片算法语法糖，用于便捷的托管所有数据节点，使用者无需关注真实表的物理分布。包括取模、哈希、范围、时间等常用分片算法的实现。

自定义分片算法

提供接口让应用开发者自行实现与业务实现紧密相关的分片算法，并允许使用者自行管理真实表的物理分布。自定义分片算法又分为：

- 标准分片算法

用于处理使用单一键作为分片键的 =、IN、BETWEEN AND、>、<、>=、<= 进行分片的场景。

- 复合分片算法

用于处理使用多键作为分片键进行分片的场景，包含多个分片键的逻辑较复杂，需要应用开发者自行处理其中的复杂度。

- Hint 分片算法

用于处理使用 `Hint` 行分片的场景。

分片策略

包含分片键和分片算法，由于分片算法的独立性，将其独立抽离。真正可用于分片操作的是分片键 + 分片算法，也就是分片策略。

强制分片路由

对于分片字段并非由 SQL 而是其他外置条件决定的场景，可使用 SQL Hint 注入分片值。例：按照员工登录主键分库，而数据库中并无此字段。SQL Hint 支持通过 Java API 和 SQL 注释两种方式使用。详情请参见强制分片路由。

行表达式

行表达式是为了解决配置的简化与一体化这两个主要问题。在繁琐的数据分片规则配置中，随着数据节点的增多，大量的重复配置使得配置本身不易被维护。通过行表达式可以有效地简化数据节点配置工作量。

对于常见的分片算法，使用 Java 代码实现并不有助于配置的统一管理。通过行表达式书写分片算法，可以有效地将规则配置一同存放，更加易于浏览与存储。

行表达式作为字符串由两部分组成，分别是字符串开头的对应 SPI 实现的 Type Name 部分和表达式部分。以 `<GROOVY>t_order_${1..3}` 为例，字符串`<GROOVY>` 部分的子字符串 `GROOVY` 为此行表达式使用的对应 SPI 实现的 Type Name，其被 `<>` 符号包裹来识别。而字符串 `t_order_${1..3}` 为此行表达式的表达式部分。当行表达式不指定 Type Name 时，例如 `t_order_${1..3}`，行表示式默认将使用 `InlineExpressionParser` SPI 的 `GROOVY` 实现来解析表达式。

以下部分介绍 `GROOVY` 实现的语法规则。

行表达式的使用非常直观，只需要在配置中使用 `${ expression }` 或 `$->{ expression }` 标识行表达式即可。目前支持数据节点和分片算法这两个部分的配置。行表达式的内容使用的是 Groovy 的语法，Groovy 能够支持的所有操作，行表达式均能够支持。例如：

`${begin..end}` 表示范围区间 `${[unit1, unit2, unit_x]}` 表示枚举值

行表达式中如果出现连续多个 `${ expression }` 或 `$->{ expression }` 表达式，整个表达式最终的结果将会根据每个子表达式的结果进行笛卡尔组合。

例如，以下行表达式：

```
${['online', 'offline']}_table${1..3}
```

最终会解析为：

```
online_table1, online_table2, online_table3, offline_table1, offline_table2,
offline_table3
```

分布式主键

传统数据库软件开发中，主键自动生成技术是基本需求。而各个数据库对于该需求也提供了相应的支持，比如 MySQL 的自增键，Oracle 的自增序列等。数据分片后，不同数据节点生成全局唯一主键是非常棘手的问题。同一个逻辑表内的不同实际表之间的自增键由于无法互相感知而产生重复主键。虽然可通过约束自增主键初始值和步长的方式避免碰撞，但需引入额外的运维规则，使解决方案缺乏完整性和可扩展性。

目前有许多第三方解决方案可以完美解决这个问题，如 UUID 等依靠特定算法自生成不重复键，或者通过引入主键生成服务等。为了方便用户使用、满足不同用户不同使用场景的需求，Apache ShardingSphere 不仅提供了内置的分布式主键生成器，例如 UUID、SNOWFLAKE，还抽离出分布式主键生成器的接口，方便用户自行实现自定义的自增主键生成器。

8.1.7 使用限制

兼容全部常用的路由至单数据节点的 SQL；路由至多数据节点的 SQL 由于场景复杂，分为稳定支持、实验性支持和不支持这三种情况。

稳定支持

全面支持 DML、DDL、DCL、TCL 和常用 DAL。支持分页、去重、排序、分组、聚合、表关联等复杂查询。支持 PostgreSQL 和 openGauss 数据库的 schema DDL 和 DML 语句，当 SQL 中不指定 schema 时，默认访问 `public` schema，其他 schema 则需要在表名前显示声明，暂不支持 `SEARCH_PATH` 修改 schema 搜索路径。

常规查询

- SELECT 主语句

```
SELECT select_expr [, select_expr ...] FROM table_reference [, table_reference ...]
[WHERE predicates]
[GROUP BY {col_name | position} [ASC | DESC], ...]
[ORDER BY {col_name | position} [ASC | DESC], ...]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

- select_expr

```
* |
[DISTINCT] COLUMN_NAME [AS] [alias] |
(MAX | MIN | SUM | AVG)(COLUMN_NAME | alias) [AS] [alias] |
COUNT(* | COLUMN_NAME | alias) [AS] [alias]
```

- table_reference

```
tbl_name [AS] alias] [index_hint_list]
| table_reference ([INNER] | {LEFT|RIGHT} [OUTER]) JOIN table_factor [JOIN ON
conditional_expr | USING (column_list)]
```

子查询

子查询和外层查询同时指定分片键，且分片键的值保持一致时，由内核提供稳定支持。

例如：

```
SELECT * FROM (SELECT * FROM t_order WHERE order_id = 1) o WHERE o.order_id = 1;
```

用于分页的子查询，由内核提供稳定支持。

例如：

```
SELECT * FROM (SELECT row_.*, rownum rownum_ FROM (SELECT * FROM t_order) row_
WHERE rownum <= ?) WHERE rownum > ?;
```

分页查询

完全支持 MySQL、PostgreSQL、openGauss，Oracle 和 SQLServer 由于分页查询较为复杂，仅部分支持。

Oracle 和 SQLServer 的分页都需要通过子查询来处理，ShardingSphere 支持分页相关的子查询。

- Oracle

支持使用 rownum 进行分页：

```
SELECT * FROM (SELECT row_.*, rownum rownum_ FROM (SELECT o.order_id as order_id
FROM t_order o JOIN t_order_item i ON o.order_id = i.order_id) row_ WHERE rownum <=
?) WHERE rownum > ?
```

- SQLServer

支持使用 TOP + ROW_NUMBER() OVER 配合进行分页：

```
SELECT * FROM (SELECT TOP (?) ROW_NUMBER() OVER (ORDER BY o.order_id DESC) AS
rownum, * FROM t_order o) AS temp WHERE temp.rownum > ? ORDER BY temp.order_id
```

支持 SQLServer 2012 之后的 OFFSET FETCH 的分页方式：

```
SELECT * FROM t_order o ORDER BY id OFFSET ? ROW FETCH NEXT ? ROWS ONLY
```

- MySQL, PostgreSQL 和 openGauss

MySQL、PostgreSQL 和 openGauss 都支持 LIMIT 分页，无需子查询：

```
SELECT * FROM t_order o ORDER BY id LIMIT ? OFFSET ?
```

聚合查询

支持 MAX, MIN, SUM, COUNT, AVG, BIT_XOR, GROUP_CONCAT 聚合语法。

运算表达式中包含分片键

当分片键处于运算表达式中时，无法通过 SQL 字面提取用于分片的值，将导致全路由。例如，假设 create_time 为分片键：

```
SELECT * FROM t_order WHERE to_date(create_time, 'yyyy-mm-dd') = '2019-01-01';
```

LOAD DATA / LOAD XML

支持 MySQL LOAD DATA 和 LOAD XML 语句加载数据到单表和广播表。

视图

1. 支持基于单个单表，或多个相同存储节点的单表创建、修改和删除视图；
2. 支持基于任意个广播表创建、修改和删除视图；
3. 支持基于任意个分片表创建、修改和删除视图，视图必须和分片表一样配置分片规则，并且视图和分片表必须为绑定表关系；
4. 支持基于广播表和分片表创建、修改和删除视图，分片表规则同单独使用分片表创建视图；
5. 支持基于广播表和单表创建、修改和删除视图；
6. 支持 MySQL SHOW CREATE TABLE viewName 查看视图的创建语句。

实验性支持

实验性支持特指使用 Federation 执行引擎提供支持。该引擎处于快速开发中，用户虽基本可用，但仍需大量优化，是实验性产品。

子查询

子查询和外层查询未同时指定分片键，或分片键的值不一致时，由 Federation 执行引擎提供支持。

例如：

```
SELECT * FROM (SELECT * FROM t_order) o;

SELECT * FROM (SELECT * FROM t_order) o WHERE o.order_id = 1;

SELECT * FROM (SELECT * FROM t_order WHERE order_id = 1) o;
```

```
SELECT * FROM (SELECT * FROM t_order WHERE order_id = 1) o WHERE o.order_id = 2;
```

跨库关联查询

当关联查询中的多个表分布在不同的数据库实例上时，由 Federation 执行引擎提供支持。假设 `t_order` 和 `t_order_item` 是多数据节点的分片表，并且未配置绑定表规则，`t_user` 和 `t_user_role` 是分布在不同的数据库实例上的单表，那么 Federation 执行引擎能够支持如下常用的关联查询：

```
SELECT * FROM t_order o INNER JOIN t_order_item i ON o.order_id = i.order_id WHERE
o.order_id = 1;

SELECT * FROM t_order o INNER JOIN t_user u ON o.user_id = u.user_id WHERE o.user_
id = 1;

SELECT * FROM t_order o LEFT JOIN t_user_role r ON o.user_id = r.user_id WHERE o.
user_id = 1;

SELECT * FROM t_order_item i LEFT JOIN t_user u ON i.user_id = u.user_id WHERE i.
user_id = 1;

SELECT * FROM t_order_item i RIGHT JOIN t_user_role r ON i.user_id = r.user_id
WHERE i.user_id = 1;

SELECT * FROM t_user u RIGHT JOIN t_user_role r ON u.user_id = r.user_id WHERE u.
user_id = 1;
```

不支持

CASE WHEN

以下 `CASE WHEN` 语句不支持：

- `CASE WHEN` 中包含子查询
- `CASE WHEN` 中使用逻辑表名（请使用表别名）

分页查询

Oracle 和 SQLServer 由于分页查询较为复杂，目前有部分分页查询不支持，具体如下：

- Oracle

目前不支持 `rownum + BETWEEN` 的分页方式。

- SQLServer

目前不支持使用 `WITH xxx AS (SELECT …)` 的方式进行分页。由于 Hibernate 自动生成的 SQLServer 分页语句使用了 `WITH` 语句，因此目前并不支持基于 Hibernate 的 SQLServer 分页。目前也不支持使用两个 TOP + 子查询的方式实现分页。

聚合查询

查询中同时包含多个聚合函数时，不支持带 DISTINCT 的聚合函数和不带 DISTINCT 的聚合函数混合使用。

LOAD DATA / LOAD XML

不支持 MySQL `LOAD DATA` 和 `LOAD XML` 语句加载数据到分片表。

8.1.8 附录

有限支持的 SQL：

- 使用 JDBC 规范 `getGeneratedKeys` 接口返回自增主键时，需要配合使用支持自增的分布式主键生成器，不支持其他类型的分布式主键生成器

不支持的 SQL：

- CASE WHEN 中包含子查询
- CASE WHEN 中使用逻辑表名（请使用表别名）
- INSERT INTO tbl_name (col1, col2, …) SELECT * FROM tbl_name WHERE col3 = ?（SELECT 子句不支持 * 和内置分布式主键生成器）
- REPLACE INTO tbl_name (col1, col2, …) SELECT * FROM tbl_name WHERE col3 = ?（SELECT 子句不支持 * 和内置分布式主键生成器）
- SELECT MAX(tbl_name.col1) FROM tbl_name（查询列是函数表达式时，查询列前不能使用表名，可以使用表别名）

其他：

- 分片规则中配置的真实表、分片列和分布式序列需要和数据库中的列保持大小写一致。

8.2 分布式事务

8.2.1 背景

数据库事务需要满足 ACID（原子性、一致性、隔离性、持久性）四个特性。

- 原子性（Atomicity）指事务作为整体来执行，要么全部执行，要么全不执行；
- 一致性（Consistency）指事务应确保数据从一个一致的状态转变为另一个一致的状态；
- 隔离性（Isolation）指多个事务并发执行时，一个事务的执行不应影响其他事务的执行；

- 持久性（Durability）指已提交的事务修改数据会被持久保存。

在单一数据节点中，事务仅限于对单一数据库资源的访问控制，称之为本地事务。几乎所有的成熟的关系型数据库都提供了对本地事务的原生支持。但是在基于微服务的分布式应用环境下，越来越多的应用场景要求对多个服务的访问及其相对应的多个数据库资源能纳入到同一个事务当中，分布式事务应运而生。

关系型数据库虽然对本地事务提供了完美的 ACID 原生支持。但在分布式的场景下，它却成为系统性能的桎梏。如何让数据库在分布式场景下满足 ACID 的特性或找寻相应的替代方案，是分布式事务的重点工作。

8.2.2 挑战

由于应用的场景不同，需要开发者能够合理的在性能与功能之间权衡各种分布式事务。

强一致的事务与柔性事务的 API 和功能并不完全相同，在它们之间并不能做到自由的透明切换。在开发决策阶段，就不得不在强一致的事务和柔性事务之间抉择，使得设计和开发成本被大幅增加。

基于 XA 的强一致事务使用相对简单，但是无法很好的应对互联网的高并发或复杂系统的长事务场景；柔性事务则需要开发者对应用进行改造，接入成本非常高，并且需要开发者自行实现资源锁定和反向补偿。

8.2.3 目标

整合现有的成熟事务方案，为本地事务、两阶段事务和柔性事务提供统一的分布式事务接口，并弥补当前方案的不足，提供一站式的分布式事务解决方案是 Apache ShardingSphere 分布式事务模块的主要设计目标。

8.2.4 原理介绍

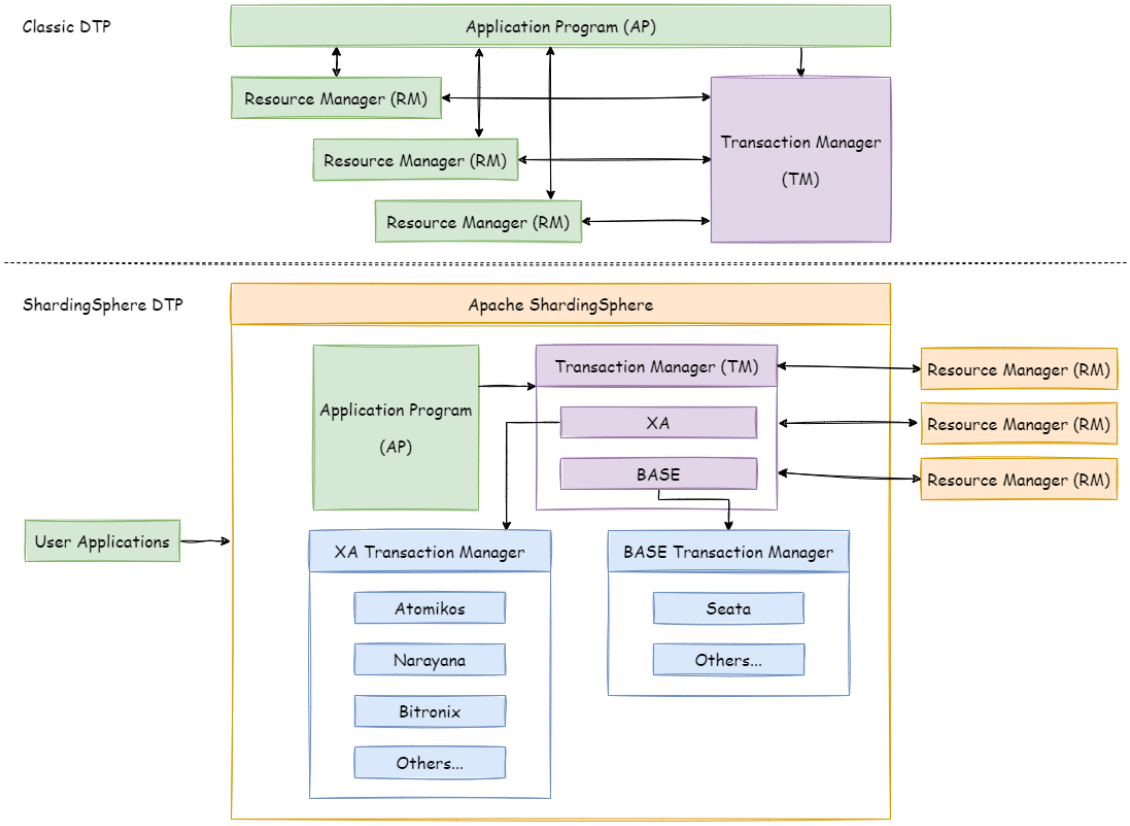
ShardingSphere 对外提供 begin/commit/rollback 传统事务接口，通过 LOCAL，XA，BASE 三种模式提供了分布式事务的能力，

LOCAL 事务

LOCAL 模式基于 ShardingSphere 代理的数据库 `begin/commit/rolllback` 的接口实现，对于一条逻辑 SQL，ShardingSphere 通过 `begin` 指令在每个被代理的数据库开启事务，并执行实际 SQL，并执行 `commit/rollback`。由于每个数据节点各自管理自己的事务，它们之间没有协调以及通信的能力，也并不互相知晓其他数据节点事务的成功与否。在性能方面无任何损耗，但在强一致性以及最终一致性方面不能够保证。

XA 事务

XA 事务采用的是 X/OPEN 组织所定义的 DTP 模型 所抽象的 AP（应用程序）, TM（事务管理器）和 RM（资源管理器）概念来保证分布式事务的强一致性。其中 TM 与 RM 间采用 XA 的协议进行双向通信，通过两阶段提交实现。与传统的本地事务相比，XA 事务增加了准备阶段，数据库除了被动接受提交指令外，还可以反向通知调用方事务是否可以被提交。`TM` 可以收集所有分支事务的准备结果，并于最后进行原子提交，以保证事务的强一致性。



XA 事务建立在 ShardingSphere 代理的数据库 xa start/end/prepare/commit/rollback/recover 的接口上。

对于一条逻辑 SQL，ShardingSphere 通过 `xa begin` 指令在每个被代理的数据库开启事务，内部集成 TM，用于协调各分支事务，并执行 `xa commit/rollback`。

基于 XA 协议实现的分布式事务，由于在执行的过程中需要对所需资源进行锁定，它更加适用于执行时间确定的短事务。对于长事务来说，整个事务进行期间对数据的独占，将会对并发场景下的性能产生一定的影响。

BASE 事务

如果将实现了 ACID 的事务要素的事务称为刚性事务的话，那么基于 BASE 事务要素的事务则称为柔性事务。BASE 是基本可用、柔性状态和最终一致性这三个要素的缩写。

- 基本可用（Basically Available）保证分布式事务参与方不一定同时在线；
- 柔性状态（Soft state）则允许系统状态更新有一定的延时，这个延时对客户来说不一定能够察觉；
- 最终一致性（Eventually consistent）通常是通过消息传递的方式保证系统的最终一致性。

在 ACID 事务中对隔离性的要求很高，在事务执行过程中，必须将所有的资源锁定。柔性事务的理念则是通过业务逻辑将互斥锁操作从资源层面上移至业务层面。通过放宽对强一致性要求，来换取系统吞吐量的提升。

基于 ACID 的强一致性事务和基于 BASE 的最终一致性事务都不是银弹，只有在最适合的场景中才能发挥它们的最大长处。Apache ShardingSphere 集成了 SEATA 作为柔性事务的使用方案。可通过下表详细对比它们之间的区别，以帮助开发者进行技术选型。

	LOCAL	*XA*	*BASE*
业务改造	无	无	需要 Seata Server
一致性	不支持	支持	最终一致
隔离性	不支持	支持	业务方保证
并发性能	无影响	严重衰退	略微衰退
适合场景	业务方处理不一致	短事务 & 低并发	长事务 & 高并发

8.2.5 应用场景

在单机应用场景中，依赖数据库提供的事务即可满足业务上对事务 ACID 的需求。但是在分布式场景下，传统数据库解决方案缺乏对全局事务的管控能力，用户在使用过程中可能遇到多个数据库节点上出现数据不一致的问题。

ShardingSphere 分布式事务，为用户屏蔽了分布式事务处理的复杂性，提供了灵活多样的分布式事务解决方案，用户可以根据自己的业务场景在 LOCAL，XA，BASE 三种模式中，选择适合自己的分布式事务解决方案。

ShardingSphere XA 事务使用场景

对于 XA 事务，提供了分布式环境下，对数据强一致性的保证。但是由于存在同步阻塞问题，对性能会有一定影响。适用于对数据一致性要求非常高且对并发性能要求不是很高的业务场景。

ShardingSphere BASE 事务使用场景

对于 BASE 事务，提供了分布式环境下，对数据最终一致性的保证。由于在整个事务过程中，不会像 XA 事务那样全程锁定资源，所以性能较好。适用于对并发性能要求很高并且允许出现短暂数据不一致的业务场景。

ShardingSphere LOCAL 事务使用场景

对于 LOCAL 事务，在分布式环境下，不保证各个数据库节点之间数据的一致性和隔离性，需要业务方自行处理可能出现的不一致问题。适用于用户希望自行处理分布式环境下数据一致性问题的业务场景。

8.2.6 相关参考

- 分布式事务的 YAML 配置

8.2.7 核心概念

XA 协议

XA 协议最早的分布式事务模型是由 `X/Open` 国际联盟提出的 `X/Open Distributed Transaction Processing (DTP)` 模型，简称 XA 协议。

8.2.8 使用限制

虽然 Apache ShardingSphere 希望能够完全兼容所有的分布式事务场景，并在性能上达到最优，但在 CAP 定理所指导下，分布式事务必然有所取舍。Apache ShardingSphere 希望能够将分布式事务的选择权交给使用者，在不同的场景使用最适合的分布式事务解决方案。

LOCAL 事务

不支持项

- 不支持因网络、硬件异常导致的跨库事务。例如：同一事务中，跨两个库更新，更新完毕后、未提交之前，第一个库宕机，则只有第二个库数据提交，且无法回滚。

XA 事务

不支持项

- 服务宕机后，在其它机器上恢复提交/回滚中的数据；
- MySQL 事务块内，SQL 执行出现异常，执行 `Commit`，数据保持一致；
- 配置 XA 事务后，存储单元名称最大长度不超过 45 个字符。

BASE 事务

不支持项

- 不支持隔离级别。

8.2.9 附录

不支持的 SQL：

- 事务中使用 DistSQL 里的 RAL、RDL 操作；
- XA 事务中使用 DDL 语句。

XA 事务所需的权限：

在 MySQL8 中需要授予用户 `XA_RECOVER_ADMIN` 权限，否则 XA 事务管理器执行 `XA RECOVER` 语句时会报错。

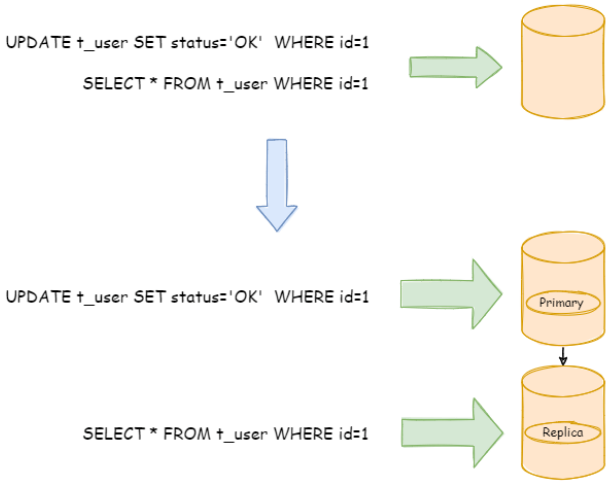
8.3 读写分离

8.3.1 背景

面对日益增加的系统访问量，数据库的吞吐量面临着巨大瓶颈。对于同一时刻有大量并发读操作和较少写操作类型的应用系统来说，将数据库拆分为主库和从库，主库负责处理事务性的增删改操作，从库负责处理查询操作，能够有效的避免由数据更新导致的行锁，使得整个系统的查询性能得到极大的改善。

通过一主多从的配置方式，可以将查询请求均匀的分散到多个数据副本，能够进一步的提升系统的处理能力。使用多主多从的方式，不但能够提升系统的吞吐量，还能够提升系统的可用性，可以达到在任何一个数据库宕机，甚至磁盘物理损坏的情况下仍然不影响系统的正常运行。

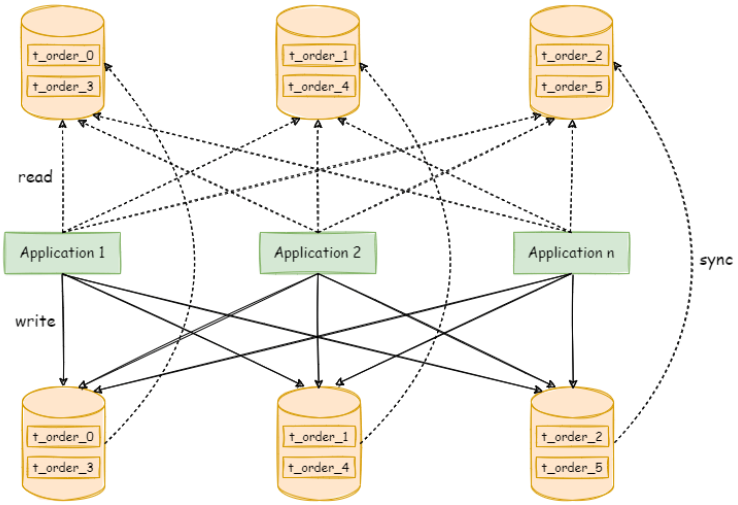
与将数据根据分片键打散至各个数据节点的水平分片不同，读写分离则是根据 SQL 语义的分析，将读操作和写操作分别路由至主库与从库。



读写分离的数据节点中的数据内容是一致的，而水平分片的每个数据节点的数据内容却并不相同。将水平分片和读写分离联合使用，能够更加有效的提升系统性能。

8.3.2 挑战

读写分离虽然可以提升系统的吞吐量和可用性，但同时也带来了数据不一致的问题。这包括多个主库之间的数据一致性，以及主库与从库之间的数据一致性的问题。并且，读写分离也带来了与数据分片同样的问题，它同样会使得应用开发和运维人员对数据库的操作和运维变得更加复杂。下图展现了将数据分片与读写分离一同使用时，应用程序与数据库集群之间的复杂拓扑关系。



8.3.3 目标

透明化读写分离所带来的影响，让使用方尽量像使用一个数据库一样使用主从数据库集群，是 Apache ShardingSphere 读写分离模块的主要设计目标。

8.3.4 应用场景

复杂的主从数据库架构

许多系统通过采用主从数据库架构的配置来提高整个系统的吞吐量，但是主从的配置也给业务的使用带来了一定的复杂性。接入 ShardingSphere，可以利用读写分离功能管理主从数据库，实现透明化的读写分离功能，让用户像使用一个数据库一样使用主从架构的数据库。

8.3.5 相关参考

Java API YAML 配置

8.3.6 核心概念

主库

添加、更新以及删除数据操作所使用的数据库，目前仅支持单主库。

从库

查询数据操作所使用的数据库，可支持多从库。

主从同步

将主库的数据异步的同步到从库的操作。由于主从同步的异步性，从库与主库的数据会短时间内不一致。

负载均衡策略

通过负载均衡策略将查询请求疏导至不同从库。

8.3.7 使用限制

- 不处理主库和从库的数据同步
- 不处理主库和从库的数据同步延迟导致的数据不一致
- 不支持主库多写
- 不处理主从库间的事务一致性。主从模型中，事务中的数据读写均用主库。

8.4 数据库网关

8.4.1 背景

随着数据库碎片化趋势的不可逆转，多种类型数据库的共存已渐成常态。使用一种 SQL 方言访问异构数据库的场景在不断增加。

8.4.2 挑战

多样化的数据库的存在，使访问数据库的 SQL 方言难于标准化，工程师需要针对不同种类的数据库使用不同的方言，缺乏统一化的查询平台。

将不同类型的数据库方言自动翻译为后端数据库所使用的方言，让工程师可以使用任意一种数据库方言访问所有的后端异构数据库，可以极大的降低开发和维护成本。

8.4.3 目标

SQL 方言的自动翻译，是 Apache ShardingSphere 数据库网关希望达成的主要目标。

8.4.4 应用场景

随着业务场景的多元化，企业内部的数据库产品也呈现多元化的趋势，业务应用与不同数据库产品的对接也变得异常复杂，ShardingSphere 数据库网关可以屏蔽业务应用与底层多元化数据库之间连接，同时为不同的业务场景提供统一的访问协议和语法体系，能够帮助企业快速打造统一的数据访问平台。

8.4.5 核心概念

SQL 方言

SQL 方言也就是数据库方言，指的是某些数据库产品除了支持 SQL 之外，还会有一些自己独有的语法，这就称之为方言，不同的数据库产品，也可能会有不同的 SQL 方言。

8.4.6 使用限制

Apache ShardingSphere 的 SQL 方言翻译处于实验阶段。

目前仅支持 MySQL/PostgreSQL 的方言自动翻译,工程师可以使用 MySQL 的方言和协议,访问 PostgreSQL 数据库，反之亦然。

8.5 流量治理

8.5.1 背景

随着数据规模的不断膨胀，使用多节点集群的分布式方式逐渐成为趋势。对集群整体视角的统一管理能力，和针对单独组件细粒度的控制能力，是基于存算分离的现代数据库体系中不可或缺的功能。

8.5.2 挑战

管控的挑战，在于对集群的集中化管理的统一管理能力以及在单点出现故障时精细化的操作能力。

集中化管理的挑战体现在将包括数据库存储节点和中间件计算节点的状态统一管理，并且能够实时的探测到分布式环境下最新的变动情况，进一步为集群的控制和调度提供依据。

面对超负荷的流量下，针对某一节点进行熔断和限流，以保证整个数据库集群得以继续运行，是分布式系统下对单一节点控制能力的挑战。

8.5.3 目标

实现从数据库到计算节点打通的一体化管理能力，在故障中为组件提供细粒度的控制能力，并尽可能的提供自愈的可能，是 Apache ShardingSphere 管控模块的主要设计目标。

8.5.4 应用场景

计算节点过载保护

当 ShardingSphere 集群内某个计算节点超过负载后，通过熔断功能，阻断应用到该计算节点的流量，保证整个集群继续提供稳定服务。

存储节点限流

在读写分离的场景下，当 ShardingSphere 集群内某个负责读流量的存储节点承接超负荷的请求时，通过限流功能，阻断集群内计算节点到该存储节点的流量，以保证存储节点集群正常响应。

8.5.5 核心概念

熔断

阻断 Apache ShardingSphere 和数据库的连接。当某个 Apache ShardingSphere 节点超过负载后，停止该节点对数据库的访问，使数据库能够保证足够的资源为其他节点提供服务。

限流

面对超负荷的请求开启限流，以保护部分请求可以得以高质量的响应。

8.6 数据迁移

8.6.1 背景

当业务持续发展，数据量和并发量达到一定程度，传统单体数据库可能面临性能、可扩展性、可用性等问题；

业界曾提出 NoSQL 解决方案，通过数据分片和水平扩容解决以上问题，但是 NoSQL 数据库通常不支持事务和 SQL；

ShardingSphere 也支持数据分片和水平扩容，可以解决以上问题，同时还支持分布式事务和 SQL；

ShardingSphere 提供的数据迁移方案可以助力传统单体数据库平滑切换到 ShardingSphere。

8.6.2 挑战

在迁移过程中，不应该对正在运行的业务造成影响。尽可能减少迁移时数据不可用的时间窗口，是数据迁移的第一个挑战；

其次，数据迁移不应该对现有的数据造成影响，如何保证数据的正确性，是数据迁移的第二个挑战。

8.6.3 目标

减少数据迁移时的业务影响，提供一站式的通用数据迁移解决方案，是 Apache ShardingSphere 数据迁移的主要设计目标。

8.6.4 应用场景

假如一个应用系统在使用传统单体数据库，单表数据量达到了 1 亿并且还在快速增长，单体数据库负载持续在高位，成为系统瓶颈。一旦数据库成为瓶颈，对应用服务器扩容是无效的，需要对数据库进行扩容。

8.6.5 相关参考

- 数据迁移的配置
- 数据迁移的实现原理

8.6.6 核心概念

节点

运行计算层或存储层组件进程的实例，可以是物理机、虚拟机、容器等。

集群

为了提供特定服务而集合在一起的多个节点。

源端

原始数据所在的存储集群。

目标端

原始数据将要迁移的目标存储集群。

数据迁移作业

把数据从某一个存储集群复制到另一个存储集群的完整流程。

存量数据

在数据迁移作业开始前，数据节点中已有的数据。

增量数据

在数据迁移作业执行过程中，业务系统所产生的新数据。

8.6.7 使用限制

支持项

- 将外围数据迁移至 Apache ShardingSphere 所管理的数据库；
- 目标端 proxy 不配置规则或配置任意规则；
- 迁移单字段主键或唯一键的表，首字段类型：整型、字符型、部分二进制类型（比如 MySQL VARBINARY）；
- 迁移复合主键或复合唯一键的表。

不支持项

- 不支持在当前存储节点之上做迁移，需要准备一个全新的数据库集群作为迁移目标库；
- 不支持目标端 proxy 使用 HINT 分片策略；
- 不支持目标端表结构和源端不一致；
- 不支持迁移过程中源端表结构变更。

8.7 数据加密

8.7.1 背景

安全控制一直是治理的重要环节，数据加密属于安全控制的范畴。无论对互联网公司还是传统行业来说，数据安全一直是极为重视和敏感的话题。数据加密是指对某些敏感信息通过加密规则进行数据的变形，实

现敏感隐私数据的可靠保护。涉及客户安全数据或者一些商业性敏感数据，如身份证号、手机号、卡号、客户号等个人信息按照相关部门规定，都需要进行数据加密。

对于数据加密的需求，在现实的业务场景中存在如下情况：

- 安全部门规定需将涉及用户敏感信息，例如银行、手机号码等进行加密后存储到数据库，在使用的时候再进行解密处理。

8.7.2 挑战

在真实业务场景中，相关业务开发团队则往往需要针对公司安全部门需求，自行实行并维护一套加解密系统。而当加密场景发生改变时，自行维护的加密系统往往又面临着重构或修改风险。此外，对于已经上线的业务，在不修改业务逻辑和 SQL 的情况下，透明化、安全低风险地实现无缝进行加密改造也相对复杂。

8.7.3 目标

根据业界对加密的需求及业务改造痛点，提供了一套完整、安全、透明化、低改造成本的数据加密整合解决方案，是 Apache ShardingSphere 数据加密模块的主要设计目标。

8.7.4 应用场景

对于想要快速上线的业务，同时又需要完成安全部门的加密规定的场景，接入 ShardingSphere encrypt 功能，可以快速完成数据的合规化加密，客户无需自行开发复杂的加密系统，同时 ShardingSphere encrypt 的灵活性，也能够帮助客户避免加密场景变更带来的复杂重构和修改风险。

8.7.5 相关参考

- 配置：数据加密
- 开发者指南：数据加密

8.7.6 核心概念

逻辑列

用于计算加解密列的逻辑名称，是 SQL 中列的逻辑标识。逻辑列包含密文列（必须）、查询辅助列（可选）、模糊查询辅助列（可选）。

密文列

加密后的数据列。

查询辅助列

用于查询的辅助列。对于一些安全级别更高的非幂等加密算法，提供不可逆的幂等列用于查询。

模糊查询列

用于模糊查询的列。

8.7.7 使用限制

- 需自行处理数据库中原始的存量数据；
- 模糊查询支持%、_，暂不支持 escape；
- 加密字段无法支持查询不区分大小写功能；
- 加密字段无法支持比较操作，如：大于、小于、ORDER BY、BETWEEN 等；
- 加密字段无法支持计算操作，如：AVG、SUM 以及计算表达式；
- 当投影子查询中包含加密字段时，必须使用别名。

8.7.8 附录

不支持的 SQL：

- 加密字段无法支持查询不区分大小写功能；
- 加密字段无法支持比较操作，如：大于、小于、ORDER BY、BETWEEN 等；
- 加密字段无法支持计算操作，如：AVG、SUM 以及计算表达式；
- 不支持子查询中包含加密字段，并且外层投影使用星号的 SQL；
- 不支持 WITH 语句中包含加密列；
- 不支持 INSERT SELECT 语句中包含加密列；
- 不支持 UNION、INTERSECT、EXCEPT 等集合运算语句中包含加密列。

其他：

- 加密规则中配置的加密列、辅助查询列、LIKE 查询列等需要和数据库中的列保持大小写一致。

8.8 数据脱敏

8.8.1 背景

随着《网络安全法》的颁布施行，对个人隐私数据的保护已经上升到法律层面。传统的应用系统普遍缺少对个人隐私数据的保护措施。数据脱敏，可实现在不需要对生产数据库中的数据进行任何改变的情况下，依据用户定义的脱敏规则，对生产数据库返回的数据进行专门的加密、遮盖和替换，确保生产环境的敏感数据能够得到保护。

8.8.2 挑战

在真实业务场景中，相关业务开发团队则往往需要根据脱敏需求，自行实现并维护一套脱敏功能，而脱敏功能往往是耦合在各个业务逻辑中，不同的业务系统难以复用，而当脱敏场景发生改变时，自行维护的脱敏功能往往又面临着重构或修改的风险。

8.8.3 目标

根据业界对脱敏的需求及业务改造痛点，提供了一套完整、安全、透明化、低改造成本的数据脱敏整合解决方案，是 Apache ShardingSphere 数据脱敏模块的主要设计目标。

8.8.4 应用场景

不论是快速上线新业务，还是已经上线的成熟业务，都可以接入 ShardingSphere 脱敏功能，快速地完成脱敏规则的配置。客户无需开发耦合于业务系统的脱敏功能，不需要改动任何业务逻辑和 SQL 就能够透明化地使用脱敏功能。

8.8.5 相关参考

- 配置：数据脱敏
- 开发者指南：数据脱敏

8.8.6 核心概念

逻辑列

用于计算脱敏列的逻辑名称，它是 SQL 中列的逻辑标识。

8.8.7 使用限制

- 脱敏列只支持字符串类型，不支持其他非字符串类型。

8.9 影子库

8.9.1 背景

在基于微服务的分布式应用架构下，业务需要多个服务是通过一系列的服务、中间件的调用来完成，所以单个服务的压力测试已无法代表真实场景。在测试环境中，如果重新搭建一整套与生产环境类似的压测环境，成本过高，并且往往无法模拟线上环境的复杂度以及流量。因此，业内通常选择全链路压测的方式，即在生产环境进行压测，这样所获得的测试结果能够准确地反应系统真实容量和性能水平。

8.9.2 挑战

全链路压测是一项复杂而庞大的工作。需要各个微服务、中间件之间配合与调整，以应对不同流量以及压测标识的透传。通常会搭建一整套压测平台以适用不同测试计划。在数据库层面需要做好数据隔离，为了保证生产数据的可靠性与完整性，需要将压测产生的数据路由到压测环境数据库，防止压测数据对生产数据库中真实数据造成污染。这就要求业务应用在执行 SQL 前，能够根据透传的压测标识，做好数据分类，将相应的 SQL 路由到与之对应的数据源。

8.9.3 目标

Apache ShardingSphere 关注于全链路压测场景下，数据库层面的解决方案。将压测数据自动路由至用户指定的数据库，是 Apache ShardingSphere 影子库模块的主要设计目标。

8.9.4 应用场景

在基于微服务的分布式应用架构下，为了提升系统压力测试的准确性，降低测试成本。通常选择在生产环境进行压力测试。测试中风险也会大大提高。通过 ShardingSphere 影子库功能，结合影子算法灵活的配置。可以解决数据污染，数据库性能等问题，满足复杂业务场景的在线压力测试需求。

8.9.5 相关参考

- Java API：影子库
- YAML 配置：影子库

8.9.6 核心概念

生产库

生产环境使用的数据库。

影子库

压测数据隔离的影子数据库，与生产数据库应当使用相同的配置。

影子算法

影子算法和业务实现紧密相关，目前提供 2 种类型影子算法。

- 基于列的影子算法通过识别 SQL 中的数据，匹配路由至影子库的场景。适用于由压测数据名单驱动的压测场景。
- 基于 Hint 的影子算法通过识别 SQL 中的注释，匹配路由至影子库的场景。适用于由上游系统透传标识驱动的压测场景。

8.9.7 使用限制

基于 Hint 的影子算法

- 无

基于列的影子算法

SQL 不支持列表：* 不支持 DDL * 不支持范围、分组和子查询，如：BETWEEN、GROUP BY …HAVING 等

SQL 支持列表：- INSERT

SQL	是否支持
INSERT INTO table (column,…) VALUES (value,…)	支持
INSERT INTO table (column,…) VALUES (value,…),(value,…),…	支持
INSERT INTO table (column,…) SELECT column1 from table1 where column1 = value1	不支持

- SELECT/UPDATE/DELETE

8.10 可观察性

8.10.1 背景

如何观测集群的运行状态，使运维人员可以快速掌握当前系统现状，并进行进一步的维护工作，是分布式系统的全新挑战。登录到具体服务器的点对点运维方式，无法适用于面向大量分布式服务器的场景。通过对系统可观察性数据的遥测是分布式系统推荐的运维方式。Tracing（链路跟踪）、Metrics（指标监控）和 Logging （日志）是系统运行状况的可观察性数据重要的获取手段。

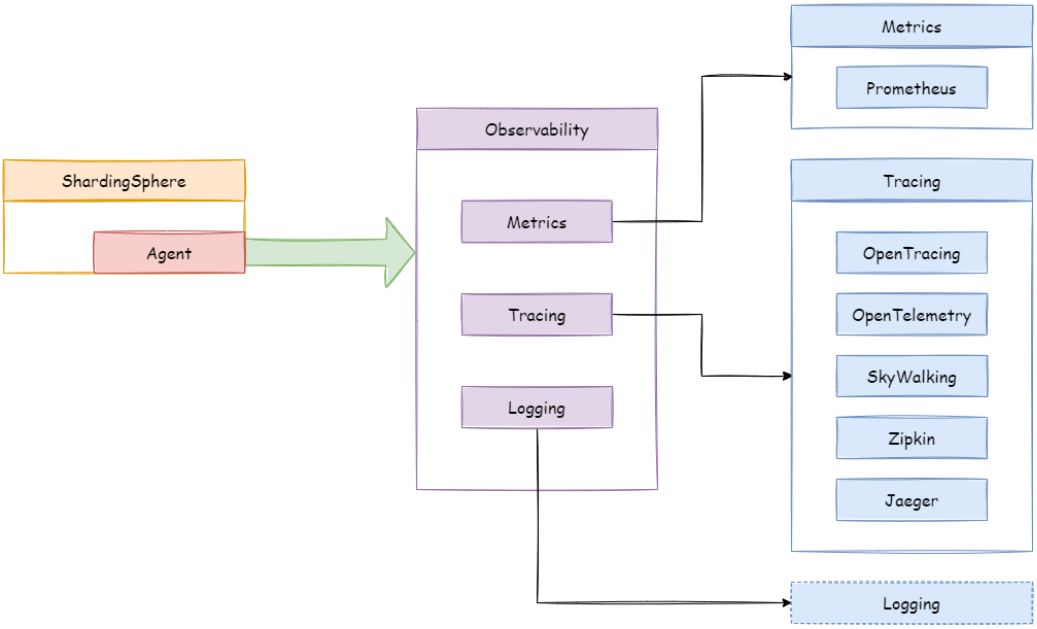
APM（应用性能监控）是通过对系统可观察性数据进行采集、存储和分析，进行系统的性能监控与诊断，主要功能包括性能指标监控、调用链分析，应用拓扑图等。

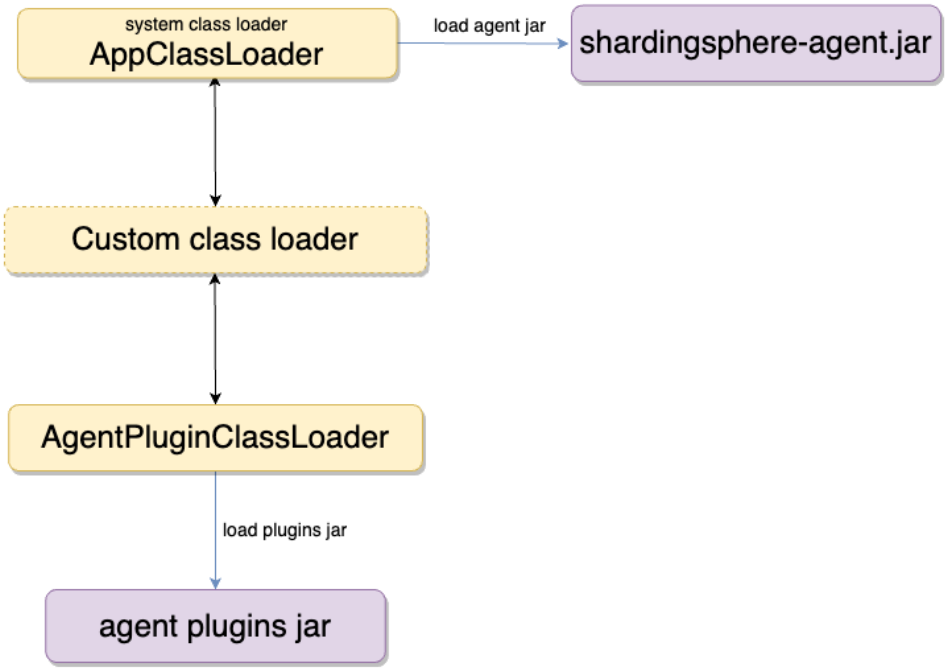
Apache ShardingSphere 并不负责如何采集、存储以及展示应用性能监控的相关数据，而是为应用监控系统提供必要的指标数据。换句话说，Apache ShardingSphere 仅负责产生具有价值的数据，并通过标准协议或插件化的方式递交给相关系统。

Tracing 用于获取 SQL 解析与 SQL 执行的链路跟踪信息。Apache ShardingSphere 默认提供了对 OpenTelemetry，SkyWalking 的支持，也支持用户通过插件化的方式开发自定义的 Tracing 组件。

- 使用 OpenTelemetry OpenTelemetry 在 2019 年由 OpenTracing 和 OpenCencus 合并而来。使用这种方式，只需要在 agent 配置文件中，根据 OpenTelemetry SDK 自动配置说明 ，填写合适的配置即可。可以导出数据到 Jaeger，Zipkin。
- 使用 SkyWalking 需要在 agent 配置中配置启用对应插件，并且需要同时配置使用 SkyWalking 的 apm-toolkit 工具。
- 使用 SkyWalking 的内置自动探针 Apache ShardingSphere 团队与 Apache SkyWalking 团队共同合作，在 SkyWalking 中实现了 Apache ShardingSphere 自动探针，可以将相关的应用性能数据自动发送到 SkyWalking 中。注意这种方式的自动探针不能与 Apache ShardingSphere 插件探针同时使用。

Metrics 则用于收集和展示整个集群的统计指标。Apache ShardingSphere 默认提供了对 Prometheus 的支持。


ShardingSphere
Agent
Observability
Metrics
Tracing
Logging
Metrics
Prometheus
Tracing
OpenTracing
OpenTelemetry
SkyWalking
Zipkin
Jaeger
Logging


system class loader
AppClassLoader
load agent jar
shardingsphere-agent.jar
Custom class loader
AgentPluginClassLoader
load plugins jar
agent plugins jar

8.10.2 挑战

Tracing 和 Metrics 需要通过埋点来收集系统信息。大量的埋点使项目核心代码支离破碎，难于维护，且不易定制化统计指标。

8.10.3 目标

提供尽量多的性能和统计指标，并隔离核心代码和埋点代码，是 Apache ShardingSphere 可观察性模块的设计目标。

8.10.4 应用场景

ShardingSphere 通过 Agent 模块为应用提供可观察性的能力，可适用于以下场景：

监控仪表盘

将系统静态信息（如应用版本）和动态信息（如线程数、SQL 处理信息）等 Metrics 指标，使用标准接口方式暴露给第三方应用（如 Prometheus），管理员能够通过可视化的方式监控系统实时状态。

应用性能监控

在 ShardingSphere 中，一条 SQL 语句要经历解析、路由、改写、执行、结果归并等流程才能最终执行完成，并输出响应。如果 SQL 语句复杂，整体执行耗时过长，如何知道哪一步存在优化空间呢?

通过 Agent + Tracing，管理员可以了解 SQL 执行过程中每一步的耗时情况，轻松定位性能风险，从而能够有针对性的制定 SQL 优化方案。

应用链路追踪

在分布式应用 + 数据分片的场景下，SQL 语句是哪个节点发出的，最终在哪些数据源执行？这是一个非常棘手的问题。如果 SQL 执行过程中发生异常，如何定位发生异常的节点呢?

Agent + Tracing，能够帮助用户解决以上问题。

通过对 SQL 执行过程的完整链路追踪，用户可以得到“SQL 从哪里来，发到哪里去”这样的完整信息，还能够通过生成的拓扑图来直观的观察 SQL 路由情况，运筹帷幄，同时获得快速定位问题根源的能力。

8.10.5 相关参考

- 可观察性的使用
- 开发者指南：可观察性
- 实现原理

8.10.6 核心概念

Agent

基于字节码增强和插件化设计，以提供 Tracing 和 Metrics 埋点，以及日志输出功能。需要开启 Agent 的插件功能后，才能将监控指标数据输出至第三方 APM 中展示。

APM

APM 是应用性能监控的缩写。着眼于分布式系统的性能诊断，其主要功能包括调用链展示，应用拓扑分析等。

Tracing

链路跟踪，通过探针收集调用链数据，并发送到第三方 APM 系统。

Metrics

系统统计指标，通过探针收集，供第三方应用展示。

Logging

日志，通过 Agent 能够方便的扩展日志内容，为分析系统运行状态提供更多信息。

8.11 联邦查询

8.11.1 背景

当用户使用数据分片对海量数据进行水平拆分时，虽然能够有效解决数据库性能瓶颈，但业务上也因此带来了一些新的问题。例如以下场景：跨节点关联查询、子查询、分页、排序、聚合查询。在进行业务实现时需要注意查询 SQL 的使用范围，尽量避免跨数据库实例查询，这使得业务层面的功能受到了数据库的限制。

8.11.2 挑战

用户业务查询 SQL 往往是复杂多变的，通过业务 SQL 改造接入 ShardingSphere 成本较大。将业务端原有的查询，转换为分布式查询并在分布式查询场景下进行相应的 SQL 优化，能够在跨数据库实例的情况下完成：关联查询、子查询、分页、排序、聚合查询。在业务实现上能够让研发人员不必再关心 SQL 的使用范围，能够专注于业务功能开发，减少业务层面的功能限制。

8.11.3 目标

实现跨数据库实例查询的分布式 SQL，是 Apache ShardingSphere 联邦查询的主要设计目标。

8.11.4 应用场景

当需要进行跨数据库关联查询，子查询，以及聚合查询。不需要改动 SQL，通过配置开启联邦查询即可完成分布式查询语句的执行。

8.11.5 相关参考

- 联邦查询的配置

8.11.6 使用限制

Apache ShardingSphere 的联邦查询处于实验阶段。

9 用户手册

本章节面向 Apache ShardingSphere 的用户，详细阐述项目的使用说明。

9.1 ShardingSphere-JDBC

配置是 ShardingSphere-JDBC 中唯一与应用开发者交互的模块，通过它可以快速清晰的理解 ShardingSphere-JDBC 所提供的功能。

本章节是 ShardingSphere-JDBC 的配置参考手册，需要时可当做字典查阅。

ShardingSphere-JDBC 提供了 2 种配置方式，用于不同的使用场景。通过配置，应用开发者可以灵活的使用数据分片、读写分离、数据加密、影子库等功能，并且能够叠加使用。

混合规则配置与单一规则配置一脉相承，只是从配置单一的规则项到配置多个规则项的异同。

需要注意的是，规则项之间的叠加使用是通过数据源名称和表名称关联的。如果前一个规则是面向数据源聚合的，下一个规则在配置数据源时，则需要使用前一个规则配置的聚合后的逻辑数据源名称；同理，如果前一个规则是面向表聚合的，下一个规则在配置表时，则需要使用前一个规则配置的聚合后的逻辑表名称。

更多使用细节请参见使用示例。

9.1.1 YAML 配置

简介

YAML 提供通过配置文件的方式与 ShardingSphere-JDBC 交互。配合治理模块一同使用时，持久化在配置中心的配置均为 YAML 格式。

说明：YAML 配置文件支持配置内容超过 3MB。

YAML 配置是最常见的配置方式，可以省略编程的复杂度，简化用户配置。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

配置 YAML

ShardingSphere-JDBC 的 YAML 配置文件通过 Database 名称、运行模式、数据源集合、规则集合以及属性配置组成。

```
# JDBC 逻辑库名称。在集群模式中，使用该参数来联通 ShardingSphere-JDBC 与 ShardingSphere-Proxy。
# 默认值: logic_db
databaseName (?):

mode:

dataSources:

rules:
- !FOO_XXX
    ...
- !BAR_XXX
    ...

props:
  key_1: value_1
  key_2: value_2
```

模式详情请参见模式配置。

数据源详情请参见数据源配置。

规则详情请参见规则配置。

构建数据源

通过 YamlShardingSphereDataSourceFactory 工厂创建的 ShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。

```
File yamlFile = // 指定 YAML 文件路径
DataSource dataSource = YamlShardingSphereDataSourceFactory.
createDataSource(yamlFile);
```

使用数据源

使用方式同 Java API。

语法说明

!! 表示实例化该类

! 表示自定义别名

- 表示可以包含一个或多个

[] 表示数组，可以与减号相互替换使用

模式配置

参数解释

```
mode (?): # 不配置则默认单机模式
  type: # 运行模式类型。可选配置：Standalone、Cluster
  repository (?): # 持久化仓库配置
```

单机模式

```
mode:
  type: Standalone
  repository:
    type: # 持久化仓库类型
    props: # 持久化仓库所需属性
      foo_key: foo_value
      bar_key: bar_value
```

集群模式 (推荐)

```
mode:
  type: Cluster
  repository:
    type: # 持久化仓库类型
    props: # 持久化仓库所需属性
      namespace: # 注册中心命名空间
      server-lists: # 注册中心连接地址
      foo_key: foo_value
      bar_key: bar_value
```

注意事项

1. 生产环境建议使用集群模式部署。
2. 集群模式部署推荐使用 ZooKeeper 注册中心。
3. ZooKeeper 存在配置信息时，则以 ZooKeeper 中的配置为准。

配置示例

单机模式

```
mode:
  type: Standalone
  repository:
    type: JDBC
```

集群模式 (推荐)

```
mode:
  type: Cluster
  repository:
    type: ZooKeeper
    props:
      namespace: governance
      server-lists: localhost:2181
      retryIntervalMilliseconds: 500
      timeToLiveSeconds: 60
```

使用持久化仓库需要额外引入对应的 Maven 依赖，推荐使用：

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
```

```
    <artifactId>shardingsphere-cluster-mode-repository-zookeeper</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

相关参考

- ZooKeeper 注册中心安装与使用
- 持久化仓库类型的详情，请参见内置持久化仓库类型列表。
- 持久化仓库的可选实现，请参见ShardingSphere-JDBC 可选插件。

数据源配置

背景信息

ShardingSphere-JDBC 支持所有的数据库 JDBC 驱动和连接池。

示例的数据库驱动为 MySQL，连接池为 HikariCP，可以更换为其他数据库驱动和连接池。当使用 ShardingSphere-JDBC 时，JDBC 池的属性名取决于各自 JDBC 池自己的定义，并不由 ShardingSphere 定义，相关的处理可以参考类 `org.apache.shardingsphere.infra.datasource.pool.creator.DataSourcePoolCreator`。例如对于 Alibaba Druid 1.2.9 而言，使用 `url` 代替如下示例中的 `jdbcUrl` 是预期行为。

参数解释

```
dataSources: # 数据源配置，可配置多个 <data-source-name>
  <data_source_name>: # 数据源名称
    dataSourceClassName: # 数据源完整类名
    driverClassName: # 数据库驱动类名，以数据库连接池自身配置为准
    jdbcUrl: # 数据库 URL 连接，以数据库连接池自身配置为准
    username: # 数据库用户名，以数据库连接池自身配置为准
    password: # 数据库密码，以数据库连接池自身配置为准
    # ... 数据库连接池的其它属性
```

配置示例

```
dataSources:
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/ds_1
    username: root
    password:
  ds_2:
```

```
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/ds_2
    username: root
    password:

  # 配置其他数据源
```

规则配置

规则是 Apache ShardingSphere 面向可插拔的一部分。本章节是 ShardingSphere-JDBC 的 YAML 规则配置参考手册。

数据分片

背景信息

数据分片 YAML 配置方式具有非凡的可读性，通过 YAML 格式，能够快速地理解分片规则之间的依赖关系，ShardingSphere 会根据 YAML 配置，自动完成 ShardingSphereDataSource 对象的创建，减少用户不必要的编码工作。

参数解释

```
rules:
- !SHARDING
  tables: # 数据分片规则配置
    <logic_table_name> (+): # 逻辑表名称
      actualDataNodes (?): # 由数据源名 + 表名组成（参考 Inline 语法规则）
      databaseStrategy (?): # 分库策略，缺省表示使用默认分库策略，以下的分片策略只能选其一
        standard: # 用于单分片键的标准分片场景
          shardingColumn: # 分片列名称
          shardingAlgorithmName: # 分片算法名称
        complex: # 用于多分片键的复合分片场景
          shardingColumns: # 分片列名称，多个列以逗号分隔
          shardingAlgorithmName: # 分片算法名称
        hint: # Hint 分片策略
          shardingAlgorithmName: # 分片算法名称
        none: # 不分片
      tableStrategy: # 分表策略，同分库策略
      keyGenerateStrategy: # 分布式序列策略
        column: # 自增列名称，缺省表示不使用自增主键生成器
        keyGeneratorName: # 分布式序列算法名称
      auditStrategy: # 分片审计策略
        auditorNames: # 分片审计算法名称
          - <auditor_name>
```

```
          - <auditor_name>
        allowHintDisable: true # 是否禁用分片审计 hint
  autoTables: # 自动分片表规则配置
    t_order_auto: # 逻辑表名称
      actualDataSources (?): # 数据源名称
      shardingStrategy: # 切分策略
        standard: # 用于单分片键的标准分片场景
          shardingColumn: # 分片列名称
          shardingAlgorithmName: # 自动分片算法名称
  bindingTables (+): # 绑定表规则列表
    - <logic_table_name_1, logic_table_name_2, ...>
    - <logic_table_name_1, logic_table_name_2, ...>
  defaultDatabaseStrategy: # 默认数据库分片策略
  defaultTableStrategy: # 默认表分片策略
  defaultKeyGenerateStrategy: # 默认的分布式序列策略
  defaultShardingColumn: # 默认分片列名称

  # 分片算法配置
  shardingAlgorithms:
    <sharding_algorithm_name> (+): # 分片算法名称
      type: # 分片算法类型
      props: # 分片算法属性配置
      # ...

  # 分布式序列算法配置
  keyGenerators:
    <key_generate_algorithm_name> (+): # 分布式序列算法名称
      type: # 分布式序列算法类型
      props: # 分布式序列算法属性配置
      # ...
  # 分片审计算法配置
  auditors:
    <sharding_audit_algorithm_name> (+): # 分片审计算法名称
      type: # 分片审计算法类型
      props: # 分片审计算法属性配置
      # ...

- !BROADCAST
  tables: # 广播表规则列表
    - <table_name>
    - <table_name>
```

操作步骤

1. 在 YAML 文件中配置数据分片规则，包含数据源、分片规则、全局属性等配置项；
2. 调用 YamlShardingSphereDataSourceFactory 对象的 createDataSource 方法，根据 YAML 文件中的配置信息创建 ShardingSphereDataSource。

配置示例

数据分片 YAML 配置示例如下：

```
dataSources:
  ds_0:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/demo_ds_0?serverTimezone=UTC&useSSL=false&
useUnicode=true&characterEncoding=UTF-8
    username: root
    password:
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/demo_ds_1?serverTimezone=UTC&useSSL=false&
useUnicode=true&characterEncoding=UTF-8
    username: root
    password:

rules:
- !SHARDING
  tables:
    t_order:
      actualDataNodes: ds_${0..1}.t_order_${0..1}
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: t_order_inline
      keyGenerateStrategy:
        column: order_id
        keyGeneratorName: snowflake
      auditStrategy:
        auditorNames:
          - sharding_key_required_auditor
        allowHintDisable: true
    t_order_item:
      actualDataNodes: ds_${0..1}.t_order_item_${0..1}
      tableStrategy:
        standard:
          shardingColumn: order_id
```

```
          shardingAlgorithmName: t_order_item_inline
      keyGenerateStrategy:
        column: order_item_id
        keyGeneratorName: snowflake
    t_account:
      actualDataNodes: ds_${0..1}.t_account_${0..1}
      tableStrategy:
        standard:
          shardingAlgorithmName: t_account_inline
      keyGenerateStrategy:
        column: account_id
        keyGeneratorName: snowflake
  defaultShardingColumn: account_id
  bindingTables:
    - t_order,t_order_item
  defaultDatabaseStrategy:
    standard:
      shardingColumn: user_id
      shardingAlgorithmName: database_inline
  defaultTableStrategy:
    none:

  shardingAlgorithms:
    database_inline:
      type: INLINE
      props:
        algorithm-expression: ds_${user_id % 2}
    t_order_inline:
      type: INLINE
      props:
        algorithm-expression: t_order_${order_id % 2}
    t_order_item_inline:
      type: INLINE
      props:
        algorithm-expression: t_order_item_${order_id % 2}
    t_account_inline:
      type: INLINE
      props:
        algorithm-expression: t_account_${account_id % 2}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
  auditors:
    sharding_key_required_auditor:
      type: DML_SHARDING_CONDITIONS

- !BROADCAST
  tables:
```

```
    - t_address

props:
  sql-show: false
```

通过 YamlShardingSphereDataSourceFactory 的 createDataSource 方法，读取 YAML 配置完成数据源的创建。

```
YamlShardingSphereDataSourceFactory.createDataSource(getFile("/META-INF/sharding-databases-tables.yaml"));
```

相关参考

- 核心特性：数据分片
- 开发者指南：数据分片

广播表

广播表 YAML 配置方式具有非凡的可读性,通过 YAML 格式,能够快速地理解广播表配置,ShardingSphere 会根据 YAML 配置，自动完成 ShardingSphereDataSource 对象的创建，减少用户不必要的编码工作。

参数解释

```
rules:
- !BROADCAST
  tables: # 广播表规则列表
    - <table_name>
    - <table_name>
```

操作步骤

1. 在 YAML 文件中配置广播表列表
2. 调用 YamlShardingSphereDataSourceFactory 对象的 createDataSource 方法，根据 YAML 文件中的配置信息创建 ShardingSphereDataSource。

配置示例

广播表 YAML 配置示例如下:

```
dataSources:
  ds_0:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/demo_ds_0?serverTimezone=UTC&useSSL=false&
useUnicode=true&characterEncoding=UTF-8
    username: root
    password:
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/demo_ds_1?serverTimezone=UTC&useSSL=false&
useUnicode=true&characterEncoding=UTF-8
    username: root
    password:

rules:
- !BROADCAST
  tables:
    - t_address
```

通过 YamlShardingSphereDataSourceFactory 的 createDataSource 方法，读取 YAML 配置完成数据源的创建。

```
YamlShardingSphereDataSourceFactory.createDataSource(getFile("/META-INF/broadcast-
databases-tables.yaml"));
```

读写分离

背景信息

读写分离 YAML 配置方式可读性高，通过 YAML 格式，能够快速地理解读写分片规则之间的依赖关系，ShardingSphere 会根据 YAML 配置，自动完成 ShardingSphereDataSource 对象的创建，减少用户不必要的编码工作。

参数解释

读写分离

```
rules:
- !READWRITE_SPLITTING
  dataSourceGroups:
    <data_source_name> (+): # 读写分离逻辑数据源名称，默认使用 Groovy 的行表达式 SPI 实现来解析
      write_data_source_name: # 写库数据源名称，默认使用 Groovy 的行表达式 SPI 实现来解析
      read_data_source_names: # 读库数据源名称，多个从数据源用逗号分隔，默认使用 Groovy 的行表达式 SPI 实现来解析
      transactionalReadQueryStrategy (?): # 事务内读请求的路由策略，可选值：PRIMARY（路由至主库）、FIXED（同一事务内路由至固定数据源）、DYNAMIC（同一事务内路由至非固定数据源）。默认值：DYNAMIC
      loadBalancerName: # 负载均衡算法名称

  # 负载均衡算法配置
  loadBalancers:
    <load_balancer_name> (+): # 负载均衡算法名称
      type: # 负载均衡算法类型
      props: # 负载均衡算法属性配置
        # ...
```

算法类型的详情，请参见内置负载均衡算法列表。

操作步骤

1. 添加读写分离数据源
2. 设置负载均衡算法
3. 使用读写分离数据源

配置示例

```
rules:
- !READWRITE_SPLITTING
  dataSourceGroups:
    readwrite_ds:
      writeDataSourceName: write_ds
      readDataSourceNames:
        - read_ds_0
        - read_ds_1
      transactionalReadQueryStrategy: PRIMARY
      loadBalancerName: random
  loadBalancers:
```

```
    random:
      type: RANDOM
```

相关参考

- 核心特性：读写分离
- Java API：读写分离

分布式事务

背景信息

ShardingSphere 提供了三种模式的分布式事务 LOCAL, XA, BASE。

参数解释

```
transaction:
  defaultType: # 事务模式，可选值 LOCAL/XA/BASE
  providerType: # 指定模式下的具体实现
```

操作步骤

使用 LOCAL 模式

global.yaml 配置文件内容如下：

```
transaction:
  defaultType: LOCAL
```

使用 XA 模式

global.yaml 配置文件内容如下：

```
transaction:
  defaultType: XA
  providerType: Narayana/Atomikos
```

手动添加 Narayana 相关依赖：

```
jta-5.12.7.Final.jar
arjuna-5.12.7.Final.jar
common-5.12.7.Final.jar
```

```
jboss-connector-api_1.7_spec-1.0.0.Final.jar
jboss-logging-3.2.1.Final.jar
jboss-transaction-api_1.2_spec-1.0.0.Alpha3.jar
jboss-transaction-spi-7.6.1.Final.jar
narayana-jts-integration-5.12.7.Final.jar
shardingsphere-transaction-xa-narayana-x.x.x-SNAPSHOT.jar
```

使用 BASE 模式

global.yaml 配置文件内容如下：

```
transaction:
  defaultType: BASE
  providerType: Seata
```

搭建 Seata Server，添加相关配置文件，和 Seata 依赖，具体步骤参考 ShardingSphere 集成 Seata 柔性事务

数据加密

背景信息

数据加密 YAML 配置方式具有非凡的可读性，通过 YAML 格式，能够快速地理解加密规则之间的依赖关系，ShardingSphere 会根据 YAML 配置，自动完成 ShardingSphereDataSource 对象的创建，减少用户不必要的编码工作。

参数解释

```
rules:
- !ENCRYPT
  tables:
    <table_name> (+): # 加密表名称
      columns:
        <column_name> (+): # 加密列名称
          cipher:
            name: # 密文列名称
            encryptorName: # 密文列加密算法名称
          assistedQuery (?):
            name: # 查询辅助列名称
            encryptorName:  # 查询辅助列加密算法名称
          likeQuery (?):
            name: # 模糊查询列名称
            encryptorName:  # 模糊查询列加密算法名称

  # 加密算法配置
```

```
  encryptors:
    <encrypt_algorithm_name> (+): # 加解密算法名称
      type: # 加解密算法类型
      props: # 加解密算法属性配置
        # ...
```

算法类型的详情，请参见内置加密算法列表。

操作步骤

1. 在 YAML 文件中配置数据加密规则，包含数据源、加密规则、全局属性等配置项；
2. 调用 YamlShardingSphereDataSourceFactory 对象的 createDataSource 方法，根据 YAML 文件中的配置信息创建 ShardingSphereDataSource。

配置示例

数据加密 YAML 配置如下：

```
dataSources:
  unique_ds:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/demo_ds?serverTimezone=UTC&useSSL=false&
useUnicode=true&characterEncoding=UTF-8
    username: root
    password:

rules:
- !ENCRYPT
  tables:
    t_user:
      columns:
        username:
          cipher:
            name: username
            encryptorName: aes_encryptor
          assistedQuery:
            name: assisted_query_username
            encryptorName: assisted_encryptor
          likeQuery:
            name: like_query_username
            encryptorName: like_encryptor
        pwd:
          cipher:
            name: pwd
            encryptorName: aes_encryptor
```

```
        assistedQuery:
          name: assisted_query_pwd
          encryptorName: assisted_encryptor
  encryptors:
    aes_encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
        digest-algorithm-name: SHA-1
    assisted_encryptor:
      type: MD5
    like_encryptor:
      type: CHAR_DIGEST_LIKE
```

然后通过 YamlShardingSphereDataSourceFactory 的 createDataSource 方法创建数据源。

```
YamlShardingSphereDataSourceFactory.createDataSource(getFile());
```

相关参考

- 核心特性：数据加密
- 开发者指南：数据加密

数据脱敏

背景信息

数据脱敏 YAML 配置方式良好的可读性，通过 YAML 格式，能够快速地理解脱敏规则之间的依赖关系，ShardingSphere 会根据 YAML 配置，自动完成 ShardingSphereDataSource 对象的创建，减少用户不必要的编码工作。

参数解释

```
rules:
- !MASK
  tables:
    <table_name> (+): # 脱敏表名称
      columns:
        <column_name> (+): # 脱敏列名称
          maskAlgorithm: # 脱敏算法

  # 脱敏算法配置
  maskAlgorithms:
    <mask_algorithm_name> (+): # 脱敏算法名称
```

```
      type: # 脱敏算法类型
      props: # 脱敏算法属性配置
      # ...
```

算法类型的详情，请参见内置脱敏算法列表。

操作步骤

1. 在 YAML 文件中配置数据脱敏规则，包含数据源、脱敏规则、全局属性等配置项；
2. 调用 YamlShardingSphereDataSourceFactory 对象的 createDataSource 方法，根据 YAML 文件中的配置信息创建 ShardingSphereDataSource。

配置示例

数据脱敏 YAML 配置如下：

```
dataSources:
  unique_ds:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/demo_ds?serverTimezone=UTC&useSSL=false&
useUnicode=true&characterEncoding=UTF-8
    username: root
    password:

rules:
- !MASK
  tables:
    t_user:
      columns:
        password:
          maskAlgorithm: md5_mask
        email:
          maskAlgorithm: mask_before_special_chars_mask
        telephone:
          maskAlgorithm: keep_first_n_last_m_mask

  maskAlgorithms:
    md5_mask:
      type: MD5
    mask_before_special_chars_mask:
      type: MASK_BEFORE_SPECIAL_CHARS
      props:
        special-chars: '@'
        replace-char: '*'
    keep_first_n_last_m_mask:
```

```
      type: KEEP_FIRST_N_LAST_M
      props:
        first-n: 3
        last-m: 4
        replace-char: '*'
```

然后通过 YamlShardingSphereDataSourceFactory 的 createDataSource 方法创建数据源。

```
YamlShardingSphereDataSourceFactory.createDataSource(getFile());
```

相关参考

- 核心特性：数据脱敏
- 开发者指南：数据脱敏

影子库

背景信息

如果您想在 ShardingSphere-Proxy 中使用 ShardingSphere 影子库功能请参考以下配置。

参数解释

```
rules:
- !SHADOW
  dataSources:
    shadowDataSource:
      productionDataSourceName: # 生产数据源名称
      shadowDataSourceName: # 影子数据源名称
  tables:
    <table_name>:
      dataSourceNames: # 影子表关联影子数据源名称列表
        - <shadow_data_source>
      shadowAlgorithmNames: # 影子表关联影子算法名称列表
        - <shadow_algorithm_name>
  defaultShadowAlgorithmName: # 默认影子算法名称（选配项）
  shadowAlgorithms:
    <shadow_algorithm_name> (+): # 影子算法名称
      type: # 影子算法类型
      props: # 影子算法属性配置
```

详情请参见内置影子算法列表

操作步骤

1. 在 YAML 文件中配置影子库规则，包含数据源、影子库规则、全局属性等配置项；
2. 调用 YamlShardingSphereDataSourceFactory 对象的 createDataSource 方法，根据 YAML 文件中的配置信息创建 ShardingSphereDataSource。

配置示例

```
dataSources:
   ds:
      url: jdbc:mysql://127.0.0.1:3306/ds?serverTimezone=UTC&useSSL=false
      username: root
      password:
      connectionTimeoutMilliseconds: 30000
      idleTimeoutMilliseconds: 60000
      maxLifetimeMilliseconds: 1800000
      maxPoolSize: 50
      minPoolSize: 1
   shadow_ds:
      url: jdbc:mysql://127.0.0.1:3306/shadow_ds?serverTimezone=UTC&useSSL=false
      username: root
      password:
      connectionTimeoutMilliseconds: 30000
      idleTimeoutMilliseconds: 60000
      maxLifetimeMilliseconds: 1800000
      maxPoolSize: 50
      minPoolSize: 1

rules:
- !SHADOW
  dataSources:
    shadowDataSource:
      productionDataSourceName: ds
      shadowDataSourceName: shadow_ds
  tables:
    t_order:
      dataSourceNames:
        - shadowDataSource
      shadowAlgorithmNames:
        - user_id_insert_match_algorithm
        - sql_hint_algorithm
  shadowAlgorithms:
    user_id_insert_match_algorithm:
      type: REGEX_MATCH
      props:
        operation: insert
        column: user_id
```

```
        regex: "[1]"
    sql_hint_algorithm:
      type: SQL_HINT
```

相关参考

- 影子库的核心特性
- JAVA API：影子库配置

SQL 解析

背景信息

SQL 解析 YAML 配置方式具有可读性高，使用简单的特点。通过 YAML 文件的方式，用户可以将代码与配置分离，并且根据需要方便地修改配置文件。

参数解释

```
sqlParser:
  sqlStatementCache: # SQL 语句本地缓存配置项
    initialCapacity: # 本地缓存初始容量
    maximumSize: # 本地缓存最大容量
  parseTreeCache: # 解析树本地缓存配置项
    initialCapacity: # 本地缓存初始容量
    maximumSize: # 本地缓存最大容量
```

操作步骤

1. 设置本地缓存配置
2. 设置解析配置
3. 使用解析引擎解析 SQL

配置示例

```
sqlParser:
  sqlStatementCache:
    initialCapacity: 2000
    maximumSize: 65535
  parseTreeCache:
    initialCapacity: 128
    maximumSize: 1024
```

相关参考

- JAVA API：SQL 解析

SQL 翻译

背景信息

SQL 翻译 YAML 配置方式具有可读性高，使用简单的特点。通过 YAML 文件的方式，用户可以将代码与配置分离，并且根据需要方便地修改配置文件。

参数解释

```
sqlTranslator:
  type: # SQL 翻译器类型
  useOriginalSQLWhenTranslatingFailed: # SQL 翻译失败是否使用原始 SQL 继续执行
```

操作步骤

1. 配置翻译类型 type
2. 配置 useOriginalSQLWhenTranslatingFailed 参数，是否在 SQL 翻译失败后使用原始 SQL 继续执行

配置示例

```
sqlTranslator:
  type: Native
  useOriginalSQLWhenTranslatingFailed: true
```

相关参考

- JAVA API：SQL 翻译

混合规则

背景信息

ShardingSphere 涵盖了很多功能，例如，分库分片、读写分离、数据加密等。这些功能用户可以单独进行使用，也可以配合一起使用，下面是基于 YAML 的参数解释和配置示例。

参数解释

```yaml
rules:
- !SHARDING
  tables:
    <logic_table_name>: # 逻辑表名称:
      actualDataNodes: # 由逻辑数据源名 + 表名组成（参考 Inline 语法规则）
      tableStrategy: # 分表策略，同分库策略
        standard:
          shardingColumn: # 分片列名称
          shardingAlgorithmName: # 分片算法名称
      keyGenerateStrategy:
        column: # 自增列名称，缺省表示不使用自增主键生成器
        keyGeneratorName: # 分布式序列算法名称
  defaultDatabaseStrategy:
    standard:
      shardingColumn: # 分片列名称
      shardingAlgorithmName: # 分片算法名称
  shardingAlgorithms:
    <sharding_algorithm_name>: # 分片算法名称
      type: INLINE
      props:
        algorithm-expression: # INLINE 表达式
    t_order_inline:
      type: INLINE
      props:
        algorithm-expression: # INLINE 表达式
  keyGenerators:
    <key_generate_algorithm_name> (+): # 分布式序列算法名称
      type: # 分布式序列算法类型
      props: # 分布式序列算法属性配置
- !ENCRYPT
  encryptors:
    <encrypt_algorithm_name> (+): # 加解密算法名称
      type: # 加解密算法类型
      props: # 加解密算法属性配置
    <encrypt_algorithm_name> (+): # 加解密算法名称
      type: # 加解密算法类型
  tables:
    <table_name>: # 加密表名称
      columns:
        <column_name> (+): # 加密列名称
          cipher:
            name: # 密文列名称
            encryptorName: # 密文列加密算法名称
          assistedQuery (?):
            name: # 查询辅助列名称
            encryptorName:  # 查询辅助列加密算法名称
```

```
        likeQuery (?):
          name: # 模糊查询列名称
          encryptorName:  # 模糊查询列加密算法名称
```

配置示例

```
rules:
- !SHARDING
  tables:
    t_order:
      actualDataNodes: replica_ds_${0..1}.t_order_${0..1}
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: t_order_inline
      keyGenerateStrategy:
        column: order_id
        keyGeneratorName: snowflake
  defaultDatabaseStrategy:
    standard:
      shardingColumn: user_id
      shardingAlgorithmName: database_inline
  shardingAlgorithms:
    database_inline:
      type: INLINE
      props:
        algorithm-expression: replica_ds_${user_id % 2}
    t_order_inline:
      type: INLINE
      props:
        algorithm-expression: t_order_${order_id % 2}
    t_order_item_inline:
      type: INLINE
      props:
        algorithm-expression: t_order_item_${order_id % 2}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
- !ENCRYPT
  encryptors:
    aes_encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
        digest-algorithm-name: SHA-1
    assisted_encryptor:
      type: MD5
```

```
    like_encryptor:
      type: CHAR_DIGEST_LIKE
  tables:
    t_encrypt:
      columns:
        user_id:
          cipher:
            name: user_cipher
            encryptorName: aes_encryptor
          assistedQuery:
            name: assisted_query_user
            encryptorName: assisted_encryptor
          likeQuery:
            name: like_query_user
            encryptorName: like_encryptor
        order_id:
          cipher:
            name: order_cipher
            encryptorName: aes_encryptor
```

数据分片路由缓存

背景信息

该项功能为**实验性功能**，需要与数据分片功能同时使用。数据分片路由缓存会将逻辑 SQL、分片键实际参数值、路由结果放入缓存中，以空间换时间，减少路由逻辑对 CPU 的使用。

建议仅在满足以下条件的情况下启用：- 纯 OLTP 场景 - ShardingSphere 进程所在机器 CPU 已达到瓶颈 - CPU 开销主要在于 ShardingSphere 路由逻辑 - 所有 SQL 已经最优且每次 SQL 执行都能命中单一分片

在不满足以上条件的情况下使用，可能对 SQL 的执行延时不会有明显改善，同时会增加内存的压力。

参数解释

```
rules:
- !SHARDING
  tables:
  shardingAlgorithms:
  # ...
  shardingCache:
    allowedMaxSqlLength: 512 # 允许缓存的 SQL 长度限制
    routeCache:
      initialCapacity: 65536 # 缓存初始容量
      maximumSize: 262144 # 缓存最大容量
      softValues: true # 是否软引用缓存值
```

相关参考

- 核心特性：数据分片

单表

背景信息

单表规则用于指定哪些单表需要被 ShardingSphere 管理，也可设置默认的单表数据源。

参数解释

```
rules:
- !SINGLE
  tables:
    # MySQL 风格
    - ds_0.t_single # 加载指定单表
    - ds_1.* # 加载指定数据源中的全部单表
    - "*.*" # 加载全部单表
    # PostgreSQL 风格
    - ds_0.public.t_config
    - ds_1.public.*
    - ds_2.*.*
    - "*.*.*"
  defaultDataSource: ds_0 # 默认数据源，仅在执行 CREATE TABLE 创建单表时有效。缺失值为空，表示随机单播路由。
```

相关参考

- 单表

联邦查询

背景信息

该功能为**实验性功能，暂不适合核心系统生产环境使用**。联邦查询 YAML 配置方式可读性高，当关联查询中的多个表分布在不同的数据库实例上时，通过开启联邦查询可以进行跨库关联查询，以及子查询。

参数解释

```
sqlFederation:
  sqlFederationEnabled: # 是否开启联邦查询
  allQueryUseSQLFederation: # 是否全部查询 SQL 使用联邦查询
  executionPlanCache: # 执行计划缓存
    initialCapacity: 2000 # 执行计划缓存初始容量
    maximumSize: 65535 # 执行计划缓存最大容量
```

配置示例

```
sqlFederation:
  sqlFederationEnabled: true
  allQueryUseSQLFederation: false
  executionPlanCache:
    initialCapacity: 2000
    maximumSize: 65535
```

相关参考

- JAVA API：联邦查询

算法配置

分片算法

```
shardingAlgorithms:
  # algorithmName 由用户指定，需要和分片策略中的 shardingAlgorithmName 属性一致
  <algorithmName>:
    # type 和 props，请参考分片内置算法：https://shardingsphere.apache.org/document/
current/cn/user-manual/common-config/builtin-algorithm/sharding/
    type: xxx
    props:
      xxx: xxx
```

加密算法

```
encryptors:
  # encryptorName 由用户指定，需要和加密规则中的 encryptorName 属性一致
  <encryptorName>:
    # type 和 props，请参考加密内置算法：https://shardingsphere.apache.org/document/current/cn/user-manual/common-config/builtin-algorithm/encrypt/
    type: xxx
    props:
      xxx: xxx
```

读写分离负载均衡算法

```
loadBalancers:
  # loadBalancerName 由用户指定，需要和读写分离规则中的 loadBalancerName 属性一致
  <loadBalancerName>:
    # type 和 props，请参考读写分离负载均衡内置算法：https://shardingsphere.apache.org/document/current/cn/user-manual/common-config/builtin-algorithm/load-balance/
    type: xxx
    props:
      xxx: xxx
```

影子算法

```
shadowAlgorithms:
  # shadowAlgorithmName 由用户指定，需要和影子库规则中的 shadowAlgorithmNames 属性一致
  <shadowAlgorithmName>:
    # type 和 props，请参考影子库内置算法：https://shardingsphere.apache.org/document/current/cn/user-manual/common-config/builtin-algorithm/shadow/
    type: xxx
    props:
      xxx: xxx
```

脱敏算法

```
maskAlgorithms:
  # maskAlgorithmName 由用户指定，需要和脱敏规则中的 maskAlgorithm 属性一致
  <maskAlgorithmName>:
    # type 和 props，请参考脱敏内置算法：https://shardingsphere.apache.org/document/current/cn/user-manual/common-config/builtin-algorithm/mask/
    type: xxx
    props:
      xxx: xxx
```

JDBC 驱动

背景信息

ShardingSphere-JDBC 提供了 JDBC 驱动，可以仅通过配置变更即可使用，无需改写代码。

参数解释

驱动类名称

`org.apache.shardingsphere.driver.ShardingSphereDriver`

URL 配置及配置示例

参考 已知实现 。

操作步骤

1. 引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

2. 使用驱动

- 使用原生驱动:

```
Class.forName("org.apache.shardingsphere.driver.ShardingSphereDriver");
String jdbcUrl = "jdbc:shardingsphere:classpath:config.yaml";

String sql = "SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_
id WHERE o.user_id=? AND o.order_id=?";
try (
        Connection conn = DriverManager.getConnection(jdbcUrl);
        PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, 10);
    ps.setInt(2, 1000);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while(rs.next()) {
            // ...
        }
    }
}
```

- 使用数据库连接池

```
String driverClassName = "org.apache.shardingsphere.driver.ShardingSphereDriver";
String jdbcUrl = "jdbc:shardingsphere:classpath:config.yaml";

// 以 HikariCP 为例
HikariDataSource dataSource = new HikariDataSource();
dataSource.setDriverClassName(driverClassName);
dataSource.setJdbcUrl(jdbcUrl);

String sql = "SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_
id WHERE o.user_id=? AND o.order_id=?";
try (
        Connection conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, 10);
    ps.setInt(2, 1000);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while(rs.next()) {
            // ...
        }
    }
}
```

已知实现

背景信息

对于 `org.apache.shardingsphere.driver.ShardingSphereDriver` 的驱动类, 通过实现 `org.apache.shardingsphere.infra.url.ShardingSphereURLLoader` 的 SPI，可允许从多种来源和 File System 获取并解析为 ShardingSphere 的 YAML 配置文件。如无特定声明，以下实现均采用 YAML 1.1 作为 YAML 的编写规范，这并不阻止 `org.apache.shardingsphere.infra.url.ShardingSphereURLLoader` 的自定义实现从 XML 或 JSON 等文件手动转化为 YAML。

在解析并加载 YAML 文件为 ShardingSphere 的元数据后，会再次通过模式配置的相关配置决定下一步行为。讨论两种情况，

1. 元数据持久化仓库中不存在 ShardingSphere 的元数据，本地元数据将被存储到元数据持久化仓库。
2. 元数据持久化仓库中已存在 ShardingSphere 的元数据，无论是否与本地元数据相同，本地元数据将被元数据持久化仓库的元数据覆盖。

对元数据持久化仓库的配置需参考元数据持久化仓库。

加载配置文件的方式

从类路径中加载配置文件

配置文件为 xxx.yaml，当 placeholder-type 为 none 或不标明时，配置文件格式与 YAML 配置一致。当 placeholder-type 存在且不为 none 时，配置文件格式的定义参考本文的 JDBC URL 参数一节。

用例：

- jdbc:shardingsphere:classpath:config.yaml
- jdbc:shardingsphere:classpath:config.yaml?placeholder-type=none
- jdbc:shardingsphere:classpath:config.yaml?placeholder-type=environment
- jdbc:shardingsphere:classpath:config.yaml?placeholder-type=system_props

从绝对路径中加载配置文件

配置文件为 xxx.yaml，当 placeholder-type 为 none 或不标明时，配置文件格式与 YAML 配置一致。当 placeholder-type 存在且不为 none 时，配置文件格式的定义参考本文的 JDBC URL 参数一节。

用例：

- jdbc:shardingsphere:absolutepath:/path/to/config.yaml
- jdbc:shardingsphere:absolutepath:/path/to/config.yaml?placeholder-type=none
- jdbc:shardingsphere:absolutepath:/path/to/config.yaml?placeholder-type=environment
- jdbc:shardingsphere:absolutepath:/path/to/config.yaml?placeholder-type=system_props

JDBC URL 参数

对于 org.apache.shardingsphere.infra.url.ShardingSphereURLLoader 的实现，并非所有的 JDBC URL 参数都必须被解析，这涉及到如何实现 org.apache.shardingsphere.infra.url.ShardingSphereURLLoader.load()。

placeholder-type

存在 `placeholder-type` 属性用于可选的加载包含动态占位符的配置文件，`placeholder-type` 存在默认值为 `none`。当 `placeholder-type` 设置为非 `none` 时，在涉及的 YAML 文件中允许通过动态占位符设置特定 YAML 属性的值，并配置可选的默认值。动态占位符的名称和其可选的默认值通过`::` 分割，在最外层通过 `$${和}` 包裹。

讨论两种情况，

1. 当对应的动态占位符的值不存在时，此 YAML 属性的值将被设置为`::` 右侧的默认值。
2. 当对应的动态占位符的值和`::` 右侧的默认值均不存在时，此属性将被设置为空。

单个动态占位符

none

配置文件为 `xxx.yaml`，配置文件格式与 YAML 配置 一致。

用例：

- `jdbc:shardingsphere:classpath:config.yaml`
- `jdbc:shardingsphere:classpath:config.yaml?placeholder-type=none`
- `jdbc:shardingsphere:absolutepath:/path/to/config.yaml`
- `jdbc:shardingsphere:absolutepath:/path/to/config.yaml?placeholder-type=none`

environment

加载包含环境变量的配置文件时，需将 `placeholder-type` 置为 `environment`，这常用于 Docker Image 的部署场景。配置文件为 `xxx.yaml`，配置文件格式与 YAML 配置 基本一致。

假设存在以下一组环境变量，

1. 存在环境变量 `FIXTURE_JDBC_URL` 为 `jdbc:h2:mem:foo_ds_1;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MySQL`。
2. 存在环境变量 `FIXTURE_USERNAME` 为 `sa`。

则对于以下 YAML 文件的截取片段，

```
ds_1:
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  driverClassName: $${FIXTURE_DRIVER_CLASS_NAME::org.h2.Driver}
  jdbcUrl: $${FIXTURE_JDBC_URL::jdbc:h2:mem:foo_ds_do_not_use}
  username: $${FIXTURE_USERNAME::}
  password: $${FIXTURE_PASSWORD::}
```

此 YAML 截取片段将被解析为，

```
ds_1:
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  driverClassName: org.h2.Driver
  jdbcUrl: jdbc:h2:mem:foo_ds_1;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;
MODE=MySQL
  username: sa
  password:
```

用例:

- `jdbc:shardingsphere:classpath:config.yaml?placeholder-type=environment`
- `jdbc:shardingsphere:absolutepath:/path/to/config.yaml?placeholder-type=environment`

system_props

加载包含系统属性的配置文件时，需将 `placeholder-type` 置为 `system_props`。配置文件为 `xxx.yaml`，配置文件格式与 YAML 配置 基本一致。

假设存在以下一组系统属性,

1. 存在系统属性 `fixture.config.driver.jdbc-url` 为 `jdbc:h2:mem:foo_ds_1;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MySQL`。
2. 存在系统属性 `fixture.config.driver.username` 为 `sa`。

则对于以下 YAML 文件的截取片段,

```
ds_1:
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  driverClassName: ${fixture.config.driver.driver-class-name::org.h2.Driver}
  jdbcUrl: ${fixture.config.driver.jdbc-url::jdbc:h2:mem:foo_ds_do_not_use}
  username: ${fixture.config.driver.username::}
  password: ${fixture.config.driver.password::}
```

此 YAML 截取片段将被解析为,

```
ds_1:
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  driverClassName: org.h2.Driver
  jdbcUrl: jdbc:h2:mem:foo_ds_1;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;
MODE=MySQL
  username: sa
  password:
```

在实际情况下，系统变量通常是动态定义的。假设如上系统变量均未定义，存在包含如上 YAML 截取片段的 YAML 文件 `config.yaml`，可参考如下方法，使用 HikariCP Java API 创建 DataSource 实例。

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import javax.sql.DataSource;

public class ExampleUtils {
    public DataSource createDataSource() {
        HikariConfig config = new HikariConfig();
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        config.setJdbcUrl("jdbc:shardingsphere:classpath:config.yaml?placeholder-
type=system_props");
        try {
            assert null == System.getProperty("fixture.config.driver.jdbc-url");
            assert null == System.getProperty("fixture.config.driver.username");
            System.setProperty("fixture.config.driver.jdbc-url", "jdbc:h2:mem:foo_
ds_1;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MySQL");
            System.setProperty("fixture.config.driver.username", "sa");
            return new HikariDataSource(config);
        } finally {
            System.clearProperty("fixture.config.driver.jdbc-url");
            System.clearProperty("fixture.config.driver.username");
        }
    }
}
```

用例:

- `jdbc:shardingsphere:classpath:config.yaml?placeholder-type=system_props`
- `jdbc:shardingsphere:absolutepath:/path/to/config.yaml?placeholder-type=system_props`

多个动态占位符

在单个动态占位符的基础上，用户可以在单行 YAML 使用多个动态占位符。在配置 YAML 属性值的时候，如果 YAML 属性值的部分需要动态替换，可以通过配置多个动态占位符的方式来实现

假设存在以下一组环境变量或系统属性，

1. 存在环境变量或系统属性 `FIXTURE_HOST` 为 `127.0.0.1`。
2. 存在环境变量或系统属性 `FIXTURE_PORT` 为 `3306`。
3. 存在环境变量或系统属性 `FIXTURE_DATABASE` 为 `test`。
4. 存在环境变量或系统属性 `FIXTURE_USERNAME` 为 `sa`。

则对于以下 YAML 文件的截取片段，

```
ds_1:
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  driverClassName: $${FIXTURE_DRIVER_CLASS_NAME::com.mysql.cj.jdbc.Driver}
  jdbcUrl: jdbc:mysql://$${FIXTURE_HOST::}:$${FIXTURE_PORT::}/$${FIXTURE_
DATABASE::}?sslMode=REQUIRED
  username: $${FIXTURE_USERNAME::}
  password: $${FIXTURE_PASSWORD::}
```

此 YAML 截取片段将被解析为,

```
ds_1:
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  driverClassName: com.mysql.cj.jdbc.Driver
  jdbcUrl: jdbc:mysql://127.0.0.1:3306/test?sslMode=REQUIRED
  username: sa
  password:
```

其他实现

具体可参考 https://github.com/apache/shardingsphere-plugin 。

Spring Boot

简介

ShardingSphere 提供 JDBC 驱动，开发者可以在 Spring Boot 中配置 `ShardingSphereDriver` 来使用 ShardingSphere。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

配置 Spring Boot

```
# 配置 DataSource Driver
spring.datasource.driver-class-name=org.apache.shardingsphere.driver.
ShardingSphereDriver
# 指定 YAML 配置文件
spring.datasource.url=jdbc:shardingsphere:classpath:xxx.yaml
```

`spring.datasource.url` 中的 YAML 配置文件当前支持通过多种方式获取，具体可参考 已知实现 。

使用数据源

直接使用该数据源；或者将 ShardingSphereDataSource 配置在 JPA、Hibernate、MyBatis 等 ORM 框架中配合使用。

针对 Spring Boot OSS 3 的处理

Spring Boot OSS 3 对 Jakarta EE 和 Java 17 进行了“大爆炸”升级，涉及大量复杂情况。

ShardingSphere 的 XA 分布式事务尚未在 Spring Boot OSS 3 上就绪，此限制同样适用于其他基于 Jakarta EE 9+ 的 Web Framework，如 Quarkus 3，Micronaut Framework 4 和 Helidon 3。

用户仅需要配置如下。

```
<project>
    <dependencies>
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>shardingsphere-jdbc</artifactId>
            <version>${shardingsphere.version}</version>
        </dependency>
    </dependencies>
</project>
```

针对低版本的 Spring Boot OSS 2 的特殊处理

ShardingSphere 的所有特性均可在 Spring Boot OSS 2 上使用，但低版本的 Spring Boot OSS 可能需要手动指定 SnakeYAML 的版本为 2.2 。这在 Maven 的 `pom.xml` 体现为如下内容。

```
<project>
    <dependencies>
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>shardingsphere-jdbc</artifactId>
            <version>${shardingsphere.version}</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.yaml</groupId>
            <artifactId>snakeyaml</artifactId>
            <version>2.2</version>
        </dependency>
    </dependencies>
</project>
```

如果用户是通过 https://start.spring.io/ 创建了 Spring Boot 项目，则可通过如下内容来简化配置。

```
<project>
    <properties>
        <snakeyaml.version>2.2</snakeyaml.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>shardingsphere-jdbc</artifactId>
            <version>${shardingsphere.version}</version>
        </dependency>
    </dependencies>
</project>
```

Spring 命名空间

简介

ShardingSphere 提供 JDBC 驱动, 开发者可以在 Spring 中配置 `ShardingSphereDriver` 来使用 ShardingSphere。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

配置 Spring Bean

配置项说明

名称	类型	说明
driverClass	属性	数据库 Driver，这里需要指定使用 ShardingSphereDriver
url	属性	YAML 配置文件路径

配置示例

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd">

    <bean id="shardingDataSource" class="com.zaxxer.hikari.HikariDataSource">
        <property name="driverClass" value="org.apache.shardingsphere.driver.
ShardingSphereDriver" />
        <property name="url" value="jdbc:shardingsphere:classpath:xxx.yaml" />
    </bean>
</beans>
```

使用数据源

使用方式同 Spring Boot。

9.1.2 Java API

简介

Java API 是 ShardingSphere-JDBC 中所有配置方式的基础，其他配置最终都将转化成为 Java API 的配置方式。

Java API 是最繁琐也是最灵活的配置方式，适合需要通过编程进行动态配置的场景下使用。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

构建数据源

ShardingSphere-JDBC 的 Java API 由 Database 名称、运行模式、数据源集合、规则集合以及属性配置组成。

通过 ShardingSphereDataSourceFactory 工厂创建的 ShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。

```
String databaseName = "foo_schema"; // 指定逻辑 Database 名称
ModeConfiguration modeConfig = ... // 构建运行模式
Map<String, DataSource> dataSourceMap = ... // 构建真实数据源
Collection<RuleConfiguration> ruleConfigs = ... // 构建具体规则
Properties props = ... // 构建属性配置
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(databaseName, modeConfig, dataSourceMap, ruleConfigs, props);
```

模式详情请参见模式配置。

数据源详情请参见数据源配置。

规则详情请参见规则配置。

使用数据源

可通过 DataSource 选择使用原生 JDBC，或 JPA、Hibernate、MyBatis 等 ORM 框架。

以原生 JDBC 使用方式为例：

```
// 创建 ShardingSphereDataSource
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(databaseName, modeConfig, dataSourceMap, ruleConfigs, props);

String sql = "SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_
id WHERE o.user_id=? AND o.order_id=?";
try (
        Connection conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, 10);
```

```
    ps.setInt(2, 1000);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while(rs.next()) {
            // ...
        }
    }
}
```

模式配置

背景信息

通过 Java API 方式构建运行模式。

参数解释

类名称：org.apache.shardingsphere.infra.config.mode.ModeConfiguration

可配置属性：

Standalone 持久化配置

类名称：org.apache.shardingsphere.mode.repository.standalone.StandalonePersistRepositoryConfiguration

可配置属性：

名称	数据类型	说明
type	String	持久化仓库类型
props	Properties	持久化仓库所需属性

Cluster 持久化配置

类名称：org.apache.shardingsphere.mode.repository.cluster.ClusterPersistRepositoryConfiguration

可配置属性：

名称	数据类型	说明
type	String	持久化仓库类型
namespace	String	注册中心命名空间
server-lists	String	注册中心连接地址
props	Properties	持久化仓库所需属性

注意事项

1. 生产环境建议使用集群模式部署。
2. 集群模式部署推荐使用 ZooKeeper 注册中心。
3. ZooKeeper 存在配置信息时，则以 ZooKeeper 中的配置为准。

操作步骤

引入 Maven 依赖。

```
<dependency>
 <groupId>org.apache.shardingsphere</groupId>
 <artifactId>shardingsphere-jdbc</artifactId>
 <version>${latest.release.version}</version>
</dependency>
```

注意：请将 ${latest.release.version} 更改为实际的版本号。

配置示例

Standalone 运行模式

```
ModeConfiguration modeConfig = createModeConfiguration();
Map<String, DataSource> dataSourceMap = ... // 构建真实数据源
Collection<RuleConfiguration> ruleConfigs = ... // 构建具体规则
Properties props = ... // 构建属性配置
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(databaseName, modeConfig, dataSourceMap, ruleConfigs, props);

private ModeConfiguration createModeConfiguration() {
    return new ModeConfiguration("Standalone", new
StandalonePersistRepositoryConfiguration("JDBC", new Properties()));
}
```

Cluster 运行模式 (推荐)

```
ModeConfiguration modeConfig = createModeConfiguration();
Map<String, DataSource> dataSourceMap = ... // 构建真实数据源
Collection<RuleConfiguration> ruleConfigs = ... // 构建具体规则
Properties props = ... // 构建属性配置
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(databaseName, modeConfig, dataSourceMap, ruleConfigs, props);
```

```
private ModeConfiguration createModeConfiguration() {
    return new ModeConfiguration("Cluster", new
ClusterPersistRepositoryConfiguration("ZooKeeper", "governance-sharding-db",
"localhost:2181", new Properties()));
}
```

相关参考

- ZooKeeper 注册中心安装与使用
- 持久化仓库类型的详情，请参见内置持久化仓库类型列表。

数据源配置

背景信息

ShardingSphere-JDBC 支持所有的数据库 JDBC 驱动和连接池。

本节将介绍，通过 `JAVA API` 的方式配置数据源。

操作步骤

1. 引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

> 注意：请将 `${latest.release.version}` 更改为实际的版本号。

配置示例

```
ModeConfiguration modeConfig = // 构建运行模式
Map<String, DataSource> dataSourceMap = createDataSources();
Collection<RuleConfiguration> ruleConfigs = ... // 构建具体规则
Properties props = ... // 构建属性配置
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(databaseName, modeConfig, dataSourceMap, ruleConfigs, props);

private Map<String, DataSource> createDataSources() {
    Map<String, DataSource> dataSourceMap = new HashMap<>();
    // 配置第 1 个数据源
    HikariDataSource dataSource1 = new HikariDataSource();
```

```
    dataSource1.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource1.setJdbcUrl("jdbc:mysql://localhost:3306/ds_1");
    dataSource1.setUsername("root");
    dataSource1.setPassword("");
    dataSourceMap.put("ds_1", dataSource1);

    // 配置第 2 个数据源
    HikariDataSource dataSource2 = new HikariDataSource();
    dataSource2.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource2.setJdbcUrl("jdbc:mysql://localhost:3306/ds_2");
    dataSource2.setUsername("root");
    dataSource2.setPassword("");
    dataSourceMap.put("ds_2", dataSource2);
}
```

规则配置

规则是 Apache ShardingSphere 面向可插拔的一部分。本章节是 ShardingSphere-JDBC 的 Java 规则配置参考手册。

数据分片

背景信息

数据分片 Java API 规则配置允许用户直接通过编写 Java 代码的方式，完成 ShardingSphereDataSource 对象的创建，Java API 的配置方式非常灵活，不需要依赖额外的 jar 包就能够集成各种类型的业务系统。

参数解释

配置入口

类名称：org.apache.shardingsphere.sharding.api.config.ShardingRuleConfiguration

可配置属性：

分片表配置

类名称：org.apache.shardingsphere.sharding.api.config.ShardingTableRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
logicTable	String	分片逻辑表名称	.
actua lDataNodes (?)	String	由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持行表达式	使用已知数据源与逻辑表名称生成数据节点，用于广播表或只分库不分表且所有库的表结构完全一致的情况
data baseShardi ngStrategy (?)	Sha rdingStrategy Configuration	分库策略	使用默认分库策略
t ableShardi ngStrategy (?)	Sha rdingStrategy Configuration	分表策略	使用默认分表策略
keyGenera teStrategy (?)	KeyGenerator Configuration	自增列生成器	使用默认自增主键生成器
aud itStrategy (?)	Sharding AuditStrategy Configuration	分片审计策略	使用默认分片审计策略

自动分片表配置

类名称：org.apache.shardingsphere.sharding.api.config.ShardingAutoTableRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
logicTable	String	分片逻辑表名称	.
act ualDataSources (?)	String	数据源名称，多个数据源以逗号分隔	使用全部配置的数据源
sh ardingStrategy (?)	S hardingStrateg yConfiguration	分片策略	使用默认分片策略
keyGe nerateStrategy (?)	KeyGenerato rConfiguration	自增列生成器	使用默认自增主键生成器
auditStrategy (?)	Shardi ngAuditStrateg yConfiguration	分片审计策略	使用默认分片审计策略

分片策略配置

标准分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.StandardShardingStrategyConfiguration

可配置属性:

名称	数据类型	说明
shardingColumn	String	分片列名称
shardingAlgorithmName	String	分片算法名称

复合分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.ComplexShardingStrategyConfiguration

可配置属性:

名称	数据类型	说明
shardingColumns	String	分片列名称，多个列以逗号分隔
shardingAlgorithmName	String	分片算法名称

Hint 分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.HintShardingStrategyConfiguration

可配置属性:

名称	数据类型	说明
shardingAlgorithmName	String	分片算法名称

不分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.NoneShardingStrategyConfiguration

可配置属性：无

算法类型的详情，请参见内置分片算法列表。

分布式序列策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.keygen.KeyGenerateStrategyConfiguration

可配置属性:

名称	数据类型	说明
column	String	分布式序列列名称
keyGeneratorName	String	分布式序列算法名称

算法类型的详情，请参见内置分布式序列算法列表。

分片审计策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.audit.ShardingAuditStrategyConfiguration

可配置属性:

名称	数据类型	说明
auditorNames	Collection<String>	分片审计算法名称
allowHintDisable	Boolean	是否禁用分片审计 hint

算法类型的详情，请参见内置分片审计列算法列表。

操作步骤

1. 创建真实数据源映射关系，key 为数据源逻辑名称，value 为 DataSource 对象;
2. 创建分片规则对象 ShardingRuleConfiguration,并初始化对象中的分片表对象 ShardingTableRuleConfiguration、绑定表集合、广播表集合，以及数据分片所依赖的分库策略和分表策略等参数;
3. 调用 ShardingSphereDataSourceFactory 对象的 createDataSource 方法，创建 ShardingSphereDataSource。

配置示例

```
public final class ShardingDatabasesAndTablesConfigurationPrecise {

    @Override
    public DataSource getDataSource() throws SQLException {
        return ShardingSphereDataSourceFactory.
createDataSource(createDataSourceMap(), Arrays.
asList(createShardingRuleConfiguration(), createBroadcastRuleConfiguration())), new
Properties());
```

```
    }

    private ShardingRuleConfiguration createShardingRuleConfiguration() {
        ShardingRuleConfiguration result = new ShardingRuleConfiguration();
        result.getTables().add(getOrderTableRuleConfiguration());
        result.getTables().add(getOrderItemTableRuleConfiguration());
        result.getBindingTableGroups().add(new
ShardingTableReferenceRuleConfiguration("foo", "t_order, t_order_item"));
        result.setDefaultDatabaseShardingStrategy(new
StandardShardingStrategyConfiguration("user_id", "inline"));
        result.setDefaultTableShardingStrategy(new
StandardShardingStrategyConfiguration("order_id", "standard_test_tbl"));
        Properties props = new Properties();
        props.setProperty("algorithm-expression", "demo_ds_${user_id % 2}");
        result.getShardingAlgorithms().put("inline", new AlgorithmConfiguration(
"INLINE", props));
        result.getShardingAlgorithms().put("standard_test_tbl", new
AlgorithmConfiguration("STANDARD_TEST_TBL", new Properties()));
        result.getKeyGenerators().put("snowflake", new AlgorithmConfiguration(
"SNOWFLAKE", new Properties()));
        result.getAuditors().put("sharding_key_required_auditor", new
AlgorithmConfiguration("DML_SHARDING_CONDITIONS", new Properties()));
        return result;
    }

    private ShardingTableRuleConfiguration getOrderTableRuleConfiguration() {
        ShardingTableRuleConfiguration result = new ShardingTableRuleConfiguration(
"t_order", "demo_ds_${0..1}.t_order_${[0, 1]}");
        result.setKeyGenerateStrategy(new KeyGenerateStrategyConfiguration("order_
id", "snowflake"));
        result.setAuditStrategy(new ShardingAuditStrategyConfiguration(Collections.
singleton("sharding_key_required_auditor"), true));
        return result;
    }

    private ShardingTableRuleConfiguration getOrderItemTableRuleConfiguration() {
        ShardingTableRuleConfiguration result = new ShardingTableRuleConfiguration(
"t_order_item", "demo_ds_${0..1}.t_order_item_${[0, 1]}");
        result.setKeyGenerateStrategy(new KeyGenerateStrategyConfiguration("order_
item_id", "snowflake"));
        return result;
    }

    private Map<String, DataSource> createDataSourceMap() {
        Map<String, DataSource> result = new HashMap<>();
        result.put("demo_ds_0", DataSourceUtil.createDataSource("demo_ds_0"));
        result.put("demo_ds_1", DataSourceUtil.createDataSource("demo_ds_1"));
        return result;
```

```
    }

    private BroadcastRuleConfiguration createBroadcastRuleConfiguration() {
        return new BroadcastRuleConfiguration(Collections.singletonList("t_address
"));;
    }
}
```

相关参考

- 核心特性：数据分片
- 开发者指南：数据分片

广播表

广播表 Java API 规则配置允许用户直接通过编写 Java 代码的方式，完成 ShardingSphereDataSource 对象的创建，Java API 的配置方式非常灵活，不需要依赖额外的 jar 包就能够集成各种类型的业务系统。

参数解释

类名称：org.apache.shardingsphere.broadcast.config.BroadcastRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
tables (+)	Collection<String>	广播表规则配置	

配置示例

广播表 Java API 配置示例如下：

```
public final class ShardingDatabasesAndTablesConfigurationPrecise {

    @Override
    public DataSource getDataSource() throws SQLException {
        return ShardingSphereDataSourceFactory.
createDataSource(createDataSourceMap(), Arrays.
asList(createBroadcastRuleConfiguration()), new Properties());
    }

    private Map<String, DataSource> createDataSourceMap() {
        Map<String, DataSource> result = new HashMap<>();
        result.put("demo_ds_0", DataSourceUtil.createDataSource("demo_ds_0"));
```

```
        result.put("demo_ds_1", DataSourceUtil.createDataSource("demo_ds_1"));
        return result;
    }

    private BroadcastRuleConfiguration createBroadcastRuleConfiguration() {
        return new BroadcastRuleConfiguration(Collections.singletonList("t_address
"));
    }
}
```

相关参考

- YAML 配置：广播表

读写分离

背景信息

Java API 形式配置的读写分离可以方便的适用于各种场景，不依赖额外的 jar 包，用户只需要通过 java 代码构造读写分离数据源便可以使用读写分离功能。

参数解释

配置入口

类名称：org.apache.shardingsphere.readwritesplitting.config.ReadwriteSplittingRuleConfiguration

可配置属性：

名称	数据类型	说明
dataSources (+)	Collection<Re adwriteSplittingDataSourceRuleConfiguration>	读写数据源配置
lo adBalancers (*)	Map<String, AlgorithmConfiguration>	从库负载均衡算法配置

主从数据源配置

类名称:org.apache.shardingsphere.readwritesplitting.config.rule.ReadwriteSplittingDataSourceGroupRuleConfigurati

可配置属性:

算法类型的详情，请参见内置负载均衡算法列表。

操作步骤

1. 添加读写分离数据源
2. 设置负载均衡算法
3. 使用读写分离数据源

配置示例

```
public DataSource getDataSource() throws SQLException {
        ReadwriteSplittingDataSourceRuleConfiguration dataSourceConfig = new
ReadwriteSplittingDataSourceRuleConfiguration(
                "demo_read_query_ds", "demo_write_ds", Arrays.asList("demo_read_ds_
0", "demo_read_ds_1"), "demo_weight_lb");
        Properties algorithmProps = new Properties();
        algorithmProps.setProperty("demo_read_ds_0", "2");
        algorithmProps.setProperty("demo_read_ds_1", "1");
        Map<String, AlgorithmConfiguration> algorithmConfigMap = new HashMap<>(1);
        algorithmConfigMap.put("demo_weight_lb", new AlgorithmConfiguration("WEIGHT
", algorithmProps));
        ReadwriteSplittingRuleConfiguration ruleConfig = new
ReadwriteSplittingRuleConfiguration(Collections.singleton(dataSourceConfig),
algorithmConfigMap);
        Properties props = new Properties();
        props.setProperty("sql-show", Boolean.TRUE.toString());
        return ShardingSphereDataSourceFactory.
createDataSource(createDataSourceMap(), Collections.singleton(ruleConfig), props);
    }

    private Map<String, DataSource> createDataSourceMap() {
        Map<String, DataSource> result = new HashMap<>(3, 1);
        result.put("demo_write_ds", DataSourceUtil.createDataSource("demo_write_ds
"));
        result.put("demo_read_ds_0", DataSourceUtil.createDataSource("demo_read_ds_
0"));
        result.put("demo_read_ds_1", DataSourceUtil.createDataSource("demo_read_ds_
1"));
        return result;
    }
```

相关参考

- 核心特性：读写分离
- YAML 配置：读写分离

分布式事务

配置入口

org.apache.shardingsphere.transaction.config.TransactionRuleConfiguration

可配置属性：

名称	数据类型	说明
defaultType	String	默认事务类型
providerType (?)	String	事务提供者类型
props (?)	Properties	事务属性配置

数据加密

背景信息

数据加密 Java API 规则配置允许用户直接通过编写 Java 代码的方式，完成 ShardingSphereDataSource 对象的创建，Java API 的配置方式非常灵活，不需要依赖额外的 jar 包就能够集成各种类型的业务系统。

参数解释

配置入口

类名称：org.apache.shardingsphere.encrypt.config.EncryptRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
tables (+)	Collectio n<EncryptTableRu leConfiguration>	加密表规则配置	
encryptors (+)	Map<String, Algorit hmConfiguration>	加解密算法名称和配置	

加密表规则配置

类名称：org.apache.shardingsphere.encrypt.config.rule.EncryptTableRuleConfiguration

可配置属性：

名称	数据类型	说明
name	String	表名称
columns (+)	Collection<EncryptColumnRuleConfiguration>	加密列规则配置列表

加密列规则配置

类名称：org.apache.shardingsphere.encrypt.config.rule.EncryptColumnRuleConfiguration

可配置属性：

名称	数据类型	说明
name	String	逻辑列名称
cipher	EncryptColumnItemRuleConfiguration	密文列配置
assistedQuery (?)	EncryptColumnItemRuleConfiguration	查询辅助列配置
likeQuery (?)	EncryptColumnItemRuleConfiguration	模糊查询列配置

加密列属性规则配置

类名称：org.apache.shardingsphere.encrypt.config.rule.EncryptColumnItemRuleConfiguration

可配置属性：

名称	数据类型	说明
name	String	加密列属性名称
encryptorName	String	加密列算法名称

加解密算法配置

类名称：org.apache.shardingsphere.infra.algorithm.core.config.AlgorithmConfiguration

可配置属性：

名称	数据类型	说明
name	String	加解密算法名称
type	String	加解密算法类型
properties	Properties	加解密算法属性配置

算法类型的详情，请参见内置加密算法列表。

操作步骤

1. 创建真实数据源映射关系，key 为数据源逻辑名称，value 为 DataSource 对象；
2. 创建加密规则对象 EncryptRuleConfiguration，并初始化对象中的加密表对象 EncryptTableRuleConfiguration、加密算法等参数；
3. 调用 ShardingSphereDataSourceFactory 对象的 createDataSource 方法，创建 ShardingSphereDataSource。

配置示例

```
public final class EncryptDatabasesConfiguration {

    public DataSource getDataSource() throws SQLException {
        Properties props = new Properties();
        props.setProperty("aes-key-value", "123456");
        props.setProperty("digest-algorithm-name", "SHA-1");
        EncryptColumnRuleConfiguration columnConfigAes = new
EncryptColumnRuleConfiguration("username", new EncryptColumnItemRuleConfiguration(
"username", "name_encryptor"));
        EncryptColumnRuleConfiguration columnConfigTest = new
EncryptColumnRuleConfiguration("pwd", new EncryptColumnItemRuleConfiguration("pwd",
"pwd_encryptor"));
        columnConfigTest.setAssistedQuery(new EncryptColumnItemRuleConfiguration(
"assisted_query_pwd", "pwd_encryptor"));
        columnConfigTest.setLikeQuery(new EncryptColumnItemRuleConfiguration("like_
pwd", "like_encryptor"));
        EncryptTableRuleConfiguration encryptTableRuleConfig = new
EncryptTableRuleConfiguration("t_user", Arrays.asList(columnConfigAes,
columnConfigTest));
        Map<String, AlgorithmConfiguration> encryptAlgorithmConfigs = new HashMap<>
();
        encryptAlgorithmConfigs.put("name_encryptor", new AlgorithmConfiguration(
"AES", props));
        encryptAlgorithmConfigs.put("pwd_encryptor", new AlgorithmConfiguration(
"assistedTest", props));
        encryptAlgorithmConfigs.put("like_encryptor", new AlgorithmConfiguration(
"CHAR_DIGEST_LIKE", new Properties()));
        EncryptRuleConfiguration encryptRuleConfig = new
EncryptRuleConfiguration(Collections.singleton(encryptTableRuleConfig),
encryptAlgorithmConfigs);
        return ShardingSphereDataSourceFactory.createDataSource(DataSourceUtil.
createDataSource("demo_ds"), Collections.singleton(encryptRuleConfig), props);
    }
}
```

相关参考

- 数据加密的核心特性
- 数据加密的开发者指南

数据脱敏

背景信息

数据脱敏 Java API 规则配置允许用户直接通过编写 Java 代码的方式，完成 ShardingSphereDataSource 对象的创建，Java API 的配置方式非常灵活，不需要依赖额外的 jar 包就能够集成各种类型的业务系统。

参数解释

配置入口

类名称：org.apache.shardingsphere.mask.config.MaskRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
tables (+)	Collec tion<MaskTableRu leConfiguration>	脱敏表规则配置	
maskAlgorithms (+)	Map<String, Algorit hmConfiguration>	脱敏算法名称和配置	

脱敏表规则配置

类名称：org.apache.shardingsphere.mask.config.rule.MaskTableRuleConfiguration

可配置属性：

名称	数据类型	说明
name	String	表名称
columns (+)	Collection<MaskColumnRuleConfiguration>	脱敏列规则配置列表

脱敏列规则配置

类名称：org.apache.shardingsphere.mask.config.rule.MaskColumnRuleConfiguration

可配置属性：

名称	数据类型	说明
logicColumn	String	逻辑列名称
maskAlgorithm	String	脱敏算法名称

加解密算法配置

类名称：org.apache.shardingsphere.infra.algorithm.core.config.AlgorithmConfiguration

可配置属性：

名称	数据类型	说明
name	String	脱敏算法名称
type	String	脱敏算法类型
properties	Properties	脱敏算法属性配置

算法类型的详情，请参见内置脱敏算法列表。

操作步骤

1. 创建真实数据源映射关系，key 为数据源逻辑名称，value 为 DataSource 对象；
2. 创建脱敏规则对象 MaskRuleConfiguration，并初始化对象中的脱敏表对象 MaskTableRuleConfiguration、脱敏算法等参数；
3. 调用 ShardingSphereDataSourceFactory 对象的 createDataSource 方法，创建 ShardingSphereDataSource。

配置示例

```
import java.sql.SQLException;
import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.Properties;

public final class MaskDatabasesConfiguration {

    @Override
    public DataSource getDataSource() throws SQLException {
```

```
        MaskColumnRuleConfiguration passwordColumn = new
MaskColumnRuleConfiguration("password", "md5_mask");
        MaskColumnRuleConfiguration emailColumn = new MaskColumnRuleConfiguration(
"email", "mask_before_special_chars_mask");
        MaskColumnRuleConfiguration telephoneColumn = new
MaskColumnRuleConfiguration("telephone", "keep_first_n_last_m_mask");
        MaskTableRuleConfiguration maskTableRuleConfig = new
MaskTableRuleConfiguration("t_user", Arrays.asList(passwordColumn, emailColumn,
telephoneColumn));
        Map<String, AlgorithmConfiguration> maskAlgorithmConfigs = new
LinkedHashMap<>(3, 1);
        maskAlgorithmConfigs.put("md5_mask", new AlgorithmConfiguration("MD5", new
Properties()));
        Properties beforeSpecialCharsProps = new Properties();
        beforeSpecialCharsProps.put("special-chars", "@");
        beforeSpecialCharsProps.put("replace-char", "*");
        maskAlgorithmConfigs.put("mask_before_special_chars_mask", new
AlgorithmConfiguration("MASK_BEFORE_SPECIAL_CHARS", beforeSpecialCharsProps));
        Properties keepFirstNLastMProps = new Properties();
        keepFirstNLastMProps.put("first-n", "3");
        keepFirstNLastMProps.put("last-m", "4");
        keepFirstNLastMProps.put("replace-char", "*");
        maskAlgorithmConfigs.put("keep_first_n_last_m_mask", new
AlgorithmConfiguration("KEEP_FIRST_N_LAST_M", keepFirstNLastMProps));
        MaskRuleConfiguration maskRuleConfig = new
MaskRuleConfiguration(Collections.singleton(maskTableRuleConfig),
maskAlgorithmConfigs);
        return ShardingSphereDataSourceFactory.createDataSource(DataSourceUtil.
createDataSource("demo_ds"), Collections.singleton(maskRuleConfig), new
Properties());
    }
}
```

相关参考

- 数据脱敏的核心特性
- 数据脱敏的开发者指南

影子库

背景信息

如果您只想使用 Java API 方式配置使用 ShardingSphere 影子库功能请参考以下配置。

参数解释

配置入口

类名称：org.apache.shardingsphere.shadow.config.ShadowRuleConfiguration

可配置属性：

名称	数据类型	说明
dataSources	Map<String, ShadowData SourceConfiguration>	影子数据源映射名称和配置
tables	Map<String, Shado wTableConfiguration>	影子表名称和配置
shadowAlgorithms	Map<String, Alg orithmConfiguration>	影子算法名称和配置
defaul tShadowAlgorithmName	String	默认影子算法名称

影子数据源配置

类名称：org.apache.shardingsphere.shadow.config.datasource.ShadowDataSourceConfiguration

可配置属性：

名称	数据类型	说明
productionDataSourceName	String	生产数据源名称
shadowDataSourceName	String	影子数据源名称

影子表配置

类名称：org.apache.shardingsphere.shadow.config.table.ShadowTableConfiguration

可配置属性：

名称	数据类型	说明
dataSourceNames	Collection<String>	影子表关联影子数据源映射名称列表
shadowAlgorithmNames	Collection<String>	影子表关联影子算法名称列表

影子算法配置

类名称：org.apache.shardingsphere.infra.algorithm.core.config.AlgorithmConfiguration

可配置属性：

名称	数据类型	说明
type	String	影子算法类型
props	Properties	影子算法配置

算法类型的详情，请参见内置影子算法列表。

操作步骤

1. 创建生产和影子数据源。
2. 配置影子规则

- 配置影子数据源
- 配置影子表
- 配置影子算法

配置示例

```
public final class ShadowConfiguration {

    @Override
    public DataSource getDataSource() throws SQLException {
        Map<String, DataSource> dataSourceMap = createDataSourceMap();
        return ShardingSphereDataSourceFactory.createDataSource(dataSourceMap,
createRuleConfigurations(), createShardingSphereProps());
    }

    private Map<String, DataSource> createDataSourceMap() {
        Map<String, DataSource> result = new LinkedHashMap<>();
        result.put("ds", DataSourceUtil.createDataSource("demo_ds"));
        result.put("ds_shadow", DataSourceUtil.createDataSource("shadow_demo_ds"));
        return result;
    }

    private Collection<RuleConfiguration> createRuleConfigurations() {
        Collection<RuleConfiguration> result = new LinkedList<>();
        ShadowRuleConfiguration shadowRule = new ShadowRuleConfiguration();
        shadowRule.setDataSources(createShadowDataSources());
        shadowRule.setTables(createShadowTables());
```

```
        shadowRule.setShadowAlgorithms(createShadowAlgorithmConfigurations());
        result.add(shadowRule);
        return result;
    }

    private Map<String, ShadowDataSourceConfiguration> createShadowDataSources() {
        Map<String, ShadowDataSourceConfiguration> result = new LinkedHashMap<>();
        result.put("shadow-data-source", new ShadowDataSourceConfiguration("ds",
"ds_shadow"));
        return result;
    }

    private Map<String, ShadowTableConfiguration> createShadowTables() {
        Map<String, ShadowTableConfiguration> result = new LinkedHashMap<>();
        result.put("t_user", new ShadowTableConfiguration(Collections.
singletonList("shadow-data-source"), createShadowAlgorithmNames()));
        return result;
    }

    private Collection<String> createShadowAlgorithmNames() {
        Collection<String> result = new LinkedList<>();
        result.add("user-id-insert-match-algorithm");
        result.add("simple-hint-algorithm");
        return result;
    }

    private Map<String, AlgorithmConfiguration>
createShadowAlgorithmConfigurations() {
        Map<String, AlgorithmConfiguration> result = new LinkedHashMap<>();
        Properties userIdInsertProps = new Properties();
        userIdInsertProps.setProperty("operation", "insert");
        userIdInsertProps.setProperty("column", "user_type");
        userIdInsertProps.setProperty("value", "1");
        result.put("user-id-insert-match-algorithm", new AlgorithmConfiguration(
"VALUE_MATCH", userIdInsertProps));
        return result;
    }
}
```

相关参考

影子库的特性描述

SQL 解析

背景信息

SQL 是使用者与数据库交流的标准语言。SQL 解析引擎负责将 SQL 字符串解析为抽象语法树，供 Apache ShardingSphere 理解并实现其增量功能。目前支持 MySQL, PostgreSQL, SQLServer, Oracle, openGauss 以及符合 SQL92 规范的 SQL 方言。由于 SQL 语法的复杂性，目前仍然存在少量不支持的 SQL。通过 Java API 形式使用 SQL 解析，可以方便得集成进入各种系统，灵活定制用户需求。

参数解释

类名称：org.apache.shardingsphere.parser.config.SQLParserRuleConfiguration

可配置属性：

名称	数据类型	说明
parseTreeCache (?)	CacheOption	解析语法树本地缓存配置
sqlStatementCache (?)	CacheOption	SQL 语句本地缓存配置

本地缓存配置

类名称：org.apache.shardingsphere.sql.parser.api.CacheOption

可配置属性：

名称	数据类型	说明	默认值
initialCapacity	int	本地缓存初始容量	语法树本地缓存默认值 128, SQL 语句缓存默认值 2000
maximumSize	long	本地缓存最大容量	语法树本地缓存默认值 1024，SQL 语句缓存默认值 65535

操作步骤

1. 设置本地缓存配置
2. 设置解析配置
3. 使用解析引擎解析 SQL

配置示例

```
CacheOption cacheOption = new CacheOption(128, 1024L);
SQLParserEngine parserEngine = new SQLParserEngine("MySQL", cacheOption);
ParseASTNode parseASTNode = parserEngine.parse("SELECT t.id, t.name, t.age FROM
table1 AS t ORDER BY t.id DESC;", false);
SQLStatementVisitorEngine visitorEngine = new SQLStatementVisitorEngine("MySQL");
MySQLStatement sqlStatement = visitorEngine.visit(parseASTNode);
System.out.println(sqlStatement.toString());
```

相关参考

- YAML 配置：SQL 解析

SQL 翻译

背景信息

通过 Java API 形式使用 SQL 翻译，可以方便得集成进入各种系统，灵活定制用户需求。

参数解释

类名称：org.apache.shardingsphere.sqltranslator.config.SQLTranslatorRuleConfiguration

可配置属性：

名称	数据类型	说明
type	String	SQL 翻译器类型
useOrigina lSQLWhenTranslatingFailed (?)	boolean	SQL 翻译失败是否使用原始 SQL 继续执行

操作步骤

1. 配置翻译类型 type
2. 配置 useOriginalSQLWhenTranslatingFailed 参数，是否在 SQL 翻译失败后使用原始 SQL 继续执行

配置示例

```
SQLTranslatorRuleConfiguration ruleConfig = new SQLTranslatorRuleConfiguration(
"Native", new Properties(), false);
String translatedSQL = new SQLTranslatorRule(ruleConfig).translate();
```

相关参考

- YAML 配置：SQL 翻译

混合规则

背景信息

ShardingSphere 涵盖了很多功能，例如，分库分片、读写分离、数据加密等。这些功能用户可以单独进行使用，也可以配合一起使用，下面是基于 JAVA API 的配置示例。

配置示例

```
// 分片配置
private ShardingRuleConfiguration createShardingRuleConfiguration() {
    ShardingRuleConfiguration result = new ShardingRuleConfiguration();
    result.getTables().add(getOrderTableRuleConfiguration());
    result.setDefaultDatabaseShardingStrategy(new
StandardShardingStrategyConfiguration("user_id", "inline"));
    result.setDefaultTableShardingStrategy(new
StandardShardingStrategyConfiguration("order_id", "standard_test_tbl"));
    Properties props = new Properties();
    props.setProperty("algorithm-expression", "demo_ds_${user_id % 2}");
    result.getShardingAlgorithms().put("inline", new AlgorithmConfiguration("INLINE
", props));
    result.getShardingAlgorithms().put("standard_test_tbl", new
AlgorithmConfiguration("STANDARD_TEST_TBL", new Properties()));
    result.getKeyGenerators().put("snowflake", new AlgorithmConfiguration(
"SNOWFLAKE", new Properties()));
    return result;
}
```

```
private ShardingTableRuleConfiguration getOrderTableRuleConfiguration() {
    ShardingTableRuleConfiguration result = new ShardingTableRuleConfiguration("t_
order", "demo_ds_${0..1}.t_order_${[0, 1]}");
    result.setKeyGenerateStrategy(new KeyGenerateStrategyConfiguration("order_id",
"snowflake"));
    return result;
}

// 读写分离配置
private static ReadwriteSplittingRuleConfiguration
createReadwriteSplittingConfiguration() {
    ReadwriteSplittingDataSourceRuleConfiguration dataSourceConfiguration1 = new
ReadwriteSplittingDataSourceRuleConfiguration("replica_ds_0", Arrays.asList(
"readwrite_ds_0"), true), "");
    ReadwriteSplittingDataSourceRuleConfiguration dataSourceConfiguration2 = new
ReadwriteSplittingDataSourceRuleConfiguration("replica_ds_1", Arrays.asList(
"readwrite_ds_1"), true), "");
    Collection<ReadwriteSplittingDataSourceRuleConfiguration> dataSources = new
LinkedList<>();
    dataSources.add(dataSourceRuleConfiguration1);
    dataSources.add(dataSourceRuleConfiguration2);
    return new ReadwriteSplittingRuleConfiguration(dataSources, Collections.
emptyMap());
}

// 数据加密配置
private static EncryptRuleConfiguration createEncryptRuleConfiguration() {
    Properties props = new Properties();
    props.setProperty("aes-key-value", "123456");
    props.setProperty("digest-algorithm-name", "SHA-1");
    EncryptColumnRuleConfiguration columnConfigAes = new
EncryptColumnRuleConfiguration("username", new EncryptColumnItemRuleConfiguration(
"username", "name_encryptor"));
    EncryptColumnRuleConfiguration columnConfigTest = new
EncryptColumnRuleConfiguration("pwd", new EncryptColumnItemRuleConfiguration("pwd",
"pwd_encryptor"));
    columnConfigTest.setAssistedQuery(new EncryptColumnItemRuleConfiguration(
"assisted_query_pwd", "pwd_encryptor"));
    columnConfigTest.setLikeQuery(new EncryptColumnItemRuleConfiguration("like_pwd
", "like_encryptor"));
    EncryptTableRuleConfiguration encryptTableRuleConfig = new
EncryptTableRuleConfiguration("t_user", Arrays.asList(columnConfigAes,
columnConfigTest));
    Map<String, AlgorithmConfiguration> encryptAlgorithmConfigs = new HashMap<>();
    encryptAlgorithmConfigs.put("name_encryptor", new AlgorithmConfiguration("AES",
props));
    encryptAlgorithmConfigs.put("pwd_encryptor", new AlgorithmConfiguration(
"assistedTest", props));
```

```
    encryptAlgorithmConfigs.put("like_encryptor", new AlgorithmConfiguration("CHAR_
DIGEST_LIKE", new Properties()));
    return new EncryptRuleConfiguration(Collections.
singleton(encryptTableRuleConfig), encryptAlgorithmConfigs);
}
```

数据分片路由缓存

背景信息

该项功能为**实验性功能**，需要与数据分片功能同时使用。数据分片路由缓存会将逻辑 SQL、分片键实际参数值、路由结果放入缓存中，以空间换时间，减少路由逻辑对 CPU 的使用。

建议仅在满足以下条件的情况下启用：- 纯 OLTP 场景 - ShardingSphere 进程所在机器 CPU 已达到瓶颈 - CPU 开销主要在于 ShardingSphere 路由逻辑 - 所有 SQL 已经最优且每次 SQL 执行都能命中单一分片

在不满足以上条件的情况下使用，可能对 SQL 的执行延时不会有明显改善，同时会增加内存的压力。

参数解释

类名称：org.apache.shardingsphere.sharding.api.config.cache.ShardingCacheConfiguration

可配置属性：

名称	数据类型	说明	默认值 *
allowe dMaxSqlLength	int	允许缓存的 SQL 长度限制	•
routeCache	org.apache.shar ding-sphere.sharding.api.con e.ShardingCacheOption:	路由缓存	•

类名称：org.apache.shardingsphere.sharding.api.config.cache.ShardingCacheOptionsConfiguration

可配置属性：

名称	数据类型	说明	默认值
softValues	boolean	是否软引用缓存值	•
initialCapacity	int	缓存初始容量	•
maximumSize	int	缓存最大容量	•

配置示例

```
public final class ShardingDatabasesAndTablesConfigurationPrecise {

    @Override
    public DataSource getDataSource() throws SQLException {
        return ShardingSphereDataSourceFactory.
createDataSource(createDataSourceMap(), Arrays.
asList(createShardingRuleConfiguration(), createBroadcastRuleConfiguration())), new
Properties());
    }

    private ShardingRuleConfiguration createShardingRuleConfiguration() {
        ShardingRuleConfiguration result = new ShardingRuleConfiguration();
        result.getTables().add(getOrderTableRuleConfiguration());
        result.getTables().add(getOrderItemTableRuleConfiguration());
        // ...
        result.setShardingCache(new ShardingCacheConfiguration(512, new
ShardingCacheConfiguration.RouteCacheConfiguration(65536, 262144, true)));
        return result;
    }

    private ShardingTableRuleConfiguration getOrderTableRuleConfiguration() {
        ShardingTableRuleConfiguration result = new ShardingTableRuleConfiguration(
"t_order", "demo_ds_${0..1}.t_order_${[0, 1]}");
        result.setKeyGenerateStrategy(new KeyGenerateStrategyConfiguration("order_
id", "snowflake"));
        result.setAuditStrategy(new ShardingAuditStrategyConfiguration(Collections.
singleton("sharding_key_required_auditor"), true));
        return result;
    }

    private ShardingTableRuleConfiguration getOrderItemTableRuleConfiguration() {
        ShardingTableRuleConfiguration result = new ShardingTableRuleConfiguration(
"t_order_item", "demo_ds_${0..1}.t_order_item_${[0, 1]}");
        result.setKeyGenerateStrategy(new KeyGenerateStrategyConfiguration("order_
item_id", "snowflake"));
        return result;
    }

    private Map<String, DataSource> createDataSourceMap() {
        Map<String, DataSource> result = new HashMap<>();
        result.put("demo_ds_0", DataSourceUtil.createDataSource("demo_ds_0"));
        result.put("demo_ds_1", DataSourceUtil.createDataSource("demo_ds_1"));
        return result;
    }

    private BroadcastRuleConfiguration createBroadcastRuleConfiguration() {
```

```
        return new BroadcastRuleConfiguration(Collections.singletonList("t_address
"));
    }
}
```

相关参考

- 核心特性：数据分片

单表

背景信息

单表规则用于指定哪些单表需要被 ShardingSphere 管理，也可设置默认的单表数据源。

参数解释

类名称：org.apache.shardingsphere.single.config.SingleRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
tables (+)	Collection<String>	单表规则列表	-
defaultDataSource (?)	String	单表默认数据源	-

操作步骤

1. 初始化 SingleRuleConfiguration；
2. 添加需要加载的单表，配置默认数据源。

配置示例

```
SingleRuleConfiguration ruleConfig = new SingleRuleConfiguration();
ShardingSphereDataSourceFactory.createDataSource(createDataSourceMap(), Arrays.
asList(ruleConfig), new Properties());
```

相关参考

- 单表

联邦查询

背景信息

该功能为**实验性功能，暂不适合核心系统生产环境使用**。当关联查询中的多个表分布在不同的数据库实例上时，通过开启联邦查询可以进行跨库关联查询，以及子查询。

参数解释

类名称：org.apache.shardingsphere.sqlfederation.config.SQLFederationRuleConfiguration

可配置属性：

本地缓存配置

类名称：org.apache.shardingsphere.sql.parser.api.CacheOption

可配置属性：

名称	数据类型	说明	默认值
initialCapacity	int	执行计划缓存初始容量	执行计划本地缓存初始默认值 2000
maximumSize	long	执行计划缓存最大容量	执行计划本地缓存最大默认值 65535

配置示例

```
private SQLFederationRuleConfiguration createSQLFederationRuleConfiguration() {
    CacheOption executionPlanCache = new CacheOption(2000, 65535L);
    return new SQLFederationRuleConfiguration(true, false, executionPlanCache);
}
```

相关参考

- YAML 配置：联邦查询

算法配置

分片算法

```
ShardingRuleConfiguration ruleConfiguration = new ShardingRuleConfiguration();
// algorithmName 由用户指定，需要和分片策略中的分片算法一致
// type 和 props，请参考分片内置算法：https://shardingsphere.apache.org/document/
current/cn/user-manual/common-config/builtin-algorithm/sharding/
ruleConfiguration.getShardingAlgorithms().put("algorithmName", new
AlgorithmConfiguration("xxx", new Properties()));
```

加密算法

```
// encryptorName 由用户指定，需要和加密规则中的 encryptorName 属性一致
// type 和 props，请参考加密内置算法：https://shardingsphere.apache.org/document/
current/cn/user-manual/common-config/builtin-algorithm/encrypt/
Map<String, AlgorithmConfiguration> algorithmConfigs = new HashMap<>();
algorithmConfigs.put("encryptorName", new AlgorithmConfiguration("xxx", new
Properties()));
```

读写分离负载均衡算法

```
// loadBalancerName 由用户指定，需要和读写分离规则中的 loadBalancerName 属性一致
// type 和 props，请参考读写分离负载均衡内置算法：https://shardingsphere.apache.org/
document/current/cn/user-manual/common-config/builtin-algorithm/load-balance/
Map<String, AlgorithmConfiguration> algorithmConfigs = new HashMap<>();
algorithmConfigs.put("loadBalancerName", new AlgorithmConfiguration("xxx", new
Properties()));
```

影子算法

```
// shadowAlgorithmName 由用户指定，需要和影子库规则中的 shadowAlgorithmNames 属性一致
// type 和 props，请参考影子库内置算法：https://shardingsphere.apache.org/document/
current/cn/user-manual/common-config/builtin-algorithm/shadow/
Map<String, AlgorithmConfiguration> algorithmConfigs = new HashMap<>();
algorithmConfigs.put("shadowAlgorithmName", new AlgorithmConfiguration("xxx", new
Properties()));
```

脱敏算法

```
// maskAlgorithmName 由用户指定，需要和脱敏规则中的 maskAlgorithm 属性一致
// type 和 props，请参考脱敏内置算法：https://shardingsphere.apache.org/document/
current/cn/user-manual/common-config/builtin-algorithm/mask/
Map<String, AlgorithmConfiguration> algorithmConfigs = new HashMap<>();
algorithmConfigs.put("maskAlgorithmName", new AlgorithmConfiguration("xxx", new
Properties()));
```

9.1.3 特殊 API

本章节将介绍 ShardingSphere-JDBC 的特殊场景 API。

数据分片

本章节将介绍 ShardingSphere-JDBC 的分片场景 API。

强制路由

背景信息

Apache ShardingSphere 使用 ThreadLocal 管理分片键值进行强制路由。可以通过编程的方式向 HintManager 中添加分片值，该分片值仅在当前线程内生效。

Hint 的主要使用场景：- 分片字段不存在 SQL 和数据库表结构中，而存在于外部业务逻辑。- 强制在指定数据库进行某些数据操作。

操作步骤

1. 调用 HintManager.getInstance() 获取 HintManager 实例；
2. 调用 HintManager.addDatabaseShardingValue, HintManager.addTableShardingValue 方法设置分片键值；
3. 执行 SQL 语句完成路由和执行；
4. 调用 HintManager.close 清理 ThreadLocal 中的内容。

配置示例

规则配置

Hint 分片算法需要用户实现 `org.apache.shardingsphere.sharding.api.sharding.hint.HintShardingAlgorithm` 接口。`org.apache.shardingsphere.sharding.api.sharding.hint.HintShardingAlgorithm` 存在两个内置实现为,

- `org.apache.shardingsphere.sharding.algorithm.sharding.hint.HintInlineShardingAlgorithm`
- `org.apache.shardingsphere.sharding.algorithm.sharding.classbased.ClassBasedShardingAlgorithm`

Apache ShardingSphere 在进行路由时，将会从 HintManager 中获取分片值进行路由操作。

参考配置如下:

```
rules:
- !SHARDING
  tables:
    t_order:
      actualDataNodes: demo_ds_${0..1}.t_order_${0..1}
      databaseStrategy:
        hint:
          shardingColumn: order_id
          shardingAlgorithmName: hint_class_based
      tableStrategy:
        hint:
          shardingColumn: order_id
          shardingAlgorithmName: hint_inline
  shardingAlgorithms:
    hint_class_based:
      type: CLASS_BASED
      props:
        strategy: STANDARD
        algorithmClassName: xxx.xxx.xxx.HintXXXAlgorithm
    hint_inline:
      type: HINT_INLINE
      props:
        algorithm-expression: t_order_$->{value % 4}
  defaultTableStrategy:
    none:
  defaultKeyGenerateStrategy:
    type: SNOWFLAKE
    column: order_id

props:
    sql-show: true
```

获取 HintManager

```
HintManager hintManager = HintManager.getInstance();
```

添加分片键值

- 使用 `hintManager.addDatabaseShardingValue` 来添加数据源分片键值。
- 使用 `hintManager.addTableShardingValue` 来添加表分片键值。

  分库不分表情况下，强制路由至某一个分库时，可使用 `hintManager.setDatabaseShardingValue` 方式设置分片值。

清除分片键值

分片键值保存在 ThreadLocal 中，所以需要在操作结束时调用 `hintManager.close()` 来清除 ThreadLocal 中的内容。

hintManager 实现了 AutoCloseable 接口，可推荐使用 try with resource 自动关闭。

完整代码示例

```
// Sharding database and table with using HintManager
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.addDatabaseShardingValue("t_order", 1);
    hintManager.addTableShardingValue("t_order", 2);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
        }
    }
}

// Sharding database without sharding table and routing to only one database with
using HintManager
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.setDatabaseShardingValue(3);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
```

```
        }
    }
}
```

相关参考

- 核心特性：数据分片
- 开发者指南：数据分片

读写分离

本章节将介绍 ShardingSphere-JDBC 的读写分离场景 API。

强制路由

背景信息

Apache ShardingSphere 使用 ThreadLocal 管理主库路由标记进行强制路由。可以通过编程的方式向 HintManager 中添加主库路由标记，该值仅在当前线程内生效。

Hint 在读写分离场景下，主要用于强制在主库进行某些数据操作。

操作步骤

1. 调用 `HintManager.getInstance()` 获取 HintManager 实例；
2. 调用 `HintManager.setWriteRouteOnly()` 方法设置主库路由标记；
3. 执行 SQL 语句完成路由和执行；
4. 调用 `HintManager.close()` 清理 ThreadLocal 中的内容。

配置示例

使用 Hint 强制主库路由

获取 HintManager

与基于 Hint 的数据分片相同。

设置主库路由

使用 hintManager.setWriteRouteOnly 设置主库路由。

清除分片键值

与基于 Hint 的数据分片相同。

完整代码示例

```
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.setWriteRouteOnly();
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...        }
    }
}
```

使用 Hint 路由至指定数据库

获取 HintManager

与基于 Hint 的数据分片相同。

设置路由至指定数据库

- 使用 `hintManager.setDataSourceName` 设置数据库名称。

完整代码示例

```
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.setDataSourceName("ds_0");
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
        }
```

```
    }
}
```

- 核心特性：读写分离
- 开发者指南：读写分离

分布式事务

通过 Apache ShardingSphere 使用分布式事务，与本地事务并无区别。除了透明化分布式事务的使用之外，Apache ShardingSphere 还能够在每次数据库访问时切换分布式事务类型。支持的事务类型包括：本地事务、XA 事务和柔性事务。可在创建数据库连接之前设置，缺省为 Apache ShardingSphere 启动时的默认事务类型。

使用 Java API

背景信息

使用 ShardingSphere-JDBC 时，可以通过 API 的方式使用 XA 和 BASE 模式的事务。

前提条件

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 的 Narayana 模式时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-narayana</artifactId>
    <version>${project.version}</version>
</dependency>

<!-- 使用 BASE 事务时，需要引入此模块 -->
<dependency>
```

```
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-base-seata-at</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

操作步骤

使用事务执行业务逻辑

配置示例

```
// 使用 ShardingSphereDataSource 获取连接，执行事务操作
try (Connection connection = dataSource.getConnection()) {
    connection.setAutoCommit(false);
    PreparedStatement preparedStatement = connection.prepareStatement("INSERT INTO
t_order (user_id, status) VALUES (?, ?)");
    preparedStatement.setObject(1, 1000);
    preparedStatement.setObject(2, "init");
    preparedStatement.executeUpdate();
    connection.commit();
}
```

Atomikos 事务

背景信息

Apache ShardingSphere 提供 XA 事务，默认的 XA 事务实现为 Atomikos。## 操作步骤

1. 配置事务类型
2. 配置 Atomikos

配置示例

配置事务类型

Yaml:

```
transaction:
  defaultType: XA
  providerType: Atomikos
```

配置 Atomikos

可以通过在项目的 classpath 中添加 `jta.properties` 来定制化 Atomikos 配置项。

详情请参见 Atomikos 官方文档 。

数据恢复

在项目的 `logs` 目录中会生成 `xa_tx.log`, 这是 XA 崩溃恢复时所需的日志，请勿删除。

Narayana 事务

背景信息

Apache ShardingSphere 提供 XA 事务，集成了 Narayana 的实现。

前提条件

引入 Maven 依赖

```
<properties>
    <narayana.version>5.12.7.Final</narayana.version>
    <jboss-transaction-spi.version>7.6.1.Final</jboss-transaction-spi.version>
    <jboss-logging.version>3.2.1.Final</jboss-logging.version>
</properties>

<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<dependency>
      <groupId>org.apache.shardingsphere</groupId>
      <artifactId>shardingsphere-transaction-xa-narayana</artifactId>
      <version>${shardingsphere.version}</version>
</dependency>
<dependency>
      <groupId>org.jboss.narayana.jta</groupId>
```

```
      <artifactId>jta</artifactId>
      <version>${narayana.version}</version>
</dependency>
<dependency>
       <groupId>org.jboss.narayana.jts</groupId>
       <artifactId>narayana-jts-integration</artifactId>
       <version>${narayana.version}</version>
</dependency>
<dependency>
       <groupId>org.jboss</groupId>
       <artifactId>jboss-transaction-spi</artifactId>
       <version>${jboss-transaction-spi.version}</version>
</dependency>
<dependency>
       <groupId>org.jboss.logging</groupId>
       <artifactId>jboss-logging</artifactId>
       <version>${jboss-logging.version}</version>
</dependency>
```

操作步骤

1. 配置 Narayana
2. 设置 XA 事务类型

配置示例

配置 Narayana

可以通过在项目的 classpath 中添加 `jbossts-properties.xml` 来定制化 Narayana 配置项。

详情请参见 Narayana 官方文档 。

对于 `jbossts-properties.xml` 的最小配置，ShardingSphere 要求定义 Narayana 的 `CoreEnvironmentBean.nodeIdentifier` 属性。如果 Narayana 的 object store 并非在不同的 Narayana 实例之间共享，你可以将此值设置为 `1`。一个可能的 `jbossts-properties.xml` 配置如下，

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="CoreEnvironmentBean.nodeIdentifier">1</entry>
</properties>
```

在特定情况下，你可能不会希望使用 XML 文件，那你需要在自有 Java 项目的启动类手动设置 `CoreEnvironmentBean.nodeIdentifier`。可参考如下方法调用 Narayana Java API。

```
import com.arjuna.ats.arjuna.common.CoreEnvironmentBeanException;
import com.arjuna.ats.arjuna.common.arjPropertyManager;

public class ExampleUtils {
    public void initNarayanaInstance() {
        try {
            arjPropertyManager.getCoreEnvironmentBean().setNodeIdentifier("1");
        } catch (CoreEnvironmentBeanException e) {
            throw new RuntimeException(e);
        }
    }
}
```

设置 XA 事务类型

Yaml:

```
transaction:
  defaultType: XA
  providerType: Narayana
```

Seata 事务

背景信息

Apache ShardingSphere 提供 BASE 事务，集成了 Seata 的实现。本文所指 Seata 集成均指向 Seata AT 模式。

前提条件

ShardingSphere 的 Seata 集成仅在 `apache/incubator-seata:v2.3.0` 或更高版本可用。对于 `org.apache.seata:seata-all` Maven 模块对应的 Seata Client，此限制同时作用于 HotSpot VM 和 GraalVM Native Image。引入 Maven 依赖，并排除 `org.apache.seata:seata-all` 中过时的 `org.antlr:antlr4-runtime:4.8` 的 Maven 依赖。

```
<project>
    <dependencies>
      <dependency>
         <groupId>org.apache.shardingsphere</groupId>
         <artifactId>shardingsphere-jdbc</artifactId>
         <version>${shardingsphere.version}</version>
      </dependency>
      <dependency>
         <groupId>org.apache.shardingsphere</groupId>
         <artifactId>shardingsphere-transaction-base-seata-at</artifactId>
```

```
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.seata</groupId>
        <artifactId>seata-all</artifactId>
        <version>2.3.0</version>
        <exclusions>
            <exclusion>
                <groupId>org.antlr</groupId>
                <artifactId>antlr4-runtime</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
  </dependencies>
</project>
```

受 Calcite 的影响，ShardingSphere JDBC 使用的 `commons-lang:commons-lang` 和 `org.apache.commons:commons-pool2` 与 Seata Client 存在依赖冲突，需用户根据实际情景考虑是否需要解决依赖冲突。如果不解决依赖冲突，Maven 等构建工具会在 classpath 随机使用一个冲突依赖的版本。

使用 ShardingSphere 的 Seata 集成模块时，ShardingSphere 连接的数据库实例应同时实现 ShardingSphere 的方言解析支持与 Seata AT 模式的方言解析支持。这类数据库包括但不限于 `mysql`，`gvenzl/oracle-free`, `gvenzl/oracle-xe`, `postgres`, `mcr.microsoft.com/mssql/server` 等 Docker Image。

undo_log 表限制

在每一个 ShardingSphere 涉及的真实数据库实例中均需要创建 `undo_log` 表。每种数据库的 SQL 的内容以 https://github.com/apache/incubator-seata/tree/v2.3.0/script/client/at/db 内对应的数据库为准。

相关配置

在自有项目的 ShardingSphere 的 YAML 配置文件写入如下内容，参考 分布式事务。若初始化 ShardingSphere JDBC DataSource 时使用的是 Java API，参考 分布式事务。

```
transaction:
   defaultType: BASE
   providerType: Seata
```

在 classpath 的根目录中增加 `seata.conf` 文件，配置文件格式参考 `org.apache.seata.config.FileConfiguration` 的 JavaDoc。

`seata.conf` 存在四个属性，

1. `shardingsphere.transaction.seata.at.enable`，当此值为 `true` 时，开启 ShardingSphere 的 Seata AT 集成。存在默认值为 `true`
2. `shardingsphere.transaction.seata.tx.timeout`，全局事务超时（秒）。存在默认值为 `60`

3. `client.application.id`，应用唯一主键，用于设置 Seata Transaction Manager Client 和 Seata Resource Manager Client 的 `applicationId`
4. `client.transaction.service.group`，所属事务组，用于设置 Seata Transaction Manager Client 和 Seata Resource Manager Client 的 `transactionServiceGroup`。存在默认值为 `default`

一个完全配置的 `seata.conf` 如下，

```
shardingsphere.transaction.seata.at.enable = true
shardingsphere.transaction.seata.tx.timeout = 60

client {
    application.id = example
    transaction.service.group = default_tx_group
}
```

一个最小配置的 `seata.conf` 如下。由 ShardingSphere 管理的 `seata.conf` 中，`client.transaction.service.group` 的默认值为 `default` 是出于历史原因。假设用户使用的 Seata Server 和 Seata Client 的 `registry.conf` 中，`registry.type` 和 `config.type` 均为 `file`，则对于 `registry.conf` 的 `config.file.name` 配置的 `.conf` 文件中，事务分组名在 `apache/incubator-seata:v1.5.1` 及之后默认值为 `default_tx_group`，在 `apache/incubator-seata:v1.5.1` 之前则为 `my_test_tx_group`。

```
client.application.id = example
```

根据实际场景修改 Seata 的 `registry.conf` 文件。

操作步骤

1. 启动 Seata Server
2. 创建 `undo_log` 表
3. 添加 Seata 配置

配置示例

启动 Seata Server 和 MySQL Server

编写 Docker Compose 文件来启动 Seata Server 和 MySQL Server。

```yaml
services:
   apache-seata-server:
      image: apache/seata-server:2.3.0
      ports:
         - "8091:8091"
   mysql:
      image: mysql:9.1.0
```

```
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
      - ./mysql/docker-entrypoint-initdb.d:/docker-entrypoint-initdb.d
    ports:
      - "3306:3306"
```

./docker-entrypoint-initdb.d 文件夹包含文件为 init.sh，内容如下，

```
#!/bin/bash
set -e

mysql -uroot -p"$MYSQL_ROOT_PASSWORD" <<EOSQL
CREATE DATABASE demo_ds_0;
CREATE DATABASE demo_ds_1;
CREATE DATABASE demo_ds_2;
EOSQL

for i in "demo_ds_0" "demo_ds_1" "demo_ds_2"
do
mysql -uroot -p"$MYSQL_ROOT_PASSWORD" "$i" <<'EOSQL'
CREATE TABLE IF NOT EXISTS `undo_log`
(
    `branch_id`     BIGINT       NOT NULL COMMENT 'branch transaction id',
    `xid`           VARCHAR(128) NOT NULL COMMENT 'global transaction id',
    `context`       VARCHAR(128) NOT NULL COMMENT 'undo_log context,such as
serialization',
    `rollback_info` LONGBLOB     NOT NULL COMMENT 'rollback info',
    `log_status`    INT(11)      NOT NULL COMMENT '0:normal status,1:defense status
',
    `log_created`   DATETIME(6)  NOT NULL COMMENT 'create datetime',
    `log_modified`  DATETIME(6)  NOT NULL COMMENT 'modify datetime',
    UNIQUE KEY `ux_undo_log` (`xid`, `branch_id`)
) ENGINE = InnoDB AUTO_INCREMENT = 1 DEFAULT CHARSET = utf8mb4 COMMENT ='AT
transaction mode undo table';
ALTER TABLE `undo_log` ADD INDEX `ix_log_created` (`log_created`);

CREATE TABLE IF NOT EXISTS t_order (
   order_id BIGINT NOT NULL AUTO_INCREMENT,
   order_type INT(11),
   user_id INT NOT NULL,
   address_id BIGINT NOT NULL,
   status VARCHAR(50),
   PRIMARY KEY (order_id)
);
EOSQL
done
```

在业务项目的 classpath 创建 seata.conf

在业务项目的 classpath 创建 seata.conf，内容如下，

```
service {
    default.grouplist = "127.0.0.1:8091"
    vgroupMapping.default_tx_group = "default"
}
```

在业务项目的 classpath 创建 file.conf

在业务项目的 classpath 创建 file.conf，内容如下，

```
client {
    application.id = test
    transaction.service.group = default_tx_group
}
```

在业务项目的 classpath 创建 registry.conf

在业务项目的 classpath 创建 registry.conf，内容如下，

```
registry {
  type = "file"
  file {
    name = "file.conf"
  }
}
config {
  type = "file"
  file {
    name = "file.conf"
  }
}
```

在业务项目添加 JDBC Driver 和创建 ShardingSphere 配置文件

在业务项目引入前提条件涉及的依赖后，额外添加 MySQL JDBC Driver 的 Maven 依赖，

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.1.0</version>
</dependency>
```

在业务项目的 classpath 上编写 ShardingSphere 数据源的配置文件 demo.yaml，

```yaml
dataSources:
   ds_0:
      dataSourceClassName: com.zaxxer.hikari.HikariDataSource
      driverClassName: com.mysql.cj.jdbc.Driver
      jdbcUrl: jdbc:mysql://localhost:3306/demo_ds_0?sslMode=REQUIRED
      username: root
      password: example
   ds_1:
      dataSourceClassName: com.zaxxer.hikari.HikariDataSource
      driverClassName: com.mysql.cj.jdbc.Driver
      jdbcUrl: jdbc:mysql://localhost:3306/demo_ds_1?sslMode=REQUIRED
      username: root
      password: example
   ds_2:
      dataSourceClassName: com.zaxxer.hikari.HikariDataSource
      driverClassName: com.mysql.cj.jdbc.Driver
      jdbcUrl: jdbc:mysql://localhost:3306/demo_ds_2?sslMode=REQUIRED
      username: root
      password: example
rules:
   - !SHARDING
      tables:
         t_order:
            actualDataNodes: ds_$->{0..2}.t_order
            keyGenerateStrategy:
               column: order_id
               keyGeneratorName: snowflake
      defaultDatabaseStrategy:
         standard:
            shardingColumn: user_id
            shardingAlgorithmName: inline
      shardingAlgorithms:
         inline:
            type: INLINE
            props:
               algorithm-expression: ds_${user_id % 2}
      keyGenerators:
         snowflake:
            type: SNOWFLAKE
transaction:
   defaultType: BASE
   providerType: Seata
```

享受集成

在 ShardingSphere 的数据源上可开始享受集成,

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
@SuppressWarnings({"SqlNoDataSourceInspection", "AssertWithSideEffects"})
public class ExampleTest {
    void test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml");
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        try (HikariDataSource dataSource = new HikariDataSource(config)) {
            try (Connection conn = dataSource.getConnection()) {
                try {
                    conn.setAutoCommit(false);
                    conn.createStatement().executeUpdate("INSERT INTO t_order
(user_id, order_type, address_id, status) VALUES (2024, 1, 2024, 'INSERT_TEST')");
                    conn.createStatement().executeUpdate("INSERT INTO t_order_does_
not_exist (test_id_does_not_exist) VALUES (2024)");
                    conn.commit();
                } catch (final SQLException ignored) {
                    conn.rollback();
                } finally {
                    conn.setAutoCommit(true);
                }
            }
            try (Connection conn = dataSource.getConnection()) {
                assert !conn.createStatement().executeQuery("SELECT * FROM t_order_
item WHERE user_id = 2024").next();
            }
        }
    }
}
```

使用限制

ShardingSphere 的 Seata 集成不支持隔离级别。

ShardingSphere 的 Seata 集成将获取到的 Seata 全局事务置入线程的局部变量。而 `org.apache.seata.spring.annotation.GlobalTransactionScanner` 则是采用 Dynamic Proxy 的方式对方法进行增强。这意味着用户在使用 ShardingSphere 的 Seata 集成时，用户应避免使用 `org.apache.seata:seata-all` 的 Java API，除非用户正在混合使用 ShardingSphere 的 Seata 集成与 Seata Client 的 TCC 模式特性。

针对 ShardingSphere 数据源，讨论 7 种情况，

1. 手动获取从 ShardingSphere 数据源创建的 `java.sql.Connection` 实例，并手动调用 `setAutoCommit()`, `commit()` 和 `rollback()` 方法，这是被允许的。
2. 在函数上使用 Jakarta EE 8 的 `javax.transaction.Transactional` 注解，这是被允许的。
3. 在函数上使用 Jakarta EE 9/10 的 `jakarta.transaction.Transactional` 注解，这是被允许的。
4. 在函数上使用 Spring Framework 的 `org.springframework.transaction.annotation.Transactional` 注解，这是被允许的。
5. 手动获取从 `org.springframework.transaction.PlatformTransactionManager` 实例创建的 `org.springframework.transaction.support.TransactionTemplate` 实例，并使用 `org.springframework.transaction.support.TransactionTemplate#execute(org.springframework.transaction.support.TransactionCallback)`，这是被允许的。
6. 在函数上使用 `org.apache.seata.spring.annotation.GlobalTransactional` 注解，这是**不被允许的**。
7. 手动从 `org.apache.seata.tm.api.GlobalTransactionContext` 创建 `org.apache.seata.tm.api.GlobalTransaction` 实例，调用 `org.apache.seata.tm.api.GlobalTransaction` 实例的 `begin()`, `commit()` 和 `rollback()` 方法，这是**不被允许的**。

在使用 Spring Boot 的实际情景中，`com.alibaba.cloud:spring-cloud-starter-alibaba-seata` 和 `org.apache.seata:seata-spring-boot-starter` 常常被其他 Maven 依赖传递引入。为了避开事务冲突，用户需要在 Spring Boot 的配置文件中将 `seata.enable-auto-data-source-proxy` 的属性置为 `false`。一个可能的依赖关系如下。

```
<project>
    <dependencies>
      <dependency>
         <groupId>org.apache.shardingsphere</groupId>
         <artifactId>shardingsphere-jdbc</artifactId>
         <version>${shardingsphere.version}</version>
      </dependency>
      <dependency>
         <groupId>org.apache.shardingsphere</groupId>
         <artifactId>shardingsphere-transaction-base-seata-at</artifactId>
         <version>${shardingsphere.version}</version>
      </dependency>
      <dependency>
         <groupId>org.apache.seata</groupId>
         <artifactId>seata-spring-boot-starter</artifactId>
         <version>2.3.0</version>
         <exclusions>
            <exclusion>
               <groupId>org.antlr</groupId>
```

```
                <artifactId>antlr4-runtime</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
  </dependencies>
</project>
```

classpath 下对应的 `application.yml` 需要包含以下配置。在此情况下，在 Spring Boot 的 `application.yaml` 内定义 Seata 的 `registry.conf` 的等价配置依然有效。当下游项目使用 `org.apache.shardingsphere:shardingsphere-transaction-base-seata-at` 的 Maven 模块时，总是被鼓励使用 `registry.conf` 配置 Seata Client。

```
seata:
  enable-auto-data-source-proxy: false
```

与 Seata TCC 模式特性混合使用

对于设置开启 ShardingSphere 的 Seata 集成的情况下，在与 ShardingSphere JDBC DataSource 无关的业务函数中，如需在业务函数使用 Seata Client 的 Seata TCC 模式相关的特性，可实例化一个未代理的普通 TCC 接口实现类，然后使用 `org.apache.seata.integration.tx.api.util.ProxyUtil` 创建一个代理的 TCC 接口类，并调用 TCC 接口实现类 `Try`，`Confirm`，`Cancel` 三个阶段对应的函数。

对于由 Seata TCC 模式而引入的 `org.apache.seata.spring.annotation.GlobalTransactional` 注解或 Seata TCC 模式涉及的业务函数中需要与数据库实例交互，此注解标记的业务函数内不应使用 ShardingSphere JDBC DataSource，而是应该手动创建 `javax.sql.DataSource` 实例，或从自定义的 Spring Bean 中获取 `javax.sql.DataSource` 实例。

跨服务调用的事务传播

跨服务调用场景下的事务传播，并不像单个微服务内的事务操作一样开箱即用。对于 Seata Server，跨服务调用场景下的事务传播，要把 XID 通过服务调用传递到服务提供方，并绑定到 `org.apache.seata.core.context.RootContext` 中去。参考 https://seata.apache.org/docs/user/api/ 。这需要讨论两种情况，

1. 在使用 ShardingSphere JDBC 的场景下，跨多个微服务的事务场景需要考虑在起点微服务的上下文使用 `org.apache.seata.core.context.RootContext.getXID()` 获取 Seata XID 后，通过 HTTP 或 RPC 等手段传递给终点微服务，并在终点微服务的 Filter 或 Spring WebMVC HandlerInterceptor 中处理。Spring WebMVC HandlerInterceptor 仅适用于 Spring Boot 微服务，对 Quarkus，Micronaut Framework 和 Helidon 无效。
2. 在使用 ShardingSphere Proxy 的场景下，多个微服务均对着 ShardingSphere Proxy 的逻辑数据源操作本地事务，这将在 ShardingSphere Proxy 的服务端来转化为对分布式事务的操作，不需要考虑额外的 Seata XID。

引入简单场景来继续讨论在使用 ShardingSphere JDBC 的场景下，跨服务调用的事务传播。假设存在以下已知微服务和中间件的 Docker Image 实例。

1. MySQL 数据库实例 `a-mysql`，所有 database 均已创建 `UNDO_LOG` 表和业务表。
2. MySQL 数据库实例 `b-mysql`，所有 database 均已创建 `UNDO_LOG` 表和业务表。
3. 使用 `file` 作为配置中心和注册中心的 Seata Server 实例 `a-seata-server`。
4. 微服务实例 `a-service`。此微服务创建仅配置数据库实例 `a-mysql` 的 ShardingSphere JDBC DataSource。此 ShardingSphere JDBC DataSource 配置使用连接到 Seata Server 实例 `a-seata-server` 的 Seata AT 集成，其 Seata Application Id 为 `service-a`，其 Seata 事务分组为 `default_tx_group`，其 `Virtual Group Mapping` 指向的 Seata Transaction Coordinator 集群分组为 `default`。此微服务实例 `a-service` 暴露单个 Restful API 的 GET 端点为 `/hello`，此 Restful API 端点的业务函数 `aMethod` 使用了普通的本地事务注解。若此微服务基于 Spring Boot 2，

```
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DemoController {
   @Transactional
   @GetMapping("/hello")
   public String aMethod() {
      // ... 对数据库实例 `a-mysql` 做 UPDATE 操作
      return "Hello World!";
   }
}
```

5. 微服务实例 `b-service`。此微服务创建仅配置数据库实例 `b-mysql` 的 ShardingSphere JDBC DataSource。此 ShardingSphere JDBC DataSource 配置使用连接到 Seata Server 实例 `a-seata-server` 的 Seata AT 集成，其 Seata Application Id 为 `service-b`，其 Seata 事务分组为 `default_tx_group`，其 `Virtual Group Mapping` 指向的 Seata Transaction Coordinator 集群分组为 `default`。此微服务实例 `b-service` 的业务函数 `bMethod` 使用普通的本地事务注解，并在 `bMethod` 通过 HTTP Client 调用微服务实例 `a-service` 的 `/hello` Restful API 端点。若此微服务基于 Spring Boot 2，

```
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.client.RestTemplate;

@Service
public class DemoService {
   @Transactional
   public void bMethod() {
      RestTemplate restTemplate = new RestTemplateBuilder().build();
      restTemplate.getForEntity("http://a-service/hello", String.class);
      // ... 对数据库实例 `b-mysql` 做 UPDATE 操作
   }
```

```
}
```

对于此简单场景，此时存在单个 Seata Server Cluster，其包含单个 `Virtual Group` 为 `default`。此 `Virtual Group` 包含单个 Seata Server 实例为 `a-seata-server`。

讨论单服务调用的事务传播。当微服务实例 `a-service` 的业务函数 `aMethod` 抛出异常，在业务函数内对 MySQL 数据库实例 `a-mysql` 的更改将被正常回滚。

讨论跨服务调用的事务传播。当微服务实例 `b-service` 的业务函数 `bMethod` 抛出异常，在业务函数内对 MySQL 数据库实例 `b-mysql` 的更改将被正常回滚，而微服务实例 `a-service` 的 `org.apache.seata.core.context.RootContext` 未绑定微服务实例 `b-service` 的业务函数 `bMethod` 的 Seata XID，因此在业务函数内对 MySQL 数据库实例 `a-mysql` 的更改将不会被回滚。

为了实现当微服务实例 `b-service` 的业务函数 `bMethod` 抛出异常，在业务函数内对 MySQL 数据库实例 `a-mysql` 和 `b-mysql` 的更改均被正常回滚，讨论不同场景下的常见处理方案。

1. 微服务实例 `a-service` 和 `b-service` 均为基于 Jakarta EE 8 的 Spring Boot 2 微服务。用户可在微服务实例 `b-service` 的业务函数 `bMethod` 使用 `org.springframework.web.client.RestTemplate` 把 XID 通过服务调用传递到微服务实例 `a-service`。可能的改造逻辑如下。

```java
import org.apache.seata.core.context.RootContext;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.client.RestTemplate;

@Service
public class DemoService {
    @Transactional
    public void bMethod() {
        RestTemplate restTemplate = new RestTemplateBuilder().
additionalInterceptors((request, body, execution) -> {
                    String xid = RootContext.getXID();
                    if (null != xid) {
                        request.getHeaders().add(RootContext.KEY_XID, xid);
                    }
                    return execution.execute(request, body);
                })
                .build();
        restTemplate.getForEntity("http://a-service/hello", String.class);
        // ... 对数据库实例 `b-mysql` 做 UPDATE 操作
    }
}
```

此时在微服务实例 `a-service` 需要添加自定义的 `org.springframework.web.servlet.config.annotation.WebMvcConfigurer` 实现。

```java
import org.apache.seata.integration.http.TransactionPropagationInterceptor;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
```

```
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class CustomWebMvcConfigurer implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new TransactionPropagationInterceptor());
    }
}
```

此时，当微服务实例 `b-service` 的业务函数 `bMethod` 抛出异常，在业务函数内对 MySQL 数据库实例 `a-mysql` 和 `b-mysql` 的更改均被正常回滚。

2. 微服务实例 `a-service` 和 `b-service` 均为基于 Jakarta EE 9/10 的 Spring Boot 3 微服务。用户可在微服务实例 `b-service` 的业务函数 `bMethod` 使用 `org.springframework.web.client.RestClient` 把 XID 通过服务调用传递到微服务实例 `a-service`。可能的改造逻辑如下。

```
import org.apache.seata.core.context.RootContext;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.client.RestClient;

@Service
public class DemoService {
    @Transactional
    public void bMethod() {
        RestClient restClient = RestClient.builder().requestInterceptor((request,
body, execution) -> {
                    String xid = RootContext.getXID();
                    if (null != xid) {
                        request.getHeaders().add(RootContext.KEY_XID, xid);
                    }
                    return execution.execute(request, body);
                })
                .build();
        restClient.get().uri("http://a-service/hello").retrieve().body(String.
class);
        // ... 对数据库实例 `b-mysql` 做 UPDATE 操作
    }
}
```

此时在微服务实例 `a-service` 需要添加自定义的 `org.springframework.web.servlet.config.annotation.WebMvcConfigurer` 实现。

```
import org.apache.seata.integration.http.JakartaTransactionPropagationInterceptor;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
```

```
@Configuration
public class CustomWebMvcConfigurer implements WebMvcConfigurer {

   @Override
   public void addInterceptors(InterceptorRegistry registry) {
      registry.addInterceptor(new JakartaTransactionPropagationInterceptor());
   }
}
```

此时，当微服务实例 `b-service` 的业务函数 `bMethod` 抛出异常，在业务函数内对 MySQL 数据库实例 `a-mysql` 和 `b-mysql` 的更改均被正常回滚。

3. 微服务实例 `a-service` 和 `b-service` 均为 Spring Boot 微服务，但使用的 API 网关中间件阻断了所有包含 `TX_XID` 的 HTTP Header 的 HTTP 请求。用户需要考虑更改把 XID 通过服务调用传递到微服务实例 `a-service` 使用的 HTTP Header，或使用 RPC 框架把 XID 通过服务调用传递到微服务实例 `a-service`。参考 https://github.com/apache/incubator-seata/tree/v2.3.0/integration 。
4. 微服务实例 `a-service` 和 `b-service` 均为 Quarkus，Micronaut Framework 和 Helidon 等微服务。此情况下无法使用 Spring WebMVC HandlerInterceptor。可参考如下 Spring Boot 3 的自定义 WebMvcConfigurer 实现，来实现 Filter。

```
import org.apache.seata.common.util.StringUtils;
import org.apache.seata.core.context.RootContext;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.context.annotation.Configuration;
import org.springframework.lang.NonNull;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
@Configuration
public class CustomWebMvcConfigurer implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new HandlerInterceptor() {
            @Override
            public boolean preHandle(@NonNull HttpServletRequest request, @NonNull
HttpServletResponse response, @NonNull Object handler) {
                String rpcXid = request.getHeader(RootContext.KEY_XID);
                String xid = RootContext.getXID();
                if (StringUtils.isBlank(xid) && StringUtils.isNotBlank(rpcXid)) {
                    RootContext.bind(rpcXid);
                }
                return true;
            }
            @Override
            public void afterCompletion(@NonNull HttpServletRequest request,
@NonNull HttpServletResponse response, @NonNull Object handler, Exception ex) {
```

```
                if (RootContext.inGlobalTransaction()) {
                    String rpcXid = request.getHeader(RootContext.KEY_XID);
                    String xid = RootContext.getXID();
                    if (StringUtils.isNotBlank(xid)) {
                        String unbindXid = RootContext.unbind();
                        if (!StringUtils.equalsIgnoreCase(rpcXid, unbindXid)) {
                            if (StringUtils.isNotBlank(unbindXid)) {
                                RootContext.bind(unbindXid);
                            }
                        }
                    }
                }
            }
        });
    }
}
```

5. 微服务实例 `a-service` 和 `b-service` 均为 Spring Boot 微服务，但使用的组件是 Spring WebFlux 而非 Spring WebMVC。在反应式编程 API 下 ShardingSphere JDBC 无法处理 R2DBC DataSource，仅可处理 JDBC DataSource。在使用 WebFlux 组件的 Spring Boot 微服务中应避免创建 ShardingSphere JDBC DataSource。

Log 配置

在业务项目启动 Seata Client 后，可能看到如下的 Error Log。

```
[ERROR] 2024-12-20 11:46:43.878 [ForkJoinPool.commonPool-worker-1] o.a.s.config.
ConfigurationFactory - failed to load non-spring configuration :not found service
provider for : org.apache.seata.config.ConfigurationProvider
org.apache.seata.common.loader.EnhancedServiceNotFoundException: not found service
provider for : org.apache.seata.config.ConfigurationProvider
```

根据 https://github.com/apache/incubator-seata/issues/6886 ，抛出此异常是 Seata Client 的预期行为。用户可通过在业务项目的 classpath 放置 `logback.xml` 对 Seata Client 的日志加以配置。

9.1.4 可选插件

ShardingSphere 默认情况下仅包含核心 SPI 的实现，在 Git Source 存在一部分包含第三方依赖的 SPI 实现的插件未包含在内。可在 https://central.sonatype.com/ 进行检索。

所有插件对应的 SPI 和 SPI 的已有实现类均可在 https://shardingsphere.apache.org/document/current/cn/dev-manual/ 检索。

下以 `groupId:artifactId` 的表现形式列出 ShardingSphere-JDBC 所有的内置插件。

- `org.apache.shardingsphere:shardingsphere-authority-core`，用户权限加载逻辑核心

- `org.apache.shardingsphere:shardingsphere-cluster-mode-core`，集群模式配置信息持久化定义核心
- `org.apache.shardingsphere:shardingsphere-db-discovery-core`，高可用核心
- `org.apache.shardingsphere:shardingsphere-encrypt-core`，数据加密核心
- `org.apache.shardingsphere:shardingsphere-infra-context`，Context 的内核运行与元数据刷新机制
- `org.apache.shardingsphere:shardingsphere-mask-core`，数据脱敏核心
- `org.apache.shardingsphere:shardingsphere-mysql-dialect-exception`,数据库网关的 MySQL 实现
- `org.apache.shardingsphere:shardingsphere-parser-core`，SQL 解析核心
- `org.apache.shardingsphere:shardingsphere-postgresql-dialect-exception`,数据库网关的 PostgreSQL 实现
- `org.apache.shardingsphere:shardingsphere-readwrite-splitting-core`，读写分离核心
- `org.apache.shardingsphere:shardingsphere-shadow-core`，影子库核心
- `org.apache.shardingsphere:shardingsphere-sharding-core`，数据分片核心
- `org.apache.shardingsphere:shardingsphere-single-core`，单表（所有的分片数据源中仅唯一存在的表）核心
- `org.apache.shardingsphere:shardingsphere-sql-federation-core`，联邦查询执行器核心
- `org.apache.shardingsphere:shardingsphere-sql-parser-mysql`,SQL 解析的 MySQL 方言实现
- `org.apache.shardingsphere:shardingsphere-sql-parser-postgresql`, SQL 解析的 PostgreSQL 方言实现
- `org.apache.shardingsphere:shardingsphere-sql-parser-opengauss`，SQL 解析的 OpenGauss 方言实现
- `org.apache.shardingsphere:shardingsphere-sql-parser-oracle`，SQL 解析的 Oracle 方言解析实现
- `org.apache.shardingsphere:shardingsphere-sql-parser-sqlserver`，SQL 解析的 SQL Server 方言实现
- `org.apache.shardingsphere:shardingsphere-sql-parser-doris`，SQL 解析的 Doris 方言实现
- `org.apache.shardingsphere:shardingsphere-sql-parser-presto`，SQL 解析的 Presto 方言实现
- `org.apache.shardingsphere:shardingsphere-sql-parser-sql92`, SQL 解析的 SQL 92 方言实现

- `org.apache.shardingsphere:shardingsphere-standalone-mode-core`，单机模式配置信息持久化定义核心
- `org.apache.shardingsphere:shardingsphere-standalone-mode-repository-jdbc-h2`，单机模式配置信息持久化定义的 H2 实现
- `org.apache.shardingsphere:shardingsphere-traffic-core`，流量治理核心
- `org.apache.shardingsphere:shardingsphere-transaction-core`，XA 分布式事务管理器核心

如果 ShardingSphere-JDBC 需要使用可选插件，需要在 Maven Central 下载包含其 SPI 实现的 JAR 和其依赖的 JAR。

下以 `groupId:artifactId` 的表现形式列出所有的可选插件。

- 集群模式配置信息持久化定义
  - `org.apache.shardingsphere:shardingsphere-cluster-mode-repository-zookeeper`，基于 Zookeeper 的持久化实现
  - `org.apache.shardingsphere:shardingsphere-cluster-mode-repository-etcd`，基于 Etcd 的持久化实现
- XA 分布式事务管理器
  - `org.apache.shardingsphere:shardingsphere-transaction-xa-narayana`，基于 Narayana 的 XA 分布式事务管理器
- 行表达式
  - `org.apache.shardingsphere:shardingsphere-infra-expr-espresso`，基于 GraalVM Truffle 的 Espresso 实现的使用 Groovy 语法的行表达式
- 数据库类型识别
  - `org.apache.shardingsphere:shardingsphere-infra-database-hive`，对 Hive 的 jdbcURL 的识别适配，元数据加载实现
  - `org.apache.shardingsphere:shardingsphere-infra-database-presto`，对 Presto 的 jdbcURL 的识别适配，元数据加载实现
- SQL 解析
  - `org.apache.shardingsphere:shardingsphere-parser-sql-clickhouse`，SQL 解析的 ClickHouse 方言实现
  - `org.apache.shardingsphere:shardingsphere-parser-sql-hive`，SQL 解析的 Hive 方言实现

除了以上可选插件外，ShardingSphere 社区开发者还贡献了大量的插件实现，可以在 ShardingSphere Plugin 仓库中查看插件的使用说明，ShardingSphere Plugin 仓库中的插件会和 ShardingSphere 保持相同的发布节奏，可以手动打包安装到 ShardingSphere 中。

ClickHouse

背景信息

ShardingSphere 默认情况下不提供对 `com.clickhouse.jdbc.ClickHouseDriver` 的 `driverClassName` 的支持。ShardingSphere 对 ClickHouse JDBC Driver 的支持位于可选模块中。

前提条件

要在 ShardingSphere 的配置文件为数据节点使用类似 `jdbc:ch://localhost:8123/demo_ds_0` 的 `jdbcUrl`，可能的 Maven 依赖关系如下，

```
<dependencies>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-jdbc</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-parser-sql-clickhouse</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>com.clickhouse</groupId>
        <artifactId>clickhouse-jdbc</artifactId>
        <classifier>http</classifier>
        <version>0.6.3</version>
    </dependency>
</dependencies>
```

配置示例

启动 ClickHouse

编写 Docker Compose 文件来启动 ClickHouse。

```
services:
  clickhouse-server:
    image: clickhouse/clickhouse-server:25.4.5.24
    ports:
      - "8123:8123"
```

创建业务表

通过第三方工具在 ClickHouse 内创建业务库与业务表。以 DBeaver Community 为例，若使用 Ubuntu 22.04.4，可通过 Snapcraft 快速安装，

```
sudo apt update && sudo apt upgrade -y
sudo snap install dbeaver-ce
snap run dbeaver-ce
```

在 DBeaver Community 内，使用 `jdbc:ch://localhost:8123/default` 的 `jdbcUrl`，`default` 的 `username` 连接至 ClickHouse，`password` 留空。执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE DATABASE demo_ds_0;
CREATE DATABASE demo_ds_1;
CREATE DATABASE demo_ds_2;
```

分别使用 `jdbc:ch://localhost:8123/demo_ds_0`，`jdbc:ch://localhost:8123/demo_ds_1` 和 `jdbc:ch://localhost:8123/demo_ds_2` 的 `jdbcUrl` 连接至 ClickHouse 来执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
create table IF NOT EXISTS t_order (
    order_id   Int64 NOT NULL,
    order_type Int32,
    user_id    Int32 NOT NULL,
    address_id Int64 NOT NULL,
    status     VARCHAR(50)
) engine = MergeTree
    primary key (order_id)
    order by (order_id);

TRUNCATE TABLE t_order;
```

在业务项目创建 ShardingSphere 数据源

在业务项目引入前提条件涉及的依赖后，在业务项目的 classpath 上编写 ShardingSphere 数据源的配置文件 `demo.yaml`，

```
dataSources:
    ds_0:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: com.clickhouse.jdbc.ClickHouseDriver
        jdbcUrl: jdbc:ch://localhost:8123/demo_ds_0
        username: default
        password:
    ds_1:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
```

```
        driverClassName: com.clickhouse.jdbc.ClickHouseDriver
        jdbcUrl: jdbc:ch://localhost:8123/demo_ds_1
        username: default
        password:
    ds_2:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: com.clickhouse.jdbc.ClickHouseDriver
        jdbcUrl: jdbc:ch://localhost:8123/demo_ds_2
        username: default
        password:
rules:
- !SHARDING
    tables:
      t_order:
        actualDataNodes: <LITERAL>ds_0.t_order, ds_1.t_order, ds_2.t_order
        keyGenerateStrategy:
          column: order_id
          keyGeneratorName: snowflake
    defaultDatabaseStrategy:
      standard:
        shardingColumn: user_id
        shardingAlgorithmName: inline
    shardingAlgorithms:
      inline:
        type: INLINE
        props:
          algorithm-expression: ds_${user_id % 2}
    keyGenerators:
      snowflake:
        type: SNOWFLAKE
```

享受集成

创建 ShardingSphere 的数据源以享受集成,

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
public class ExampleUtils {
    void test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml");
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        try (HikariDataSource dataSource = new HikariDataSource(config);
```

```
            Connection connection = dataSource.getConnection();
            Statement statement = connection.createStatement()) {
            statement.execute("INSERT INTO t_order (user_id, order_type, address_
id, status) VALUES (1, 1, 1, 'INSERT_TEST')");
            statement.executeQuery("SELECT * FROM t_order");
            statement.execute("alter table t_order delete where user_id=1");
        }
    }
}
```

使用限制

SQL 限制

ShardingSphere JDBC DataSource 尚不支持执行 ClickHouse 的 `create table`, `truncate table` 和 `drop table` 语句。用户应考虑为 ShardingSphere 提交包含单元测试的 PR。

分布式序列限制

ClickHouse 自身的，对应分布式序列功能的列类型是 `UUID`，`UUID` 在 ClickHouse JDBC Driver 中接收为 `java.util.UUID`, 参考 https://github.com/ClickHouse/ClickHouse/issues/56228 。而 ShardingSphere 的 `SNOWFLAKE` 的分布式序列 SPI 实现对应的列类型是 `UInt64`，在 ShardingSphere JDBC Driver 中接收为 `java.lang.Long`。

当为 ShardingSphere 配置连接至 ClickHouse 时，若同时配置了 ShardingSphere 使用 `SNOWFLAKE` 的分布式序列 SPI 实现，ShardingSphere 的分布式序列功能使用的 ClickHouse 真实数据库中的列类型不应该被设置为 `UUID`。

由于 `com.clickhouse:clickhouse-jdbc:0.6.3:http` Maven 模块的 `com.clickhouse.jdbc.ClickHouseConnection#prepareStatement(String, int)` 故意在 `autoGeneratedKeys` 为 `java.sql.Statement.RETURN_GENERATED_KEYS` 时抛出异常，以阻止 ShardingSphere 正常代理 `com.clickhouse.jdbc.internal.ClickHouseConnectionImpl`，因此如果用户需要从 JDBC 业务代码获取 ShardingSphere 生成的分布式序列，需要将 `autoGeneratedKeys` 置为 `java.sql.Statement.NO_GENERATED_KEYS`。

一个可能的示例如下，

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.*;
public class ExampleTest {
    long test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml");
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        try (HikariDataSource dataSource = new HikariDataSource(config);
```

```
            Connection connection = dataSource.getConnection();
            PreparedStatement preparedStatement = connection.prepareStatement(
                    "INSERT INTO t_order (user_id, order_type, address_id, status)
VALUES (1, 1, 1, 'INSERT_TEST')",
                    Statement.NO_GENERATED_KEYS
            )) {
            preparedStatement.executeUpdate();
            try (ResultSet resultSet = preparedStatement.getGeneratedKeys()) {
                if (resultSet.next()) {
                    return resultSet.getLong(1);
                }
                throw new RuntimeException();
            }
        }
    }
}
```

事务限制

ClickHouse 不支持 ShardingSphere 集成级别的本地事务，XA 事务或 Seata 的 AT 模式事务，更多讨论位于 https://github.com/ClickHouse/clickhouse-docs/issues/2300 。

这与 https://clickhouse.com/docs/en/guides/developer/transactional 为 ClickHouse 提供的 `Transactions, Commit, and Rollback` 功能无关,仅与 `com.clickhouse.jdbc.ConnectionImpl` 未实现 `java.sql.Connection#rollback()` 有关。参考 https://github.com/ClickHouse/clickhouse-java/issues/2023 。

嵌入式 ClickHouse 限制

嵌入式 ClickHouse chDB 尚未发布 Java 客户端，ShardingSphere 不针对 SNAPSHOT 版本的 https://github.com/chdb-io/chdb-java 做集成测试。参考 https://github.com/chdb-io/chdb/issues/243 。

Firebird

背景信息

ShardingSphere 默认情况下不提供对 `org.firebirdsql.jdbc.FBDriver` 的 `driverClassName` 的支持。ShardingSphere 对 Firebird JDBC Driver 的支持位于可选模块中。

前提条件

要在 ShardingSphere 的配置文件为数据节点使用类似 `jdbc:firebird://localhost:3050//var/lib/firebird/data/demo_ds_0.fdb` 的 `jdbcUrl`，可能的 Maven 依赖关系如下，

```
<dependencies>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-jdbc</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-parser-sql-firebird</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.firebirdsql.jdbc</groupId>
        <artifactId>jaybird</artifactId>
        <version>5.0.6.java8</version>
    </dependency>
</dependencies>
```

配置示例

启动 Firebird

编写 Docker Compose 文件来启动 Firebird。

```
services:
  firebird:
    image: firebirdsql/firebird:5.0.1
    environment:
      FIREBIRD_ROOT_PASSWORD: masterkey
      FIREBIRD_USER: alice
      FIREBIRD_PASSWORD: masterkey
      FIREBIRD_DATABASE: mirror.fdb
      FIREBIRD_DATABASE_DEFAULT_CHARSET: UTF8
    ports:
      - "3050:3050"
```

创建业务库

通过第三方工具在 Firebird 内创建业务库。

包括 DBeaver Community 在内的第三方工具无法为 Firebird 创建 databases，下以 Maven 模块 `org.firebirdsql.jdbc:jaybird:5.0.6.java8` 的 Java API 为例，

```
import org.firebirdsql.management.FBManager;
import org.firebirdsql.management.PageSizeConstants;
class Solution {
    void test() throws Exception {
        try (FBManager fbManager = new FBManager()) {
            fbManager.setServer("localhost");
            fbManager.setUserName("alice");
            fbManager.setPassword("masterkey");
            fbManager.setFileName("/var/lib/firebird/data/mirror.fdb");
            fbManager.setPageSize(PageSizeConstants.SIZE_16K);
            fbManager.setDefaultCharacterSet("UTF8");
            fbManager.setPort(3050);
            fbManager.start();
            fbManager.createDatabase("/var/lib/firebird/data/demo_ds_0.fdb", "alice
", "masterkey");
            fbManager.createDatabase("/var/lib/firebird/data/demo_ds_1.fdb", "alice
", "masterkey");
            fbManager.createDatabase("/var/lib/firebird/data/demo_ds_2.fdb", "alice
", "masterkey");
        }
    }
}
```

在业务项目创建 ShardingSphere 数据源

在业务项目引入前提条件涉及的依赖后，在业务项目的 classpath 上编写 ShardingSphere 数据源的配置文件 `demo.yaml`，

```
dataSources:
  ds_0:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.firebirdsql.jdbc.FBDriver
    jdbcUrl: jdbc:firebird://localhost:3050//var/lib/firebird/data/demo_ds_0.fdb
    username: alice
    password: masterkey
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.firebirdsql.jdbc.FBDriver
    jdbcUrl: jdbc:firebird://localhost:3050//var/lib/firebird/data/demo_ds_1.fdb
    username: alice
    password: masterkey
```

```
  ds_2:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.firebirdsql.jdbc.FBDriver
    jdbcUrl: jdbc:firebird://localhost:3050//var/lib/firebird/data/demo_ds_2.fdb
    username: alice
    password: masterkey
rules:
- !SHARDING
    tables:
      t_order:
        actualDataNodes: <LITERAL>ds_0.t_order, ds_1.t_order, ds_2.t_order
        keyGenerateStrategy:
          column: order_id
          keyGeneratorName: snowflake
    defaultDatabaseStrategy:
      standard:
        shardingColumn: user_id
        shardingAlgorithmName: inline
    shardingAlgorithms:
      inline:
        type: INLINE
        props:
          algorithm-expression: ds_${user_id % 2}
    keyGenerators:
      snowflake:
        type: SNOWFLAKE
```

享受集成

创建 ShardingSphere 的数据源以享受集成,

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
class Solution {
    void test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml");
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        try (HikariDataSource dataSource = new HikariDataSource(config);
             Connection connection = dataSource.getConnection();
             Statement statement = connection.createStatement()) {
            statement.execute("CREATE TABLE t_order (order_id BIGINT generated by
default as identity PRIMARY KEY, order_type INT, user_id INT NOT NULL, address_id
```

```
BIGINT NOT NULL, status VARCHAR(50))");
            statement.execute("INSERT INTO t_order (user_id, order_type, address_
id, status) VALUES (1, 1, 1, 'INSERT_TEST')");
            statement.executeQuery("SELECT * FROM t_order");
            statement.execute("DELETE FROM t_order WHERE user_id=1");
            statement.execute("DROP TABLE t_order");
        }
    }
}
```

使用限制

事务限制

Firebird 支持 ShardingSphere 集成级别的本地事务，但不支持 XA 事务或 Seata 的 AT 模式事务。

对 XA 事务的讨论位于 https://github.com/apache/shardingsphere/issues/34973 。

对 Seata 的 AT 模式事务的处理，则应在 https://github.com/apache/incubator-seata 提交包含对应实现的 PR。

HiveServer2

背景信息

ShardingSphere 默认情况下不提供对 `org.apache.hive.jdbc.HiveDriver` 的 `driverClassName` 的支持。ShardingSphere 对 HiveServer2 JDBC Driver 的支持位于可选模块中。

前提条件

要在 ShardingSphere 的配置文件为数据节点使用类似 `jdbc:hive2://localhost:10000/` 的 `jdbcUrl`，可能的 Maven 依赖关系如下，

```
<dependencies>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-jdbc</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-parser-sql-hive</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
```

```
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-jdbc</artifactId>
        <version>4.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-service</artifactId>
        <version>4.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-mapreduce-client-core</artifactId>
        <version>3.3.6</version>
        <exclusions>
            <exclusion>
                <groupId>*</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

可选的解决依赖冲突的捷径

直接使用 `org.apache.hive:hive-jdbc:4.0.1` 会导致大量的依赖冲突。如果用户不希望手动解决潜在的数千行的依赖冲突，可以使用 HiveServer2 JDBC Driver 的 Thin JAR 的第三方构建。可能的配置例子如下，

```
<dependencies>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-jdbc</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-parser-sql-hive</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>io.github.linghengqian</groupId>
        <artifactId>hive-server2-jdbc-driver-thin</artifactId>
        <version>1.7.0</version>
        <exclusions>
            <exclusion>
                <groupId>com.fasterxml.woodstox</groupId>
                <artifactId>woodstox-core</artifactId>
```

```
            </exclusion>
            <exclusion>
                <groupId>org.apache.commons</groupId>
                <artifactId>commons-text</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-mapreduce-client-core</artifactId>
        <version>3.3.6</version>
        <exclusions>
            <exclusion>
                <groupId>*</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

配置示例

启动 HiveServer2

编写 Docker Compose 文件来启动 HiveServer2。

```
services:
  hive-server2:
    image: apache/hive:4.0.1
    environment:
      SERVICE_NAME: hiveserver2
    ports:
      - "10000:10000"
```

创建业务表

通过第三方工具在 HiveServer2 内创建业务库与业务表。以 DBeaver Community 为例，若使用 Ubuntu 22.04.4，可通过 Snapcraft 快速安装，

```
sudo apt update && sudo apt upgrade -y
sudo snap install dbeaver-ce
snap run dbeaver-ce
```

在 DBeaver Community 内，使用 `jdbc:hive2://localhost:10000/` 的 `jdbcUrl` 连接至 HiveServer2，`username` 和 `password` 留空。执行如下 SQL，

```sql
-- noinspection SqlNoDataSourceInspectionForFile
CREATE DATABASE demo_ds_0;
CREATE DATABASE demo_ds_1;
CREATE DATABASE demo_ds_2;
```

分别使用 `jdbc:hive2://localhost:10000/demo_ds_0`，`jdbc:hive2://localhost:10000/demo_ds_1` 和 `jdbc:hive2://localhost:10000/demo_ds_2` 的 `jdbcUrl` 连接至 HiveServer2 来执行如下 SQL，

```sql
-- noinspection SqlNoDataSourceInspectionForFile
CREATE TABLE IF NOT EXISTS t_order
(
    order_id   BIGINT NOT NULL,
    order_type INT,
    user_id    INT    NOT NULL,
    address_id BIGINT NOT NULL,
    status     string,
    PRIMARY KEY (order_id) disable novalidate
) STORED BY ICEBERG STORED AS ORC TBLPROPERTIES ('format-version' = '2');

TRUNCATE TABLE t_order;
```

在业务项目创建 ShardingSphere 数据源

在业务项目引入前提条件涉及的依赖后，在业务项目的 classpath 上编写 ShardingSphere 数据源的配置文件 `demo.yaml`，

```yaml
dataSources:
    ds_0:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: org.apache.hive.jdbc.HiveDriver
        jdbcUrl: jdbc:hive2://localhost:10000/demo_ds_0
    ds_1:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: org.apache.hive.jdbc.HiveDriver
        jdbcUrl: jdbc:hive2://localhost:10000/demo_ds_1
    ds_2:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: org.apache.hive.jdbc.HiveDriver
        jdbcUrl: jdbc:hive2://localhost:10000/demo_ds_2
rules:
- !SHARDING
    tables:
      t_order:
        actualDataNodes: <LITERAL>ds_0.t_order, ds_1.t_order, ds_2.t_order
        keyGenerateStrategy:
          column: order_id
```

```
          keyGeneratorName: snowflake
    defaultDatabaseStrategy:
      standard:
        shardingColumn: user_id
        shardingAlgorithmName: inline
    shardingAlgorithms:
      inline:
        type: INLINE
        props:
          algorithm-expression: ds_${user_id % 2}
    keyGenerators:
      snowflake:
        type: SNOWFLAKE
```

享受集成

创建 ShardingSphere 的数据源以享受集成,

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
public class ExampleUtils {
    void test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml");
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        try (HikariDataSource dataSource = new HikariDataSource(config);
             Connection connection = dataSource.getConnection();
             Statement statement = connection.createStatement()) {
            statement.execute("INSERT INTO t_order (user_id, order_type, address_
id, status) VALUES (1, 1, 1, 'INSERT_TEST')");
            statement.executeQuery("SELECT * FROM t_order");
            statement.execute("DELETE FROM t_order WHERE user_id=1");
        }
    }
}
```

外部集成

连接至开启 ZooKeeper Service Discovery 的 HiveServer2

ShardingSphere 配置文件中的 `jdbcUrl` 可配置连接至开启 ZooKeeper Service Discovery 的 HiveServer2。

引入讨论,假设存在如下 Docker Compose 文件来启动开启 ZooKeeper Service Discovery 的 HiveServer2。

```
name: test-1
services:
  zookeeper:
    image: zookeeper:3.9.3-jre-17
    ports:
      - "2181:2181"
  apache-hive-1:
    image: apache/hive:4.0.1
    depends_on:
      - zookeeper
    environment:
      SERVICE_NAME: hiveserver2
      SERVICE_OPTS: >-
        -Dhive.server2.support.dynamic.service.discovery=true
        -Dhive.zookeeper.quorum=zookeeper:2181
        -Dhive.server2.thrift.bind.host=0.0.0.0
        -Dhive.server2.thrift.port=10000
    ports:
      - "10000:10000"
```

在 DBeaver Community 内, 使用 `jdbc:hive2://127.0.0.1:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2` 的 `jdbcUrl` 连接至 HiveServer2, `username` 和 `password` 留空。执行如下 SQL,

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE DATABASE demo_ds_0;
CREATE DATABASE demo_ds_1;
CREATE DATABASE demo_ds_2;
```

分别使用 `jdbc:hive2://127.0.0.1:2181/demo_ds_0;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2`, `jdbc:hive2://127.0.0.1:2181/demo_ds_1;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2` 和 `jdbc:hive2://127.0.0.1:2181/demo_ds_2;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2` 的 `jdbcUrl` 连接至 HiveServer2 来执行如下 SQL,

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE TABLE IF NOT EXISTS t_order
(
    order_id   BIGINT NOT NULL,
    order_type INT,
```

```
    user_id    INT    NOT NULL,
    address_id BIGINT NOT NULL,
    status     string,
    PRIMARY KEY (order_id) disable novalidate
) STORED BY ICEBERG STORED AS ORC TBLPROPERTIES ('format-version' = '2');

TRUNCATE TABLE t_order;
```

在业务项目引入前提条件涉及的依赖后，在业务项目的 classpath 上编写 ShardingSphere 数据源的配置文件 `demo.yaml`，

```
dataSources:
    ds_0:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: org.apache.hive.jdbc.HiveDriver
        jdbcUrl: jdbc:hive2://127.0.0.1:2181/demo_ds_0;
serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2
    ds_1:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: org.apache.hive.jdbc.HiveDriver
        jdbcUrl: jdbc:hive2://127.0.0.1:2181/demo_ds_1;
serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2
    ds_2:
        dataSourceClassName: com.zaxxer.hikari.HikariDataSource
        driverClassName: org.apache.hive.jdbc.HiveDriver
        jdbcUrl: jdbc:hive2://127.0.0.1:2181/demo_ds_2;
serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2
rules:
- !SHARDING
    tables:
      t_order:
        actualDataNodes: <LITERAL>ds_0.t_order, ds_1.t_order, ds_2.t_order
        keyGenerateStrategy:
          column: order_id
          keyGeneratorName: snowflake
    defaultDatabaseStrategy:
      standard:
        shardingColumn: user_id
        shardingAlgorithmName: inline
    shardingAlgorithms:
      inline:
        type: INLINE
        props:
          algorithm-expression: ds_${user_id % 2}
    keyGenerators:
      snowflake:
        type: SNOWFLAKE
```

此时可正常创建 ShardingSphere 的数据源并在虚拟数据源上执行逻辑 SQL，

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
public class ExampleUtils {
    void test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml");
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        try (HikariDataSource dataSource = new HikariDataSource(config);
             Connection connection = dataSource.getConnection();
             Statement statement = connection.createStatement()) {
            statement.execute("INSERT INTO t_order (user_id, order_type, address_
id, status) VALUES (1, 1, 1, 'INSERT_TEST')");
            statement.executeQuery("SELECT * FROM t_order");
            statement.execute("DELETE FROM t_order WHERE order_id=1");
        }
    }
}
```

此时，若上文通过 Docker Compose 文件启动的，对于 `service` 名为 `apache-hive-1` 的 HiveServer2 示例被手动销毁，通过另一份如下内容的 Docker Compose 文件启动第 2 个 HiveServer2 实例，

```
name: test-2
services:
  apache-hive-2:
    image: apache/hive:4.0.1
    environment:
      SERVICE_NAME: hiveserver2
      SERVICE_OPTS: >-
        -Dhive.server2.support.dynamic.service.discovery=true
        -Dhive.zookeeper.quorum=zookeeper:2181
        -Dhive.server2.thrift.bind.host=0.0.0.0
        -Dhive.server2.thrift.port=20000
    ports:
      - "20000:20000"
    networks:
      - test-1_default
networks:
  test-1_default:
    external: true
```

在 DBeaver Community 内，使用 `jdbc:hive2://127.0.0.1:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2` 的 `jdbcUrl` 连接至 HiveServer2，`username` 和 `password` 留空。执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE DATABASE demo_ds_0;
CREATE DATABASE demo_ds_1;
CREATE DATABASE demo_ds_2;
```

分别使用 `jdbc:hive2://127.0.0.1:2181/demo_ds_0;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2`，`jdbc:hive2://127.0.0.1:2181/demo_ds_1;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2` 和 `jdbc:hive2://127.0.0.1:2181/demo_ds_2;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2` 的 `jdbcUrl` 连接至 HiveServer2 来执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE TABLE IF NOT EXISTS t_order
(
    order_id   BIGINT NOT NULL,
    order_type INT,
    user_id    INT    NOT NULL,
    address_id BIGINT NOT NULL,
    status     string,
    PRIMARY KEY (order_id) disable novalidate
) STORED BY ICEBERG STORED AS ORC TBLPROPERTIES ('format-version' = '2');

TRUNCATE TABLE t_order;
```

此时，旧的 ShardingSphere JDBC DataSource 仍可在不重新创建 JDBC DataSource 的情况下，正常切换到 `service` 名为 `apache-hive-2` 的 HiveServer2 实例执行逻辑 SQL，

```
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
public class ExampleUtils {
    void test(HikariDataSource dataSource) throws SQLException {
        try (Connection connection = dataSource.getConnection();
             Statement statement = connection.createStatement()) {
            statement.execute("INSERT INTO t_order (user_id, order_type, address_
id, status) VALUES (1, 1, 1, 'INSERT_TEST')");
            statement.executeQuery("SELECT * FROM t_order");
            statement.execute("DELETE FROM t_order WHERE order_id=1");
        }
    }
}
```

使用限制

版本限制

HiveServer2 `2.x` 和 HiveServer2 `3.x` 发行版的生命周期已经结束。参考 https://lists.apache.org/thread/0mh4hvpllzv877bkx1f9srv1c3hlbtt9 和 https://lists.apache.org/thread/mpzrv7v1hqqo4cmp0zorswnbvd7ltmbp 。ShardingSphere 仅针对 HiveServer2 `4.0.1` 进行集成测试。

HiveServer2 JDBC Driver 的 Uber JAR 限制

受 https://issues.apache.org/jira/browse/HIVE-28445 影响，用户不应该使用 `classifier` 为 `standalone` 的 `org.apache.hive:hive-jdbc:4.0.1`，以避免依赖冲突。

嵌入式 HiveServer2 限制

嵌入式 HiveServer2 不再被 Hive 社区认为是用户友好的，用户不应该尝试通过 ShardingSphere 的配置文件启动嵌入式 HiveServer2。用户总应该通过 HiveServer2 的 Docker Image `apache/hive:4.0.1` 启动 HiveServer2。参考 https://issues.apache.org/jira/browse/HIVE-28418 。

Hadoop 限制

用户仅可使用 Hadoop `3.3.6` 来作为 HiveServer2 JDBC Driver `4.0.1` 的底层 Hadoop 依赖。HiveServer2 JDBC Driver `4.0.1` 不支持 Hadoop `3.4.1`，参考 https://github.com/apache/hive/pull/5500 。

对于 HiveServer2 JDBC Driver `org.apache.hive:hive-jdbc:4.0.1` 或 `classifier` 为 `standalone` 的 `org.apache.hive:hive-jdbc:4.0.1`，实际上并不额外依赖 `org.apache.hadoop:hadoop-mapreduce-client-core:3.3.6`。

但 `org.apache.shardingsphere:shardingsphere-infra-database-hive` 的 `org.apache.shardingsphere.infra.database.hive.metadata.data.loader.HiveMetaDataLoader` 会使用 `org.apache.hadoop.hive.conf.HiveConf`，这进一步使用了 `org.apache.hadoop:hadoop-mapreduce-client-core:3.3.6` 的 `org.apache.hadoop.mapred.JobConf` 类。

ShardingSphere 仅需要使用 `org.apache.hadoop.mapred.JobConf` 类，因此排除 `org.apache.hadoop:hadoop-mapreduce-client-core:3.3.6` 的所有额外依赖是合理行为。

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>3.3.6</version>
    <exclusions>
        <exclusion>
            <groupId>*</groupId>
            <artifactId>*</artifactId>
        </exclusion>
```

```
    </exclusions>
</dependency>
```

SQL 限制

HiveServer2 并不能保证每一条 `insert` 相关的 DML SQL 都能成功执行，尽管可能没有任何异常被抛出。

ShardingSphere JDBC DataSource 尚不支持执行 HiveServer2 的 `set`，`create table`，`truncate table` 和 `drop table` 语句。用户应考虑为 ShardingSphere 提交包含单元测试的 PR。

使用 `initFile` 参数部分绕开 SQL 限制

受 https://issues.apache.org/jira/browse/HIVE-28835 影响，HiveServer2 JDBC Driver 的 `initFile` 参数仅可在 Linux 环境下使用。

以 `set` 为代表的 SQL 语句很容易在 HiveServer2 Client 级别被动态配置。即便 ShardingSphere JDBC 不支持在虚拟 DataSource 上执行 HiveServer2 的 `set` 语句，用户也可以通过 `initFile` 的 Hive Session 参数来直接为真实 DataSource 执行一系列 SQL。引入讨论，可能的 ShardingSphere 配置文件如下，

```
dataSources:
  ds_0:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.apache.hive.jdbc.HiveDriver
    jdbcUrl: jdbc:hive2://localhost:10000/demo_ds_0;initFile=/tmp/init.sql
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.apache.hive.jdbc.HiveDriver
    jdbcUrl: jdbc:hive2://localhost:10000/demo_ds_0;initFile=/tmp/init.sql
  ds_2:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.apache.hive.jdbc.HiveDriver
    jdbcUrl: jdbc:hive2://localhost:10000/demo_ds_0;initFile=/tmp/init.sql
```

`/tmp/init.sql` 的可能内容如下，

```
-- noinspection SqlNoDataSourceInspectionForFile
set metastore.compactor.initiator.on=true;
set metastore.compactor.cleaner.on=true;
set metastore.compactor.worker.threads=1;

set hive.support.concurrency=true;
set hive.exec.dynamic.partition.mode=nonstrict;
set hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
```

受 https://issues.apache.org/jira/browse/HIVE-28317 影响，`initFile` 参数仅可使用绝对路径。但 ShardingSphere JDBC Driver 存在 `placeholder-type` 参数来动态定义 YAML 属性。进一步讨论，可能的 ShardingSphere 配置文件如下，

```yaml
dataSources:
  ds_0:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.apache.hive.jdbc.HiveDriver
    jdbcUrl: $${fixture.hive.ds0.jdbc-url::}
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.apache.hive.jdbc.HiveDriver
    jdbcUrl: $${fixture.hive.ds1.jdbc-url::}
  ds_2:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: org.apache.hive.jdbc.HiveDriver
    jdbcUrl: $${fixture.hive.ds2.jdbc-url::}
```

此时使用 ShardingSphere JDBC Driver 时可以通过拼接字符串的手段传入业务项目的 classpath 上的文件的绝对路径。

```java
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import javax.sql.DataSource;
import java.nio.file.Paths;
public class ExampleUtils {
    public DataSource createDataSource() {
        HikariConfig config = new HikariConfig();
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml?placeholder-
type=system_props");
        try {
            assert null == System.getProperty("fixture.hive.ds0.jdbc-url");
            assert null == System.getProperty("fixture.hive.ds1.jdbc-url");
            assert null == System.getProperty("fixture.hive.ds2.jdbc-url");
            String absolutePath = Paths.get("src/test/resources/init.sql").
toAbsolutePath().toString();
            System.setProperty("fixture.hive.ds0.jdbc-url", "jdbc:hive2://
localhost:10000/demo_ds_0;initFile=" + absolutePath);
            System.setProperty("fixture.hive.ds1.jdbc-url", "jdbc:hive2://
localhost:10000/demo_ds_1;initFile=" + absolutePath);
            System.setProperty("fixture.hive.ds2.jdbc-url", "jdbc:hive2://
localhost:10000/demo_ds_2;initFile=" + absolutePath);
            return new HikariDataSource(config);
        } finally {
            System.clearProperty("fixture.hive.ds0.jdbc-url");
            System.clearProperty("fixture.hive.ds1.jdbc-url");
            System.clearProperty("fixture.hive.ds2.jdbc-url");
        }
    }
}
```

在 ShardingSphere 数据源上使用 DML SQL 语句的前提条件

为了能够使用 `delete` 等 DML SQL 语句，当连接到 HiveServer2 时，用户应当考虑在 ShardingSphere JDBC 中仅使用支持 ACID 的表。`apache/hive` 提供了多种事务解决方案。

第 1 种选择是使用 ACID 表，可能的建表流程如下。ACID 表使用过时的基于目录的表格式。

```
-- noinspection SqlNoDataSourceInspectionForFile
set metastore.compactor.initiator.on=true;
set metastore.compactor.cleaner.on=true;
set metastore.compactor.worker.threads=1;

set hive.support.concurrency=true;
set hive.exec.dynamic.partition.mode=nonstrict;
set hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;

create table IF NOT EXISTS t_order
(
    order_id   BIGINT NOT NULL,
    order_type INT,
    user_id    INT    NOT NULL,
    address_id BIGINT NOT NULL,
    status     VARCHAR(50),
    PRIMARY KEY (order_id) disable novalidate
) CLUSTERED BY (order_id) INTO 2 BUCKETS STORED AS ORC TBLPROPERTIES (
'transactional' = 'true');
```

第 2 种选择是使用 Iceberg 表，可能的建表流程如下。Apache Iceberg 表格式有望在未来几年取代传统的 Hive 表格式，参考 https://blog.cloudera.com/from-hive-tables-to-iceberg-tables-hassle-free/ 。

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE TABLE IF NOT EXISTS t_order
(
    order_id   BIGINT NOT NULL,
    order_type INT,
    user_id    INT    NOT NULL,
    address_id BIGINT NOT NULL,
    status     string,
    PRIMARY KEY (order_id) disable novalidate
) STORED BY ICEBERG STORED AS ORC TBLPROPERTIES ('format-version' = '2');
```

Iceberg 表格式支持的 Hive type 相对较少，为 HiveServer2 执行 SQL `set iceberg.mr.schema.auto.conversion=true;` 有助于缓解这一问题。SQL `set iceberg.mr.schema.auto.conversion=true;` 存在 https://issues.apache.org/jira/browse/HIVE-26507 涉及的弊端。

事务限制

HiveServer2 不支持 ShardingSphere 集成级别的本地事务，XA 事务或 Seata 的 AT 模式事务，更多讨论位于 https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions 。

这与 `https://iceberg.apache.org/docs/1.7.0/hive/#table-rollback` 为 HiveServer2 提供的 `Table rollback` 功能无关，仅与 `org.apache.hive.jdbc.HiveConnection` 未实现 `java.sql.Connection#rollback()` 有关。

DBeaver Community 版本限制

当用户使用 DBeaver Community 连接至 HiveServer2 时，需确保 DBeaver Community 版本大于或等于 `24.2.5`。参考 https://github.com/dbeaver/dbeaver/pull/35059 。

Presto

背景信息

ShardingSphere 默认情况下不提供对 `com.facebook.presto.jdbc.PrestoDriver` 的 `driverClassName` 的支持。ShardingSphere 对 Presto JDBC Driver 的支持位于可选模块中。

前提条件

要在 ShardingSphere 的配置文件为数据节点使用类似 `jdbc:presto://localhost:8080/iceberg/demo_ds_0` 的 `jdbcUrl`，可能的 Maven 依赖关系如下，

```
<dependencies>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-jdbc</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-parser-sql-presto</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>com.facebook.presto</groupId>
        <artifactId>presto-jdbc</artifactId>
        <version>0.292</version>
    </dependency>
</dependencies>
```

配置示例

启动 Presto

编写 Docker Compose 文件来启动 Presto。这将启动一个既为协调器又为工作节点的 Presto 节点，并为该节点配置 Iceberg 连接器。此外，此 Iceberg 连接器将使用本地文件系统目录启动 Hive Metastore Server。

```
services:
  presto:
    image: prestodb/presto:0.292
    ports:
      - "8080:8080"
    volumes:
      - ./iceberg.properties:/opt/presto-server/etc/catalog/iceberg.properties
```

同级文件夹包含文件 `iceberg.properties`，内容如下，

```
connector.name=iceberg
iceberg.catalog.type=hive
hive.metastore=file
hive.metastore.catalog.dir=file:/home/iceberg_data
```

创建业务相关的 schema 和表

通过第三方工具在 Presto 内创建业务相关的 schema 和表。以 DBeaver Community 为例，若使用 Ubuntu 22.04.5，可通过 Snapcraft 快速安装，

```
sudo apt update && sudo apt upgrade -y
sudo snap install dbeaver-ce
snap run dbeaver-ce
```

在 DBeaver Community 内，使用 `jdbc:presto://localhost:8080/iceberg` 的 `jdbcUrl`，`test` 的 `username` 连接至 Presto，`password` 留空。执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE SCHEMA iceberg.demo_ds_0;
CREATE SCHEMA iceberg.demo_ds_1;
CREATE SCHEMA iceberg.demo_ds_2;
```

分别使用 `jdbc:presto://localhost:8080/iceberg/demo_ds_0`，`jdbc:presto://localhost:8080/iceberg/demo_ds_1` 和 `jdbc:presto://localhost:8080/iceberg/demo_ds_2` 的 `jdbcUrl` 连接至 Presto 来执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE TABLE IF NOT EXISTS t_order (
    order_id BIGINT NOT NULL,
    order_type INTEGER,
```

```
    user_id INTEGER NOT NULL,
    address_id BIGINT NOT NULL,
    status VARCHAR(50)
);
truncate table t_order;
```

在业务项目创建 ShardingSphere 数据源

在业务项目引入前提条件涉及的依赖后，在业务项目的 classpath 上编写 ShardingSphere 数据源的配置文件 `demo.yaml`，

```
dataSources:
  ds_0:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.facebook.presto.jdbc.PrestoDriver
    jdbcUrl: jdbc:presto://localhost:8080/iceberg/demo_ds_0
    username: test
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.facebook.presto.jdbc.PrestoDriver
    jdbcUrl: jdbc:presto://localhost:8080/iceberg/demo_ds_1
    username: test
  ds_2:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.facebook.presto.jdbc.PrestoDriver
    jdbcUrl: jdbc:presto://localhost:8080/iceberg/demo_ds_2
    username: test
rules:
  - !SHARDING
    tables:
      t_order:
        actualDataNodes: <LITERAL>ds_0.t_order, ds_1.t_order, ds_2.t_order
        keyGenerateStrategy:
          column: order_id
          keyGeneratorName: snowflake
    defaultDatabaseStrategy:
      standard:
        shardingColumn: user_id
        shardingAlgorithmName: inline
    shardingAlgorithms:
      inline:
        type: INLINE
        props:
          algorithm-expression: ds_${user_id % 2}
    keyGenerators:
      snowflake:
        type: SNOWFLAKE
```

享受集成

创建 ShardingSphere 的数据源以享受集成,

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
public class ExampleUtils {
    void test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:shardingsphere:classpath:demo.yaml");
        config.setDriverClassName("org.apache.shardingsphere.driver.
ShardingSphereDriver");
        try (HikariDataSource dataSource = new HikariDataSource(config);
             Connection connection = dataSource.getConnection();
             Statement statement = connection.createStatement()) {
            statement.execute("INSERT INTO t_order (user_id, order_type, address_
id, status) VALUES (1, 1, 1, 'INSERT_TEST')");
            statement.executeQuery("SELECT * FROM t_order");
            statement.execute("DELETE FROM t_order WHERE user_id=1");
            statement.execute("DROP TABLE IF EXISTS t_order");
        }
    }
}
```

使用限制

SQL 限制

ShardingSphere JDBC DataSource 尚不支持执行 Presto 的 `create table` 和 `truncate table` 语句。

事务限制

Presto 不支持 ShardingSphere 集成级别的本地事务，XA 事务或 Seata 的 AT 模式事务。Presto 自身的事务支持存在问题，参考 https://github.com/prestodb/presto/issues/25204 。

连接器限制

受 https://github.com/prestodb/presto/issues/23226 影响，Presto Memory 连接器的健康检查存在已知问题，不应在 ShardingSphere 的配置文件内连接至 Presto Memory 连接器。

9.1.5 不支持项

配置

- 不支持配置多个逻辑数据库。

DataSource 接口

- 不支持 timeout 相关操作。

Connection 接口

- 不支持存储过程，函数，游标的操作；
- 不支持执行 native SQL；
- 不支持 savepoint 相关操作；
- 不支持 Schema/Catalog 的操作；
- 不支持自定义类型映射。

Statement 和 PreparedStatement 接口

- 不支持返回多结果集的语句（即存储过程，非 SELECT 多条数据）；
- 不支持国际化字符的操作。

ResultSet 接口

- 不支持对于结果集指针位置判断；
- 不支持通过非 next 方法改变结果指针位置；
- 不支持修改结果集内容；
- 不支持获取国际化字符；
- 不支持获取 Array。

JDBC 4.1

- 不支持 JDBC 4.1 接口新功能。

查询所有未支持方法，请阅读 `org.apache.shardingsphere.driver.jdbc.unsupported` 包。

9.1.6 可观察性

Agent

源码编译

从 Github 下载 Apache ShardingSphere 源码，对源码进行编译，操作命令如下。

```
git clone --depth 1 https://github.com/apache/shardingsphere.git
cd shardingsphere
mvn clean install -DskipITs -DskipTests -Prelease,default-dep
```

Agent 制品 `distribution/agent/target/apache-shardingsphere-${latest.release.version}-shardingsphere-agent-bin.tar.gz`

目录说明

创建 agent 目录，解压 agent 二进制包到 agent 目录。

```
mkdir agent
tar -zxvf apache-shardingsphere-${latest.release.version}-shardingsphere-agent-bin.
tar.gz -C agent
cd agent
tree
├── LICENSE
├── NOTICE
├── conf
│   └── agent.yaml
├── plugins
│   ├── lib
│   │   ├── shardingsphere-agent-metrics-core-${latest.release.version}.jar
│   │   └── shardingsphere-agent-plugin-core-${latest.release.version}.jar
│   ├── logging
│   │   └── shardingsphere-agent-logging-file-${latest.release.version}.jar
│   ├── metrics
│   │   └── shardingsphere-agent-metrics-prometheus-${latest.release.version}.jar
│   └── tracing
│       ├── shardingsphere-agent-tracing-opentelemetry-${latest.release.version}.
jar
└── shardingsphere-agent-${latest.release.version}.jar
```

配置说明

`conf/agent.yaml` 用于管理 agent 配置。内置插件包括 File、Prometheus、OpenTelemetry。

```
plugins:
#  logging:
#    File:
#      props:
#        level: "INFO"
#  metrics:
#    Prometheus:
#      host:  "localhost"
#      port: 9090
#      props:
#        jvm-information-collector-enabled: "true"
#  tracing:
#    OpenTelemetry:
#      props:
#        otel.service.name: "shardingsphere"
#        otel.traces.exporter: "jaeger"
#        otel.exporter.otlp.traces.endpoint: "http://localhost:14250"
#        otel.traces.sampler: "always_on"
```

插件说明

File

目前 File 插件只有构建元数据耗时日志输出，暂无其他日志输出。

Prometheus

用于暴露监控指标

- 参数说明

名称	说明
host	主机
port	端口
jvm-information-collector-enabled	是否采集 JVM 指标信息

OpenTelemetry

OpenTelemetry 可以导出 tracing 数据到 Jaeger，Zipkin。

- 参数说明

名称	说明
otel.service.name	服务名称
otel.traces.exporter	traces exporter
otel.exporter.otlp.traces.endpoint	traces endpoint
otel.traces.sampler	traces sampler

参数参考 OpenTelemetry SDK Autoconfigure

使用方式

- 1 准备好已集成 `ShardingSphere-JDBC` 的 `SpringBoot` 项目，test-project.jar
- 2 启动项目

```
java -javaagent:/agent/shardingsphere-agent-${latest.release.version}.jar -jar
test-project.jar
```

- 3 访问启动的服务
- 4 查看对应的插件是否生效

Docker

本地构建

ShardingSphere Agent 存在可用的 `Dockerfile` 用于方便分发。可执行如下命令以构建 Docker Image，

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -am -pl distribution/agent -Prelease,default-dep,docker -T1C -DskipTests
clean package
```

此后若在自定义 `Dockerfile` 中添加以下语句，这会将 ShardingSphere Agent 的目录复制到 `/shardingsphere-agent/` 。

```
COPY --from=ghcr.io/apache/shardingsphere-agent:latest /usr/agent/ /shardingsphere-
agent/
```

社区构建

自 ShardingSphere 5.5.2 开始，ShardingSphere Agent 在 https://github.com/apache/shardingsphere/pkgs/container/shardingsphere-agent 发布社区构建。此 Docker Image 不属于 ASF 分发产物之一，只是为了方便而提供。

若在自定义 `Dockerfile` 中添加以下语句，这会将 ShardingSphere Agent 的目录复制到 `/shardingsphere-agent/`。

```
COPY --from=ghcr.io/apache/shardingsphere-agent:5.5.2 /usr/agent/ /shardingsphere-agent/
```

夜间构建

ShardingSphere Agent 在 https://github.com/apache/shardingsphere/pkgs/container/shardingsphere-agent 存在夜间构建的 Docker Image。

若在自定义 `Dockerfile` 中添加以下语句，这会将 ShardingSphere Agent 的目录复制到 `/shardingsphere-agent/`。

```
COPY --from=ghcr.io/apache/shardingsphere-agent:latest /usr/agent/ /shardingsphere-agent/
```

通过 Dockerfile 使用

引入一个典型场景，

1. 假设通过如下的 Bash 命令部署了 Jaeger All in One 的 Docker Container，

```
docker network create example-net
docker run --rm -d \
  --name jaeger \
  -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 \
  --network example-net \
  jaegertracing/all-in-one:1.62.0
```

2. 假设 `./custom-agent.yaml` 包含 ShardingSphere Agent 的配置，内容可能如下，

```
plugins:
  tracing:
    OpenTelemetry:
      props:
        otel.service.name: "example"
        otel.exporter.otlp.traces.endpoint: "http://jaeger:4318"
```

3. 假设 `./target/example.jar` 是一个即将使用 ShardingSphere Agent 的 Spring Boot 的 Uber JAR，可通过类似如下的 `Dockerfile` 来为类似 `example.jar` 的 JAR 使用夜间构建的 Docker Image 中的 ShardingSphere Agent。

```
FROM ghcr.io/apache/shardingsphere-agent:latest
COPY ./target/example.jar /app.jar
COPY ./custom-agent.yaml /usr/agent/conf/agent.yaml
ENTRYPOINT ["java","-javaagent:/usr/agent/shardingsphere-agent.jar","-jar","/app.jar"]
```

如果是通过本地构建 `ghcr.io/apache/shardingsphere-agent:latest` 的 Docker Image, `Dockerfile` 可能如下,

```
FROM ghcr.io/apache/shardingsphere-agent:latest
COPY ./target/example.jar /app.jar
COPY ./custom-agent.yaml /usr/agent/conf/agent.yaml
ENTRYPOINT ["java","-javaagent:/usr/agent/shardingsphere-agent.jar","-jar","/app.jar"]
```

4. 享受它,

```
docker build -t example/gs-spring-boot-docker:latest .
docker run --network example-net example/gs-spring-boot-docker:latest
```

Metrics

指标名称	指标类型	指标描述
build_info	GAUGE	构建信息
parsed_sql_total	COUNTER	按类型（INSERT、UPDATE、DELETE、SELECT、DDL、DCL、DAL、TCL、RQL、RDL、RAL、RUL）分类的解析总数
routed_sql_total	COUNTER	按类型（INSERT、UPDATE、DELETE、SELECT）分类的路由总数
routed_result_total	COUNTER	路由结果总数 (数据源路由结果、表路由结果)
jdbc_state	GAUGE	ShardingSphere-JDBC 状态信息。0 表示正常状态；1 表示熔断状态；2 锁定状态
jdbc_meta_data_info	GAUGE	ShardingSphere-JDBC 元数据信息
jdbc_statement_execute_total	COUNTER	语句执行总数
jdbc_statement_execute_errors_total	COUNTER	语句执行错误总数
jdbc_statement_execute_latency_millis	HISTOGRAM	语句执行耗时
jdbc_transactions_total	COUNTER	事务总数，按 commit，rollback 分类

9.1.7 GraalVM Native Image

背景信息

ShardingSphere JDBC 已在 GraalVM Native Image 下完成可用性验证。

构建包含 `org.apache.shardingsphere:shardingsphere-jdbc:${shardingsphere.version}` 的 Maven 依赖的 GraalVM Native Image, 你需要借助于 GraalVM Native Build Tools。GraalVM Native Build Tools 提供了 Maven Plugin 和 Gradle Plugin 来简化 GraalVM CE 的 `native-image` 命令行工具的长篇大论的 shell 命令。

ShardingSphere JDBC 要求在如下或更高版本的 `GraalVM CE` 完成构建 GraalVM Native Image。使用者可通过 SDKMAN! 快速切换 JDK。这同理适用于 https://sdkman.io/jdks#graal , https://sdkman.io/jdks#nik 和 https://sdkman.io/jdks#mandrel 等 `GraalVM CE` 的下游发行版。

- GraalVM CE For JDK 22.0.2, 对应于 SDKMAN! 的 `22.0.2-graalce`

用户依然可以使用 SDKMAN! 上的 `21.0.2-graalce` 等旧版本的 GraalVM CE 来构建 ShardingSphere 的 GraalVM Native Image 产物。但这将导致集成部分第三方依赖时, 构建 GraalVM Native Image 失败。典型的例子来自 HiveServer2 JDBC Driver 相关的 `org.apache.hive:hive-jdbc:4.0.0`, HiveServer2 JDBC Driver 使用了 AWT 相关的类, 而 GraalVM CE 对 `java.beans.**` package 的支持仅位于 GraalVM CE For JDK22 及更高版本。

```
com.sun.beans.introspect.ClassInfo was unintentionally initialized at build time.
To see why com.sun.beans.introspect.ClassInfo got initialized use --trace-class-
initialization=com.sun.beans.introspect.ClassInfo
java.beans.Introspector was unintentionally initialized at build time. To see why
java.beans.Introspector got initialized use --trace-class-initialization=java.
beans.Introspector
```

Maven 生态

使用者需要主动使用 GraalVM Reachability Metadata 中央仓库。如下配置可供参考, 以配置项目额外的 Maven Profiles, 以 GraalVM Native Build Tools 的文档为准。

```
<project>
    <dependencies>
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>shardingsphere-jdbc</artifactId>
            <version>${shardingsphere.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.graalvm.buildtools</groupId>
```

```
                <artifactId>native-maven-plugin</artifactId>
                <version>0.10.6</version>
                <extensions>true</extensions>
                <configuration>
                    <buildArgs>
                        <buildArg>-H:+AddAllCharsets</buildArg>
                    </buildArgs>
                </configuration>
                <executions>
                    <execution>
                        <id>build-native</id>
                        <goals>
                            <goal>compile-no-fork</goal>
                        </goals>
                        <phase>package</phase>
                    </execution>
                    <execution>
                        <id>test-native</id>
                        <goals>
                            <goal>test</goal>
                        </goals>
                        <phase>test</phase>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Gradle 生态

使用者需要主动使用 GraalVM Reachability Metadata 中央仓库。如下配置可供参考，以配置项目额外的 Gradle Tasks，以 GraalVM Native Build Tools 的文档为准。由于 https://github.com/gradle/gradle/issues/17559 的限制，用户需要通过 Maven 依赖的形式引入 Metadata Repository 的 JSON 文件。参考 https://github.com/graalvm/native-build-tools/issues/572 。

```
plugins {
   id 'org.graalvm.buildtools.native' version '0.10.6'
}

dependencies {
   implementation 'org.apache.shardingsphere:shardingsphere-jdbc:${shardingsphere.
version}'
   implementation(group: 'org.graalvm.buildtools', name: 'graalvm-reachability-
metadata', version: '0.10.6', classifier: 'repository', ext: 'zip')
}
```

```
graalvmNative {
   binaries {
      main {
         buildArgs.add('-H:+AddAllCharsets')
      }
      test {
         buildArgs.add('-H:+AddAllCharsets')
      }
   }
   metadataRepository {
        enabled.set(false)
   }
}
```

对于 sbt 等不被 GraalVM Native Build Tools 支持的构建工具

此类需求需要在 https://github.com/graalvm/native-build-tools 打开额外的 issue 并提供对应构建工具的 Plugin 实现。

使用限制

1. 如下的算法类由于涉及到 https://github.com/oracle/graal/issues/5522，暂未可在 GraalVM Native Image 下使用。
   - `org.apache.shardingsphere.sharding.algorithm.sharding.inline.InlineShardingAlgorithm`
   - `org.apache.shardingsphere.sharding.algorithm.sharding.inline.ComplexInlineShardingAlgorithm`
   - `org.apache.shardingsphere.sharding.algorithm.sharding.hint.HintInlineShardingAlgorithm`

对于常规案例，使用者可通过 CLASS_BASE 算法自行模拟 GroovyShell 的行为。例如对于如下配置。

```
rules:
- !SHARDING
  defaultDatabaseStrategy:
      standard:
        shardingColumn: user_id
        shardingAlgorithmName: inline
  shardingAlgorithms:
    inline:
      type: INLINE
      props:
        algorithm-expression: ds_${user_id % 2}
        allow-range-query-with-inline-sharding: false
```

可首先定义 CLASS_BASE 的实现类。

```
package org.example.test;

import org.apache.shardingsphere.sharding.api.sharding.standard.
PreciseShardingValue;
import org.apache.shardingsphere.sharding.api.sharding.standard.RangeShardingValue;
import org.apache.shardingsphere.sharding.api.sharding.standard.
StandardShardingAlgorithm;

import java.util.Collection;

public final class TestShardingAlgorithmFixture implements
StandardShardingAlgorithm<Integer> {

    @Override
    public String doSharding(final Collection<String> availableTargetNames, final
PreciseShardingValue<Integer> shardingValue) {
        String resultDatabaseName = "ds_" + shardingValue.getValue() % 2;
        for (String each : availableTargetNames) {
            if (each.equals(resultDatabaseName)) {
                return each;
            }
        }
        return null;
    }

    @Override
    public Collection<String> doSharding(final Collection<String>
availableTargetNames, final RangeShardingValue<Integer> shardingValue) {
        throw new RuntimeException("This algorithm class does not support range
queries.");
    }
}
```

修改相关 YAML 配置如下。

```
rules:
- !SHARDING
  defaultDatabaseStrategy:
      standard:
        shardingColumn: user_id
        shardingAlgorithmName: inline
  shardingAlgorithms:
    inline:
        type: CLASS_BASED
        props:
          strategy: STANDARD
          algorithmClassName: org.example.test.TestShardingAlgorithmFixture
```

在 src/main/resources/META-INF/native-image/exmaple-test-metadata/reflect-config.json 加入如下内容即可在正常在 GraalVM Native Image 下使用。

```
[
{
  "name":"org.example.test.TestShardingAlgorithmFixture",
  "methods":[{"name":"<init>","parameterTypes":[] }]
}
]
```

2. 对于 读写分离的功能，你需要使用 行表达式 SPI 的其他实现，以在配置 logic database name，writeDataSourceName 和 readDataSourceNames 时绕开对 GroovyShell 的调用。一个可能的配置是使用 LITERAL 的 行表达式 SPI 的实现。

```
rules:
- !READWRITE_SPLITTING
   dataSourceGroups:
    <LITERAL>readwrite_ds:
      writeDataSourceName: <LITERAL>ds_0
      readDataSourceNames:
        - <LITERAL>ds_1
        - <LITERAL>ds_2
```

对于 数据分片的功能的 actualDataNodes 同理。

```
- !SHARDING
   tables:
      t_order:
         actualDataNodes: <LITERAL>ds_0.t_order_0, ds_0.t_order_1, ds_1.t_order_0,
ds_1.t_order_1
         keyGenerateStrategy:
            column: order_id
            keyGeneratorName: snowflake
```

3. 使用者依然需要在 src/main/resources/META-INF/native-image 文件夹或 src/test/resources/META-INF/native-image 文件夹配置独立文件的 GraalVM Reachability Metadata。使用者可通过 GraalVM Native Build Tools 的 GraalVM Tracing Agent 来快速采集 GraalVM Reachability Metadata。

4. 以 MS SQL Server 的 JDBC Driver 为代表的 com.microsoft.sqlserver:mssql-jdbc 等 Maven 模块会根据数据库中使用的编码动态加载不同的字符集，这是不可预测的行为。当遇到如下 Error，使用者需要添加 -H:+AddAllCharsets 的 buildArg 到 GraalVM Native Build Tools 的配置中。

```
Caused by: java.io.UnsupportedEncodingException: Codepage Cp1252 is not supported
by the Java environment.
 com.microsoft.sqlserver.jdbc.Encoding.checkSupported(SQLCollation.java:572)
 com.microsoft.sqlserver.jdbc.SQLCollation$SortOrder.getEncoding(SQLCollation.
java:473)
```

```
 com.microsoft.sqlserver.jdbc.SQLCollation.encodingFromSortId(SQLCollation.
java:501)
 [...]
```

5. 讨论在 ShardingSphere JDBC 的 GraalVM Native Image 下使用 XA 分布式事务的所需步骤，则需要引入额外的已知前提，
   - `org.apache.shardingsphere.transaction.xa.jta.datasource.swapper.DataSourceSwapper#loadXADataSource(String)` 会通过 `java.lang.Class#getDeclaredConstructors` 实例化各数据库驱动的 `javax.sql.XADataSource` 实现类。
   - 各数据库驱动的 `javax.sql.XADataSource` 实现类的全类名通过实现 `org.apache.shardingsphere.transaction.xa.jta.datasource.properties.XADataSourceDefinition` 的 SPI，来存入 ShardingSphere 的元数据。

在 GraalVM Native Image 内部，这实际上要求定义第三方依赖的 GraalVM Reachability Metadata，而 ShardingSphere 自身仅为 `com.h2database:h2` 提供对应的 GraalVM Reachability Metadata。`com.mysql:mysql-connector-j` 等其他数据库驱动的 GraalVM Reachability Metadata 应自行定义，或将对应 JSON 提交到 https://github.com/oracle/graalvm-reachability-metadata 一侧。

以 `com.mysql:mysql-connector-j:9.0.0` 的 `com.mysql.cj.jdbc.MysqlXADataSource` 类为例，这是 MySQL JDBC Driver 的 `javax.sql.XADataSource` 的实现。用户需要在自有项目的 claapath 的 `/META-INF/native-image/com.mysql/mysql-connector-j/9.0.0/` 文件夹的 `reflect-config.json` 文件内定义如下 JSON，以在 GraalVM Native Image 内部定义 `com.mysql.cj.jdbc.MysqlXADataSource` 的构造函数。

```
[
{
   "condition":{"typeReachable":"com.mysql.cj.jdbc.MysqlXADataSource"},
   "name":"com.mysql.cj.jdbc.MysqlXADataSource",
   "allPublicMethods": true,
   "methods": [{"name":"<init>","parameterTypes":[] }]
}
]
```

6. 当需要通过 ShardingSphere JDBC 使用 ClickHouse 方言时，用户需要手动引入相关的可选模块和 classifier 为 `http` 的 ClickHouse JDBC 驱动。原则上，ShardingSphere 的 GraalVM Native Image 集成不希望使用 classifier 为 `all` 的 `com.clickhouse:clickhouse-jdbc`，因为 Uber Jar 会导致采集重复的 GraalVM Reachability Metadata。可能的配置例子如下，

```
<project>
    <dependencies>
      <dependency>
         <groupId>org.apache.shardingsphere</groupId>
         <artifactId>shardingsphere-jdbc</artifactId>
         <version>${shardingsphere.version}</version>
      </dependency>
       <dependency>
```

```
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-parser-sql-clickhouse</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
     <dependency>
        <groupId>com.clickhouse</groupId>
        <artifactId>clickhouse-jdbc</artifactId>
        <version>0.6.3</version>
        <classifier>http</classifier>
     </dependency>
  </dependencies>
</project>
```

7. 受 https://github.com/grpc/grpc-java/issues/10601 影响，用户如果在项目中引入了 `org.apache.hive:hive-jdbc`，则需要在项目的 classpath 的 `META-INF/native-image/io.grpc/grpc-netty-shaded` 文件夹下创建包含如下内容的文件 `native-image.properties`，

```
Args=--initialize-at-run-time=\
    io.grpc.netty.shaded.io.netty.channel.ChannelHandlerMask,\
    io.grpc.netty.shaded.io.netty.channel.nio.AbstractNioChannel,\
    io.grpc.netty.shaded.io.netty.channel.socket.nio.SelectorProviderUtil,\
    io.grpc.netty.shaded.io.netty.util.concurrent.DefaultPromise,\
    io.grpc.netty.shaded.io.netty.util.internal.MacAddressUtil,\
    io.grpc.netty.shaded.io.netty.util.internal.SystemPropertyUtil,\
    io.grpc.netty.shaded.io.netty.util.NetUtilInitializations,\
    io.grpc.netty.shaded.io.netty.channel.AbstractChannel,\
    io.grpc.netty.shaded.io.netty.util.NetUtil,\
    io.grpc.netty.shaded.io.netty.util.internal.PlatformDependent,\
    io.grpc.netty.shaded.io.netty.util.internal.PlatformDependent0,\
    io.grpc.netty.shaded.io.netty.channel.DefaultChannelPipeline,\
    io.grpc.netty.shaded.io.netty.channel.DefaultChannelId,\
    io.grpc.netty.shaded.io.netty.util.ResourceLeakDetector,\
    io.grpc.netty.shaded.io.netty.channel.AbstractChannelHandlerContext,\
    io.grpc.netty.shaded.io.netty.channel.ChannelOutboundBuffer,\
    io.grpc.netty.shaded.io.netty.util.internal.InternalThreadLocalMap,\
    io.grpc.netty.shaded.io.netty.util.internal.CleanerJava9,\
    io.grpc.netty.shaded.io.netty.util.internal.StringUtil,\
    io.grpc.netty.shaded.io.netty.util.internal.CleanerJava6,\
    io.grpc.netty.shaded.io.netty.buffer.ByteBufUtil$HexUtil,\
    io.grpc.netty.shaded.io.netty.buffer.AbstractByteBufAllocator,\
    io.grpc.netty.shaded.io.netty.util.concurrent.FastThreadLocalThread,\
    io.grpc.netty.shaded.io.netty.buffer.PoolArena,\
    io.grpc.netty.shaded.io.netty.buffer.EmptyByteBuf,\
    io.grpc.netty.shaded.io.netty.buffer.PoolThreadCache,\
    io.grpc.netty.shaded.io.netty.util.AttributeKey
```

ShardingSphere 的单元测试仅使用 Maven 模块 `io.github.linghengqian:hive-server2-jdbc-driver-thin` 来在 GraalVM Native Image 下验证可用性。

8. 由于 https://github.com/oracle/graal/issues/7979 的影响，对应 `com.oracle.database.jdbc:ojdbc8` Maven 模块的 Oracle JDBC Driver 无法在 GraalVM Native Image 下使用。
9. 由于 https://github.com/apache/doris/issues/9426 的影响，当通过 Shardinghere JDBC 连接至 Apache Doris FE，用户需自行提供 `apache/doris` 集成模块相关的 GraalVM Reachability Metadata。

开发和测试

本文旨在向潜在贡献者,介绍如何参与开发和贡献与 ShardingSphere 相关的 GraalVM Reachability Metadata。

背景信息

当贡献者发现缺少与 ShardingSphere 无关的第三方库的 GraalVM Reachability Metadata 时，最优解是在 https://github.com/oracle/graalvm-reachability-metadata 打开新的 issue，并提交包含依赖的，第三方库缺失的 GraalVM Reachability Metadata 的 PR。

为了简化流程,ShardingSphere 在 Maven 模块 `shardingsphere-infra-reachability-metadata` 主动托管了部分第三方库的 GraalVM Reachability Metadata。

ShardingSphere 对在 GraalVM Native Image 下的可用性的验证，是通过 GraalVM Native Build Tools 的 Maven Plugin 子项目来完成的。通过在 JVM 下运行单元测试，为单元测试打上 Junit 标签，此后构建为 GraalVM Native Image 进行 nativeTest 来测试在 GraalVM Native Image 下的单元测试覆盖率。

ShardingSphere 定义了，1. `shardingsphere-test-native` 的 Maven Module，用于为 nativeTest 提供小型的单元测试子集。此单元测试子集避免了使用 nativeTest 下无法使用的第三方库。2. `nativeTestInShardingSphere` 的 Maven Profile，用于为 `shardingsphere-test-native` 模块编译单元测试所需的 GraalVM Native Image 和执行 nativeTest。3. `generateMetadata` 的 Maven Profile, 用于在 GraalVM JIT Compiler 下携带 GraalVM Tracing Agent 执行单元测试, 以采集初步的 GraalVM Reachability Metadata。

前提条件

贡献者必须在设备安装,

1. GraalVM CE 22.0.2，或与 GraalVM CE 22.0.2 兼容的 GraalVM 下游发行版。以 GraalVM Native Image 为准。
2. 编译 GraalVM Native Image 所需要的本地工具链。以 https://www.graalvm.org/latest/reference-manual/native-image/#prerequisites 为准。
3. 可运行 Linux Containers 的 Docker Engine，或与 testcontainers-java 兼容的 Container Runtime。以 https://java.testcontainers.org/supported_docker_environment/ 为准。

本文不讨论 `LLVM Backend for Native Image`。下文分别讨论在 Ubuntu，Windows 与 Windows Server 下可能的所需操作。

Ubuntu

假设贡献者处于新的 Ubuntu 22.04.5 LTS 实例下，且已配置 git。

可在 bash 通过如下命令利用 SDKMAN！安装 GraalVM CE。

```
sudo apt install unzip zip -y
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
sdk install java 22.0.2-graalce
sdk use java 22.0.2-graalce
```

可在 bash 通过如下命令安装编译 GraalVM Native Image 所需要的本地工具链。

```
sudo apt-get install build-essential zlib1g-dev -y
```

可在 bash 通过如下命令安装 Rootful 模式的 Docker Engine。本文不讨论更改 `/etc/docker/daemon.json` 的默认 logging driver。

```
sudo apt update && sudo apt upgrade -y
sudo apt-get remove docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd runc
cd /tmp/
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin -y
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

Windows

假设贡献者处于新的 Windows 11 Home 24H2 实例下，且已安装和配置 `git-for-windows/git` 和 `PowerShell/PowerShell`。

可在 Powershell 7 通过如下命令利用 `version-fox/vfox` 安装 GraalVM CE。

```
winget install version-fox.vfox
if (-not (Test-Path -Path $PROFILE)) { New-Item -Type File -Path $PROFILE -Force };
Add-Content -Path $PROFILE -Value 'Invoke-Expression "$(vfox activate pwsh)"'
# 此时需要打开新的 Powershell 7 终端
vfox add java
vfox install java@22.0.2-graalce
vfox use --global java@22.0.2-graalce
```

当 Windows 弹出窗口，要求允许类似 `C:\users\shard\.version-fox\cache\java\v-22.0.2-graalce\java-22.0.2-graalce\bin\java.exe` 路径的应用通过 Windows 防火墙时，应当批准。背景参考 https://support.microsoft.com/en-us/windows/risks-of-allowing-apps-through-windows-firewall-654559af-3f54-3dcf-349f-71ccd90bcc5c 。

可在 Powershell 7 通过如下命令安装编译 GraalVM Native Image 所需要的本地工具链。**特定情况下，开发者可能需要为 Visual Studio 的使用购买许可证。**

```
winget install --id Microsoft.VisualStudio.2022.Community
```

打开 `Visual Studio Installer` 以修改 `Visual Studio Community 2022` 的工作负荷，勾选 桌面应用与移动应用的 使用 `C++` 的桌面开发后点击修改。

可在 Powershell 7 通过如下命令启用 WSL2 并设置 `Ubuntu WSL` 为默认 Linux 发行版。

```
wsl --install
```

完成 WSL2 的启用后，在 `https://rancherdesktop.io/` 下载和安装 `rancher-sandbox/rancher-desktop`，并设置使用 `dockerd(moby)` 的 `Container Engine`。本文不讨论更改 Linux 发行版 `rancher-desktop` 的 `/etc/docker/daemon.json` 的默认 logging driver。

Windows Server

对于通常的 Windows Server 2025 实例，操作等同于 Windows 11 Home 24H2。但 `windows-latest` 的 Github Actions Runner 实例无法运行 Linux Containers，因此 ShardingSphere 不为 Windows 设置 nativeTest 的 CI。

处理单元测试

在 GraalVM JIT Compiler 下执行单元测试

若仅需在 GraalVM JIT Compiler 下，即在避开编译 GraalVM Native Image 的情况下执行 nativeTest 相关的单元测试，可执行如下命令。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -PgenerateMetadata -e -T 1C clean test
```

在 GraalVM Native Image 下执行单元测试

可通过如下命令为 `shardingsphere-test-native` 子模块编译单元测试所需的 GraalVM Native Image 和执行 nativeTest。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -PnativeTestInShardingSphere -e -T 1C clean test
```

当 Windows 弹出窗口，要求允许类似 `C:\users\shard\shardingsphere\test\native\target\native-tests.exe.exe` 路径的应用通过 Windows 防火墙时，应当批准。背景参考 https://support.microsoft.com/en-us/windows/risks-of-allowing-apps-through-windows-firewall-654559af-3f54-3dcf-349f-71ccd90bcc5c 。

生成和更改 GraalVM Reachability Metadata

如果 nativeTest 执行失败，应为单元测试生成初步的 GraalVM Reachability Metadata，并手动调整 `shardingsphere-infra-reachability-metadata` 子模块的 classpath 中，`META-INF/native-image/org.apache.shardingsphere/shardingsphere-infra-reachability-metadata/` 文件夹下的内容，以修复 nativeTest。如有需要，可使用 `org.junit.jupiter.api.condition.DisabledInNativeImage` 注解或 `org.graalvm.nativeimage.imagecode` 的 System Property，以屏蔽部分单元测试在 GraalVM Native Image 下运行。

`generateMetadata` 的 Maven Profile 将在 `shardingsphere-infra-reachability-metadata` 子模块的 classpath 中，`META-INF/native-image/org.apache.shardingsphere/generated-reachability-metadata/` 文件夹下，生成或覆盖已有的 GraalVM Reachability Metadata 文件。可通过如下命令简单处理此流程。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -PgenerateMetadata -e -T 1C clean test native:metadata-copy
```

贡献者仍可能需要手动调整具体的 JSON 条目，并适时调整 Maven Profile 和 GraalVM Tracing Agent 的 Filter 链。针对 `shardingsphere-infra-reachability-metadata` 子模块，手动增删改动的 JSON 条目应位于 `META-INF/native-image/org.apache.shardingsphere/shardingsphere-infra-reachability-metadata/` 文件夹下，而 `META-INF/native-image/org.apache.shardingsphere/generated-reachability-metadata/` 中的条目仅应由 `generateMetadata` 的 Maven Profile 生成。

对于测试类和测试文件独立使用的 GraalVM Reachability Metadata，贡献者应该放置到 `shardingsphere-test-native` 子模块的 classpath 的 `META-INF/native-image/shardingsphere-test-native-test-metadata/` 下。

已知限制

`resource-config.json` 限制

受 https://github.com/apache/shardingsphere/issues/33206 影响，开发者执行 `./mvnw -PgenerateMetadata -T 1C -e clean test native:metadata-copy` 后，`infra/reachability-metadata/src/main/resources/META-INF/native-image/org.apache.shardingsphere/generated-reachability-metadata/resource-config.json` 会生成不必要的包含绝对路径的 JSON 条目，

对于 Ubuntu，类似如下，

```
{
   "resources":{
      "includes":[{
         "condition":{"typeReachable":"org.apache.shardingsphere.proxy.backend.
config.ProxyConfigurationLoader"},
         "pattern":"\\Qhome/runner/work/shardingsphere/shardingsphere/test/native/
src/test/resources/test-native/yaml/proxy/databases/postgresql//global.yaml\\E"
      }, {
         "condition":{"typeReachable":"org.apache.shardingsphere.proxy.backend.
config.ProxyConfigurationLoader"},
         "pattern":"\\Qhome/runner/work/shardingsphere/shardingsphere/test/native/
src/test/resources/test-native/yaml/proxy/databases/postgresql/\\E"
      }, {
         "condition":{"typeReachable":"org.apache.shardingsphere.proxy.backend.
config.ProxyConfigurationLoader"},
         "pattern":"\\Qhome/runner/work/shardingsphere/shardingsphere/test/native/
src/test/resources/test-native/yaml/proxy/features/sharding//global.yaml\\E"
      }, {
         "condition":{"typeReachable":"org.apache.shardingsphere.proxy.backend.
config.ProxyConfigurationLoader"},
```

```
        "pattern":"\\Qhome/runner/work/shardingsphere/shardingsphere/test/native/
src/test/resources/test-native/yaml/proxy/features/sharding/\\E"
      }]},
   "bundles":[]
}
```

需要为 ShardingSphere 提交 PR 的贡献者应始终手动删除这些包含绝对路径的 JSON 条目，并等待 https://github.com/oracle/graal/issues/8417 被解决。

单元测试库限制

对于 `shardingsphere-test-native` 的 Maven Module，应避免使用 `io.kotest:kotest-runner-junit5-jvm:5.5.4` 等在 Junit 的 `test listener mode` 下存在 `failed to discover tests` 问题的测试库。

由于 Mockito Inline 无法在 GraalVM Native Image 下运行，应避免在此 Maven 模块的单元测试使用 Mockito。

对于 testcontainers,对 `org.testcontainers.utility.MountableFile#forClasspathResource(String)` 的使用应更改为 `org.testcontainers.utility.MountableFile#forHostPath(java.nio.file.Path)`，以避开 https://github.com/testcontainers/testcontainers-java/issues/7954 的影响。举例如下，

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.utility.MountableFile;
class SolutionTest {
    @Test
    void test() {
        try (PostgreSQLContainer<?> container = new PostgreSQLContainer<>(
"postgres:17.5-bookworm").withCopyFileToContainer(
                MountableFile.forClasspathResource("test-native/sh/postgres.sh"),
                "/docker-entrypoint-initdb.d/postgres.sh")) {
            container.start();
        }
    }
}
```

应更改为，

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.utility.MountableFile;
import java.nio.file.Paths;
class SolutionTest {
    @Test
    void test() {
        try (PostgreSQLContainer<?> container = new PostgreSQLContainer<>(
```

```
"postgres:17.5-bookworm").withCopyFileToContainer(
                MountableFile.forHostPath(Paths.get("src/test/resources/test-
native/sh/postgres.sh").toAbsolutePath()),
                "/docker-entrypoint-initdb.d/postgres.sh")) {
            container.start();
        }
    }
}
```

单元测试的已知问题

受 https://github.com/apache/shardingsphere/issues/35052 影响，`org.apache.shardingsphere.test.natived.jdbc.modes.cluster.EtcdTest` 的单元测试无法在通过 Windows 11 Home 24H2 编译的 GraalVM Native Image 下运行。

`org.apache.shardingsphere.test.natived.proxy.transactions.base.SeataTest` 已被禁用，因为在 Github Actions Runner 执行此单元测试将导致其他单元测试出现 JDBC 连接泄露。

9.2 ShardingSphere-Proxy

配置是 ShardingSphere-Proxy 中唯一与开发者交互的模块,通过它可以快速清晰的理解 ShardingSphere-Proxy 所提供的功能。

本章节是 ShardingSphere-Proxy 的配置参考手册，需要时可当做字典查阅。

ShardingSphere-Proxy 提供基于 YAML 的配置方式，并使用 DistSQL 进行交互。通过配置，应用开发者可以灵活的使用数据分片、读写分离、数据加密、影子库等功能，并且能够叠加使用。

规则配置部分与 ShardingSphere-JDBC 的 YAML 配置完全一致。DistSQL 与 YAML 配置能够相互取代。

9.2.1 启动手册

本章节将介绍 ShardingSphere-Proxy 相关部署和启动等相关操作。

使用二进制发布包

背景信息

本节主要介绍如何通过二进制发布包启动 ShardingSphere-Proxy。

前提条件

使用二进制发布包启动 Proxy，需要环境具备 Java JRE 8 或更高版本。

操作步骤

1. 获取 ShardingSphere-Proxy 二进制发布包

在下载页面获取。

2. 配置 `conf/global.yaml`

ShardingSphere-Proxy 运行模式在 `global.yaml` 中配置，配置格式与 ShardingSphere-JDBC 一致，请参考模式配置。

其他配置项请参考：* 权限配置 * 属性配置

3. 配置 `conf/database-*.yaml`

修改 `conf` 目录下以 `database-` 前缀开头的文件，如：`conf/database-sharding.yaml` 文件，进行分片规则、读写分离规则配置。配置方式请参考配置手册。`database-*.yaml` 文件的 `*` 部分可以任意命名。ShardingSphere-Proxy 支持配置多个逻辑数据源，每个以 `database-` 前缀命名的 YAML 配置文件，即为一个逻辑数据源。

4. （可选）引入数据库驱动

如果后端连接 PostgreSQL 或 openGauss 数据库，不需要引入额外依赖。

如果后端连接 MySQL 数据库，请下载 mysql-connector-java-5.1.49.jar 或者 mysql-connector-java-8.0.11.jar，并将其放入 `ext-lib` 目录。

5. （可选）引入集群模式所需依赖

ShardingSphere-Proxy 默认集成 ZooKeeper Curator 客户端，集群模式使用 ZooKeeper 无须引入其他依赖。

如果集群模式使用 Etcd，需要将 Etcd 依赖的 vertx-grpc 4.5.1 和 vertx-core 4.5.1 复制至目录 `ext-lib`。

6. （可选）引入分布式事务所需依赖

与 ShardingSphere-JDBC 使用方式相同。具体可参考分布式事务。

7. （可选）引入自定义算法

当用户需要使用自定义的算法类时，可通过以下方式配置使用自定义算法，以分片为例：

```
1. 实现 `ShardingAlgorithm` 接口定义的算法实现类。
2. 在项目 `resources` 目录下创建 `META-INF/services` 目录。
3. 在 `META-INF/services` 目录下新建文件 `org.apache.shardingsphere.sharding.spi.
ShardingAlgorithm`
4. 将实现类的全限定类名写入至文件 `org.apache.shardingsphere.sharding.spi.
ShardingAlgorithm`
5. 将上述 Java 文件打包成 jar 包。
6. 将上述 jar 包拷贝至 `ext-lib` 目录。
7. 将上述自定义算法实现类的 Java 文件引用配置在 YAML 文件中，具体可参考 [配置规则](https://
```

```
shardingsphere.apache.org/document/current/cn/user-manual/shardingsphere-proxy/
yaml-config/)。
```

8. 启动 ShardingSphere-Proxy

Linux/macOS 操作系统请运行 `bin/start.sh`，Windows 操作系统请运行 `bin/start.bat` 启动 ShardingSphere-Proxy。默认监听端口 `3307`，默认配置目录为 Proxy 内的 `conf` 目录。启动脚本可以指定监听端口、配置文件所在目录，命令如下：

```
bin/start.sh [port] [/path/to/conf]
```

9. 使用客户端连接 ShardingSphere-Proxy

执行 MySQL / PostgreSQL / openGauss 的客户端命令直接操作 ShardingSphere-Proxy 即可。

使用 MySQL 客户端连接 ShardingSphere-Proxy：

```
mysql -h${proxy_host} -P${proxy_port} -u${proxy_username} -p${proxy_password}
```

使用 PostgreSQL 客户端连接 ShardingSphere-Proxy：

```
psql -h ${proxy_host} -p ${proxy_port} -U ${proxy_username}
```

使用 openGauss 客户端连接 ShardingSphere-Proxy：

```
gsql -r -h ${proxy_host} -p ${proxy_port} -U ${proxy_username} -W ${proxy_password}
```

使用 Docker

背景信息

本节主要介绍如何通过 Docker 启动 ShardingSphere-Proxy。

注意事项

使用 Docker 启动 ShardingSphere-Proxy 无须额外依赖。

操作步骤

1. 获取 Docker 镜像

- 方式一（推荐）：从 DockerHub 获取

```
docker pull apache/shardingsphere-proxy
```

- 方式二：获取 master 分支最新镜像：https://github.com/apache/shardingsphere/pkgs/container/shardingsphere-proxy
- 方式三：自行构建镜像

```
git clone https://github.com/apache/shardingsphere
./mvnw clean install
cd shardingsphere-distribution/shardingsphere-proxy-distribution
./mvnw clean package -Prelease,default-dep,docker
```

如果遇到以下问题，请确保 Docker daemon 进程已经运行。

```
I/O exception (java.io.IOException) caught when processing request to {}->unix://
localhost:80: Connection refused?
```

2. 配置 `conf/global.yaml` 和 `conf/database-*.yaml`

可以从 Docker 容器中获取配置文件模板，拷贝到宿主机任意目录中：

```
docker run -d --name tmp --entrypoint=bash apache/shardingsphere-proxy
docker cp tmp:/opt/shardingsphere-proxy/conf /host/path/to/conf
docker rm tmp
```

由于容器内的网络环境可能与宿主机的网络环境有差异，如果启动时报无法连接到数据库错误等错误，请确保 `conf/database-*.yaml` 配置文件中指定的数据库的 IP 可以被 Docker 容器内部访问到。

具体配置请参考 ShardingSphere-Proxy 启动手册 - 使用二进制发布包。

3. （可选）引入第三方依赖或自定义算法

如果存在以下任意需求：* ShardingSphere-Proxy 后端使用 MySQL 数据库；* 使用自定义算法；* 使用 Etcd 作为集群模式的注册中心。

请在宿主机中任意位置创建 `ext-lib` 目录，并参考 ShardingSphere-Proxy 启动手册 - 使用二进制发布包中的对应步骤。

4. 启动 ShardingSphere-Proxy 容器

将宿主机中的 `conf` 与 `ext-lib` 目录挂载到容器中，启动容器：

```
docker run -d \
    -v /host/path/to/conf:/opt/shardingsphere-proxy/conf \
    -v /host/path/to/ext-lib:/opt/shardingsphere-proxy/ext-lib \
    -e PORT=3308 -p13308:3308 apache/shardingsphere-proxy:latest
```

其中，`ext-lib` 非必需，用户可按需挂载。ShardingSphere-Proxy 默认端口 `3307`，可以通过环境变量 `-e PORT` 指定。自定义 JVM 相关参数可通过环境变量 `JVM_OPTS` 设置。

说明：

支持设置 CGROUP_MEM_OPTS 环境变量: 用于在容器环境中设置相关内存参数，脚本中的默认值为：

```
-XX:InitialRAMPercentage=80.0 -XX:MaxRAMPercentage=80.0 -XX:MinRAMPercentage=80.0
```

5. 使用客户端连接 ShardingSphere-Proxy

请参考 ShardingSphere-Proxy 启动手册 - 使用二进制发布包。

构建 GraalVM Native Image(Alpha)

背景信息

本节主要介绍如何通过 `GraalVM CE` 的 `native-image` 命令行工具构建 ShardingSphere Proxy 的 `GraalVM Native Image`，以及包含此 `GraalVM Native Image` 的 `Docker Image`。ShardingSphere Proxy 的 `GraalVM Native Image` 在本文即指代 ShardingSphere Proxy Native。

本节内容涉及的所有 Docker Image 均不通过 https://downloads.apache.org，https://repository.apache.org 等 ASF 官方渠道进行分发。Docker Image 仅在 `GitHub Packages`，`Docker Hub` 等下游渠道提供以方便使用。

ShardingSphere Proxy Native 可执行 DistSQL，这意味着实际上不需要任何定义逻辑数据库的 YAML 文件。默认情况下，ShardingSphere Proxy Native 中仅包含，

1. 与 ShardingSphere Proxy 默认配置一致的，一系列 JAR 的编译后产物
2. ShardingSphere 维护的自有及部分第三方依赖的 GraalVM Reachability Metadata

本节内容假定处于以下的系统环境之一，

1. Linux（amd64，aarch64）
2. MacOS（amd64，aarch64/M1）
3. Windows（amd64）

本节内容依然受到 ShardingSphere JDBC 一侧的 GraalVM Native Image 的已记录内容的限制。

如果用户需要在 ShardingSphere Proxy Native 中使用第三方 JAR，或者使用 UPX 压缩编译后的 GraalVM Native Image，则需要修改 Maven 模块 `org.apache.shardingsphere:shardingsphere-proxy-native-distribution` 的源码。参考下文的从源码构建一节。

若不需要修改 ShardingSphere Proxy Native 的默认配置，开发者可从通过夜间构建的 `Docker Image` 使用一节开始处理。

通过夜间构建的 Docker Image 使用

包含 ShardingSphere Proxy Native 的 Docker Image 在 https://github.com/apache/shardingsphere/pkgs/container/shardingsphere-proxy-native 存在每夜构建。ShardingSphere Proxy Native 的默认端口为 `3307`，配置文件从 `/opt/shardingsphere-proxy/conf` 加载。

夜间构建的 Docker Image 存在多种形态的 GraalVM Native Image 的变体 Docker Image Tag。

动态链接的 GraalVM Native Image

假设存在包含 `global.yaml` 的 `conf` 文件夹为 `./custom/conf`，开发者可通过如下的 Docker Compose 文件对 动态链接的 `GraalVM Native Image` 形态的 ShardingSphere Proxy Native 进行测试。

```
services:
  apache-shardingsphere-proxy-native:
    image: ghcr.io/apache/shardingsphere-proxy-
native:da826af47804dae79b1ba5717af86792726745fd
    volumes:
      - ./custom/conf:/opt/shardingsphere-proxy/conf
    ports:
      - "3307:3307"
```

作为补充，`ghcr.io/apache/shardingsphere-proxy-native:latest` 的 Docker Image Tag 将指向 动态链接的 `GraalVM Native Image`。

大部分静态链接的 GraalVM Native Image

此节内容仅限于支持运行 `linux/amd64` OS/Arch Containers 的 Container Runtime 使用。

假设存在包含 `global.yaml` 的 `conf` 文件夹为 `./custom/conf`，开发者可通过如下的 Docker Compose 文件对 大部分静态链接的 `GraalVM Native Image` 形态的 ShardingSphere Proxy Native 进行测试。只需为特定的，动态链接的 `GraalVM Native Image` 对应的 Docker Image Tag 加上 `-mostly` 后缀。

```
services:
  apache-shardingsphere-proxy-native:
    image: ghcr.io/apache/shardingsphere-proxy-
native:da826af47804dae79b1ba5717af86792726745fd-mostly
    volumes:
      - ./custom/conf:/opt/shardingsphere-proxy/conf
    ports:
      - "3307:3307"
```

完全静态链接的 GraalVM Native Image

此节内容仅限于支持运行 `linux/amd64` OS/Arch Containers 的 Container Runtime 使用。

假设存在包含 `global.yaml` 的 `conf` 文件夹为 `./custom/conf`，开发者可通过如下的 Docker Compose 文件对 完全静态链接的 `GraalVM Native Image` 形态的 ShardingSphere Proxy Native 进行测试。只需为特定的，动态链接的 `GraalVM Native Image` 对应的 Docker Image Tag 加上 `-static` 后缀。

```
services:
  apache-shardingsphere-proxy-native:
    image: ghcr.io/apache/shardingsphere-proxy-
```

```
native:da826af47804dae79b1ba5717af86792726745fd-static
    volumes:
      - ./custom/conf:/opt/shardingsphere-proxy/conf
    ports:
      - "3307:3307"
```

从源码构建

若从源码构建，开发者有 2 种选择，

1. 在不安装本地工具链的情况下，构建包含 ShardingSphere Proxy Native 产物的 `Linux Docker Image`
2. 在安装本地工具链的情况下，构建包含 ShardingSphere Proxy Native 产物。对于 Windows，可通过此途径创建`.exe` 形态的 GraalVM Native Image

使用存在自定义 SPI 实现的 JAR 或第三方依赖的 JAR

开发者可能需要使用存在自定义 SPI 实现的 JAR 或第三方依赖的 JAR。可在从源码构建前，修改 `distribution/proxy-native/pom.xml` 文件的 `dependencies` 部分。一个添加 MySQL JDBC Driver 依赖的示例如下，相关 JAR 应预先置入本地 Maven 仓库或 Maven Central 等远程 Maven 仓库。

```
<dependencies>
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>9.3.0</version>
    </dependency>
</dependencies>
```

构建 Linux Docker Image

前提条件

贡献者必须在设备安装，

1. OpenJDK 11 或更高版本
2. 可运行 Linux Containers 的 Docker Engine

下文分别讨论在 Ubuntu 与 Windows 下可能的所需操作。

Ubuntu

假设贡献者处于新的 Ubuntu 22.04.5 LTS 实例下，且已配置 git。可在 bash 通过如下命令利用 SDKMAN! 安装 OpenJDK 21。

```
sudo apt install unzip zip -y
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
sdk install java 21.0.7-ms
sdk use java 21.0.7-ms
```

可在 bash 通过如下命令安装 Rootful 模式的 Docker Engine。本文不讨论更改 `/etc/docker/daemon.json` 的默认 logging driver。

```
sudo apt update && sudo apt upgrade -y
sudo apt-get remove docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd runc
cd /tmp/
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin -y
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

Windows

假设贡献者处于新的 Windows 11 Home 24H2 实例下，且已安装和配置 `git-for-windows/git` 和 `PowerShell/PowerShell`。

可在 Powershell 7 通过如下命令利用 `version-fox/vfox` 安装 OpenJDK 21。

```
winget install version-fox.vfox
if (-not (Test-Path -Path $PROFILE)) { New-Item -Type File -Path $PROFILE -Force };
Add-Content -Path $PROFILE -Value 'Invoke-Expression "$(vfox activate pwsh)"'
# 此时需要打开新的 Powershell 7 终端
```

```
vfox add java
vfox install java@21.0.7-ms
vfox use --global java@21.0.7-ms
```

当 Windows 弹出窗口，要求允许类似 `C:\users\lingh\.version-fox\cache\java\v-21.0.7-ms\java-21.0.7-ms\bin\java.exe` 路径的应用通过 Windows 防火墙时，应当批准。背景参考 https://support.microsoft.com/en-us/windows/risks-of-allowing-apps-through-windows-firewall-654559af-3f54-3dcf-349f-71ccd90bcc5c 。

可在 Powershell 7 通过如下命令启用 WSL2 并设置 `Ubuntu WSL` 为默认 Linux 发行版。

```
wsl --install
```

完成 WSL2 的启用后，在 https://rancherdesktop.io/ 下载和安装 `rancher-sandbox/rancher-desktop`，并设置使用 `dockerd(moby)` 的 `Container Engine`。本文不讨论更改 Linux 发行版 `rancher-desktop` 的 `/etc/docker/daemon.json` 的默认 logging driver。

构建包含动态链接的 GraalVM Native Image 的 Docker Image

可执行如下命令构建。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -am -pl distribution/proxy-native -T1C "-Pdocker.build.native.linux" "-DskipTests" clean package
```

一个可能的 Docker Compose 示例为，

```
services:
  apache-shardingsphere-proxy-native:
    image: apache/shardingsphere-proxy-native:5.5.3-SNAPSHOT
    volumes:
      - ./custom/conf:/opt/shardingsphere-proxy/conf
    ports:
      - "3307:3307"
```

构建包含大部分静态链接的 GraalVM Native Image 的 Docker Image

可执行如下命令构建。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -am -pl distribution/proxy-native -T1C "-Pdocker.build.native.linux" "-Dproxy.native.dockerfile=Dockerfile-linux-mostly" "-Dproxy.native.image.tag=5.5.3-SNAPSHOT-mostly" "-DskipTests" clean package
```

一个可能的 Docker Compose 示例为，

```
services:
  apache-shardingsphere-proxy-native:
    image: apache/shardingsphere-proxy-native:5.5.3-SNAPSHOT-mostly
    volumes:
      - ./custom/conf:/opt/shardingsphere-proxy/conf
    ports:
      - "3307:3307"
```

构建包含完全静态链接的 GraalVM Native Image 的 Docker Image

可执行如下命令构建。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -am -pl distribution/proxy-native -T1C "-Pdocker.build.native.linux" "-Dproxy.native.dockerfile=Dockerfile-linux-static" "-Dproxy.native.image.tag=5.5.3-SNAPSHOT-static" "-DskipTests" clean package
```

一个可能的 Docker Compose 示例为,

```
services:
  apache-shardingsphere-proxy-native:
    image: apache/shardingsphere-proxy-native:5.5.3-SNAPSHOT-static
    volumes:
      - ./custom/conf:/opt/shardingsphere-proxy/conf
    ports:
      - "3307:3307"
```

仅构建产物

前提条件

贡献者必须在设备安装,

1. GraalVM CE 22.0.2，或与 GraalVM CE 22.0.2 兼容的 GraalVM 下游发行版。以 GraalVM Native Image 为准。
2. 编译 GraalVM Native Image 所需要的本地工具链。以 https://www.graalvm.org/latest/reference-manual/native-image/#prerequisites 为准。

在 Ubuntu 与 Windows 下可能的所需操作与开发和测试一致。但不需要安装 Container Runtime。

静态编译所需的本地工具链

开发者如需构建 大部分静态链接的 `GraalVM Native Image` 或 完全静态链接的 `GraalVM Native Image`，则需要按 https://www.graalvm.org/latest/reference-manual/native-image/guides/build-static-executables/ 要求，从源代码构建 musl。

构建动态链接的 GraalVM Native Image

可执行如下命令构建。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -am -pl distribution/proxy-native -T1C -DskipTests "-Prelease.native" clean
package
```

构建大部分静态链接的 GraalVM Native Image

可执行如下命令构建。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -am -pl distribution/proxy-native -T1C -DskipTests "-Prelease.native" "-
DbuildArgs=-H:+AddAllCharsets,-H:+StaticExecutableWithDynamicLibC" clean package
```

构建完全静态链接的 GraalVM Native Image

可执行如下命令构建。

```
git clone git@github.com:apache/shardingsphere.git
cd ./shardingsphere/
./mvnw -am -pl distribution/proxy-native -T1C -DskipTests "-Prelease.native" "-
DbuildArgs=-H:+AddAllCharsets,--static,--libc=musl" clean package
```

使用 GraalVM Native Image

无论 GraalVM Native Image 属于什么变体，通过命令行启动 Native Image, 都需要带上 3 个参数,

1. 第 1 个参数为 ShardingSphere Proxy Native 使用的端口,
2. 第 2 个参数为用户编写的包含 `global.yaml` 配置文件的文件夹,
3. 第 3 个参数为要侦听的主机，如果为 `0.0.0.0` 则允许任意数据库客户端均可访问 ShardingSphere Proxy Native。

已完成构建的 GraalVM Native Image 的二进制文件仅可设置命令行参数。这意味着,

1. 用户仅可在构建 GraalVM Native Image 的过程中设置 JVM 参数

2. 用户无法针对已完成构建的 GraalVM Native Image 的二进制文件设置 JVM 参数

Ubuntu 下假设已存在包含 `global.yaml` 的 `conf` 文件夹为 `/tmp/conf`，可能的示例为，

```
cd ./shardingsphere/
cd ./distribution/proxy-native/target/apache-shardingsphere-5.5.3-SNAPSHOT-shardingsphere-proxy-bin/bin
./proxy-native "3307" "/tmp/conf" "0.0.0.0"
```

Windows 下假设已存在包含 `global.yaml` 的 `conf` 文件夹为 `C:\Users\shard\Downloads\conf`，可能的示例为，

```
cd ./shardingsphere/
cd ./distribution/proxy-native/target/apache-shardingsphere-5.5.3-SNAPSHOT-shardingsphere-proxy-bin/bin
./proxy-native.exe "3307" "C:\Users\shard\Downloads\conf" "0.0.0.0"
```

使用限制

GraalVM Native Image 变体选择

一般情况下开发者仅需使用动态链接的 `GraalVM Native Image`。

当开发者仅使用可运行 `linux/amd64` OS/Arch Containers 的 Container Runtime，并希望获得更小体积的 Docker Image 时，可考虑使用 大部分静态链接的 `GraalVM Native Image` 或 完全静态链接的 `GraalVM Native Image`。背景参考 https://www.graalvm.org/latest/reference-manual/native-image/guides/build-static-executables/ 和 https://github.com/oracle/graal/issues/2589 。大部分静态链接的可执行文件，是 `golang/go` 推广的，针对静态链接 musl libc 实现的另一种替代方案。

可观察性

针对 ShardingSphere Proxy Native，其提供的可观察性的能力与可观察性并不一致。

用户可以使用 https://www.graalvm.org/latest/reference-manual/tools/ 提供的一系列命令行工具或可视化工具观察 GraalVM Native Image 的内部行为，并根据其要求在 Linux 下使用 VSCode 完成 Debug 工作。如果用户正在使用 IntelliJ IDEA 并且希望调试生成的 GraalVM Native Image，用户可以关注 https://blog.jetbrains.com/idea/2022/06/intellij-idea-2022-2-eap-5/#Experimental_GraalVM_Native_Debugger_for_Java 及其后继。如果用户使用的不是 Linux，则无法对 GraalVM Native Image 进行 Debug，请关注尚未关闭的 https://github.com/oracle/graal/issues/5648 。

对于使用 `ShardingSphere Agent` 等 Java Agent 的情形，GraalVM 的 `native-image` 组件尚未完全支持在构建 Native Image 时使用 javaagent，用户需要关注尚未关闭的 https://github.com/oracle/graal/issues/8177 。若用户期望在 ShardingSphere Proxy Native 下使用这类 Java Agent，则需要关注 https://github.com/oracle/graal/pull/8077 涉及的变动。

`linux/riscv64` OS/Arch 限制

当前，ShardingSphere Proxy Native 未提供在 `linux/riscv64` OS/Arch 上的可用性。如果开发者使用 `linux/riscv64` 设备，应参考 https://medium.com/graalvm/graalvm-native-image-meets-risc-v-899be38eddd9 的处理修改 Proxy Native 的构建配置。

自 https://github.com/oracle/graal/issues/6855 开始，`LLVM backend` 需要从 GraalVM 的源码构建才可使用。参考 https://github.com/oracle/graal/blob/master/substratevm/src/com.oracle.svm.core.graal.llvm/src/com/oracle/svm/core/graal/llvm/LLVMBackend.md 。

Windows Containers 限制

ShardingSphere Proxy Native 可通过包含 `Microsoft.VisualStudio.2022.Community` 的本地工具链，在 Windows 上开箱即用地构建 GraalVM Native Image。

当前受 https://github.com/graalvm/container/issues/106 影响, ShardingSphere 暂时不为通过 Windows 编译的 动态链接的 `GraalVM Native Image` 提供构建 Docker Image 所需的构建配置。

Wasm 模块限制

尽管 `Oracle GraalVM Early Access Builds For JDK 25 EA 24` 已支持构建 `Wasm` 模块形态的 GraalVM Native Image，但 ShardingSphere 尚未准备好在 OpenJDK 25 下测试 CI。

当前，ShardingSphere Proxy Native 未提供编译为 `Wasm` 模块所需的构建配置。

使用 Helm

背景信息

使用 Helm 在 Kubernetes 集群中引导 ShardingSphere-Proxy 实例进行安装。关于 ShardingSphere Helm Charts 的更多内容可以参考：ShardingSphere-on-Cloud 子项目。

前提条件

- kubernetes 1.18+
- kubectl
- helm 3.3.0+
- 可以动态申请 PV(Persistent Volumes) 的 StorageClass 用于持久化数据。（可选）

操作步骤

在线安装

1. 将 ShardingSphere-Proxy 添加到 Helm 本地仓库：

```
helm repo add shardingsphere https://shardingsphere.apache.org/charts
```

2. 以 ShardingSphere-Proxy 命名安装 charts：

```
helm install shardingsphere-proxy shardingsphere/shardingsphere-proxy
```

源码安装

1. 执行下述命令以执行默认配置进行安装。

```
git clone https://github.com/apache/shardingsphere-on-cloud.git
cd charts/shardingsphere-proxy/charts/governance
helm dependency build
cd ../..
helm dependency build
cd ..
helm install shardingsphere-proxy shardingsphere-proxy
```

说明：

1. 其他的配置详见下方的配置列表。
2. 执行 helm list 获取所有安装的 release。

卸载

1. 默认删除所有发布记录，增加 `--keep-history` 参数保留发布记录。

```
helm uninstall shardingsphere-proxy
```

参数解释

治理节点配置项

配置项	描述	值
`governance.enabled`	用来切换是否使用治理节点的 chart	`true`

治理节点 ZooKeeper 配置项

配置项	描述	值
`governanc e.zookeeper. enabled`	用来切换是否使用 ZooKeeper 的 chart	‘ *true*‘
`governance.zoo keeper. replicaCount`	ZooKeeper 节点数量	`1`
“ governance.zookeeper. persistence.enabled“	标识 ZooKeeper 是否使用持久卷申领（PersistentVolumeClaim）用来申请持久卷（PersistentVolume）	“ false“
`gover nance.zookeeper. persi stence. storageClass`	持久卷（PersistentVolume）的存储类（StorageClass）	`""`
`gove rnance.zookeeper. pers istence. accessModes`	持久卷（PersistentVolume）的访问模式	`["Rea dWriteO nce"]`
`governance.zookeep er. persistence.size`	持久卷（PersistentVolume）大小	`8Gi`
`governance.zookeep er. resources.limits`	ZooKeeper 容器的资源限制	`{}`
`govern ance.zookeeper. resour ces.requests. memory`	ZooKeeper 容器申请的内存	“ 256Mi“
`gov ernance.zookeeper. res ources.requests.cpu`	ZooKeeper 容器申请的 cpu 核数	‘ *250m*‘

计算节点 ShardingSphere-Proxy 配置项

配置项	描述	值
comp ute.image.repository	ShardingSphere-Proxy 的镜像名	apache/shar dingsphere-proxy
comp ute.image.pullPolicy	ShardingSphere-Proxy 镜像拉取策略	IfNotPresent
compute.image.tag	ShardingSphere-Proxy 镜像标签	5.1.2
comp ute.imagePullSecrets	拉取私有仓库的凭证	[]
comp ute.resources.limits	ShardingSphere-Proxy 容器的资源限制	{}
compute.resou rces. requests.memory	ShardingSphere-Proxy 容器申请的内存	2Gi
compute.re sources. requests.cpu	ShardingSphere-Proxy 容器申请的 cpu 核数	200m
compute.replicas	ShardingSphere-Proxy 节点个数	3
“compute.service.type“	ShardingSphere-Proxy 网络模式	ClusterIP
“compute.service.port“	ShardingSphere-Proxy 暴露端口	3307
compute.my sqlConnector. version	MySQL 驱动版本	5.1.49
compute.startPort	ShardingSphere-Proxy 启动端口	3307
“compute.serverConfig“	ShardingSphere-Proxy 模式配置文件	""

配置示例

```
#
#  Licensed to the Apache Software Foundation (ASF) under one or more
#  contributor license agreements.  See the NOTICE file distributed with
#  this work for additional information regarding copyright ownership.
#  The ASF licenses this file to You under the Apache License, Version 2.0
#  (the "License"); you may not use this file except in compliance with
#  the License.  You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
#  Unless required by applicable law or agreed to in writing, software
#  distributed under the License is distributed on an "AS IS" BASIS,
#  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#  See the License for the specific language governing permissions and
#  limitations under the License.
#

## @section Governance-Node parameters
```

```
## @param governance.enabled Switch to enable or disable the governance helm chart
##
governance:
  enabled: true
  ## @section Governance-Node ZooKeeper parameters
  zookeeper:
    ## @param governance.zookeeper.enabled Switch to enable or disable the
ZooKeeper helm chart
    ##
    enabled: true
    ## @param governance.zookeeper.replicaCount Number of ZooKeeper nodes
    ##
    replicaCount: 1
    ## ZooKeeper Persistence parameters
    ## ref: https://kubernetes.io/docs/user-guide/persistent-volumes/
    ## @param governance.zookeeper.persistence.enabled Enable persistence on
ZooKeeper using PVC(s)
    ## @param governance.zookeeper.persistence.storageClass Persistent Volume
storage class
    ## @param governance.zookeeper.persistence.accessModes Persistent Volume access
modes
    ## @param governance.zookeeper.persistence.size Persistent Volume size
    ##
    persistence:
      enabled: false
      storageClass: ""
      accessModes:
        - ReadWriteOnce
      size: 8Gi
    ## ZooKeeper's resource requests and limits
    ## ref: https://kubernetes.io/docs/user-guide/compute-resources/
    ## @param governance.zookeeper.resources.limits The resources limits for the
ZooKeeper containers
    ## @param governance.zookeeper.resources.requests.memory The requested memory
for the ZooKeeper containers
    ## @param governance.zookeeper.resources.requests.cpu The requested cpu for the
ZooKeeper containers
    ##
    resources:
      limits: {}
      requests:
        memory: 256Mi
        cpu: 250m

## @section Compute-Node parameters
##
compute:
  ## @section Compute-Node ShardingSphere-Proxy parameters
```

```
  ## ref: https://kubernetes.io/docs/concepts/containers/images/
  ## @param compute.image.repository Image name of ShardingSphere-Proxy.
  ## @param compute.image.pullPolicy The policy for pulling ShardingSphere-Proxy
image
  ## @param compute.image.tag ShardingSphere-Proxy image tag
  ##
  image:
    repository: "apache/shardingsphere-proxy"
    pullPolicy: IfNotPresent
    ## Overrides the image tag whose default is the chart appVersion.
    ##
    tag: "5.1.2"
  ## @param compute.imagePullSecrets Specify docker-registry secret names as an
array
  ## e.g:
  ## imagePullSecrets:
  ##   - name: myRegistryKeySecretName
  ##
  imagePullSecrets: []
  ## ShardingSphere-Proxy resource requests and limits
  ## ref: https://kubernetes.io/docs/concepts/configuration/manage-resources-
containers/
  ## @param compute.resources.limits The resources limits for the ShardingSphere-
Proxy containers
  ## @param compute.resources.requests.memory The requested memory for the
ShardingSphere-Proxy containers
  ## @param compute.resources.requests.cpu The requested cpu for the
ShardingSphere-Proxy containers
  ##
  resources:
    limits: {}
    requests:
      memory: 2Gi
      cpu: 200m
  ## ShardingSphere-Proxy Deployment Configuration
  ## ref: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
  ## ref: https://kubernetes.io/docs/concepts/services-networking/service/
  ## @param compute.replicas Number of cluster replicas
  ##
  replicas: 3
  ## @param compute.service.type ShardingSphere-Proxy network mode
  ## @param compute.service.port ShardingSphere-Proxy expose port
  ##
  service:
    type: ClusterIP
    port: 3307
  ## MySQL connector Configuration
  ## ref: https://shardingsphere.apache.org/document/current/en/quick-start/
```

```
shardingsphere-proxy-quick-start/
  ## @param compute.mysqlConnector.version MySQL connector version
  ##
  mysqlConnector:
    version: "5.1.49"
  ## @param compute.startPort ShardingSphere-Proxy start port
  ## ShardingSphere-Proxy start port
  ## ref: https://shardingsphere.apache.org/document/current/en/user-manual/
shardingsphere-proxy/startup/docker/
  ##
  startPort: 3307
  ## @section Compute-Node ShardingSphere-Proxy ServerConfiguration parameters
  ## NOTE: If you use the sub-charts to deploy Zookeeper, the server-lists field
must be "{{ printf \"%s-zookeeper.%s:2181\" .Release.Name .Release.Namespace }}",
  ## otherwise please fill in the correct zookeeper address
  ## The global.yaml is auto-generated based on this parameter.
  ## If it is empty, the global.yaml is also empty.
  ## ref: https://shardingsphere.apache.org/document/current/en/user-manual/
shardingsphere-jdbc/yaml-config/mode/
  ## ref: https://shardingsphere.apache.org/document/current/en/user-manual/common-
config/builtin-algorithm/metadata-repository/
  ##
  serverConfig:
    ## @section Compute-Node ShardingSphere-Proxy ServerConfiguration authority
parameters
    ## NOTE: It is used to set up initial user to login compute node, and authority
data of storage node.
    ## ref: https://shardingsphere.apache.org/document/current/en/user-manual/
shardingsphere-proxy/yaml-config/authentication/
    ## @param compute.serverConfig.authority.privilege.type authority provider for
storage node, the default value is ALL_PERMITTED
    ## @param compute.serverConfig.authority.users[0].password Password for compute
node.
    ## @param compute.serverConfig.authority.users[0].user Username,authorized host
for compute node. Format: <username>@<hostname> hostname is % or empty string means
do not care about authorized host
    ##
    authority:
      privilege:
        type: ALL_PRIVILEGES_PERMITTED
      users:
      - password: root
        user: root@%
    ## @section Compute-Node ShardingSphere-Proxy ServerConfiguration mode
Configuration parameters
    ## @param compute.serverConfig.mode.type Type of mode configuration. Now only
support Cluster mode
    ## @param compute.serverConfig.mode.repository.props.namespace Namespace of
```

```
registry center
    ## @param compute.serverConfig.mode.repository.props.server-lists Server lists
of registry center
    ## @param compute.serverConfig.mode.repository.props.maxRetries Max retries of
client connection
    ## @param compute.serverConfig.mode.repository.props.
operationTimeoutMilliseconds Milliseconds of operation timeout
    ## @param compute.serverConfig.mode.repository.props.retryIntervalMilliseconds
Milliseconds of retry interval
    ## @param compute.serverConfig.mode.repository.props.timeToLiveSeconds Seconds
of ephemeral data live
    ## @param compute.serverConfig.mode.repository.type Type of persist repository.
Now only support ZooKeeper
    ## @param compute.serverConfig.mode.overwrite Whether overwrite persistent
configuration with local configuration
    ##
    mode:
      type: Cluster
      repository:
        type: ZooKeeper
        props:
          maxRetries: 3
          namespace: governance_ds
          operationTimeoutMilliseconds: 5000
          retryIntervalMilliseconds: 500
          server-lists: "{{ printf \"%s-zookeeper.%s:2181\" .Release.Name .Release.
Namespace }}"
          timeToLiveSeconds: 60
      overwrite: true
```

添加依赖

本章主要介绍 ShardingSphere 可选依赖的下载方式。

添加 Narayana 依赖

添加 Narayana 依赖包

添加 Narayana 依赖需要下载以下 jar 文件并将其添加至 `ext-lib` 下。

jar 文件下载地址

- arjuna-5.12.7.Final.jar
- common-5.12.7.Final.jar
- jboss-connector-api_1.7_spec-1.0.0.Final.jar
- jboss-logging-3.2.1.Final.jar
- jboss-transaction-api_1.2_spec-1.0.0.Alpha3.jar
- jboss-transaction-spi-7.6.1.Final.jar
- jta-5.12.7.Final.jar
- narayana-jts-integration-5.12.7.Final.jar
- shardingsphere-transaction-xa-narayana.jar

请根据 `proxy` 版本下载对应 `shardingsphere-transaction-xa-narayana.jar` 文件。

9.2.2 YAML 配置

ShardingSphere-JDBC 的 YAML 配置是 ShardingSphere-Proxy 的子集。在 `global.yaml` 文件中，ShardingSphere-Proxy 能够额外配置权限功能和更多的 Proxy 专有属性。

说明：YAML 配置文件支持配置内容超过 3MB。

本章节将介绍 ShardingSphere-Proxy 的 YAML 额外配置。

认证和授权

背景信息

在 ShardingSphere-Proxy 中，通过 `authority` 来配置用户的认证和授权信息。

得益于 ShardingSphere 的可插拔架构，Proxy 提供了两种级别的权限提供者，分别是：

- `ALL_PERMITTED`：每个用户都拥有所有权限，无需专门授权；（将在未来版本中删除）
- `DATABASE_PERMITTED`：为用户授予指定逻辑库的权限，通过 `user-database-mappings` 进行定义。（推荐使用）

在配置 `authority` 时，管理员可根据需要选择使用哪一种权限提供者。

参数解释

```
authority:
  users:
    - user: # 用于登录计算节点的用户名和授权主机的组合，格式：<username>@<hostname>，
hostname 为 % 或空字符串表示不限制授权主机，username 和 hostname 大小写不敏感
      password: # 用户密码
      admin: # 可选项，管理员身份标识。若为 true，该用户拥有最高权限，缺省值为 false
      authenticationMethodName: # 可选项，用于为用户指定密码认证方式
  authenticators: # 可选项，默认不需要配置，Proxy 根据前端协议类型自动选择
    authenticatorName:
      type: # 密码认证类型
  defaultAuthenticator: # 可选项，指定一个 authenticatorName 作为默认的密码认证方式
  privilege:
    type: # 权限提供者类型，缺省值为 ALL_PERMITTED
```

配置示例

极简配置

```
authority:
  users:
    - user: root@%
      password: root
    - user: sharding
      password: sharding
```

说明：- 定义了两个用户：root@% 和 sharding；- 未定义 authenticators 和 authenticationMethodName，Proxy 将根据前端协议自动选择；- 未指定 privilege type，采用默认的 ALL_PERMITTED。

认证配置

自定义认证配置能够满足用户在一些特定场景下的需求。以 openGauss 作为前端协议类型为例，其默认的认证算法为 scram-sha-256。如果用户 sharding 需要用旧版本的 psql 客户端（不支持 scram-sha-256）连接 Proxy，则管理员可能允许 sharding 使用 md5 方式进行密码认证。配置方式如下：

```
authority:
  users:
    - user: root@127.0.0.1
      password: root
    - user: sharding
      password: sharding
      authenticationMethodName: md5
```

```
  authenticators:
    md5:
      type: MD5
  privilege:
    type: ALL_PERMITTED
```

说明：- 定义了两个用户：root@127.0.0.1 和 sharding；- 为用户 sharding 指定了 MD5 方式进行密码认证；- 没有为 root@127.0.0.1 指定认证方式，Proxy 将根据前端协议自动选择；- 指定权限提供者为 ALL_PERMITTED。

授权配置

ALL_PERMITTED （将在未来版本中删除）

```
authority:
  users:
    - user: root@127.0.0.1
      password: root
    - user: sharding
      password: sharding
  privilege:
    type: ALL_PERMITTED
```

说明：- 定义了两个用户：root@127.0.0.1 和 sharding；- 未定义 authenticators 和 authenticationMethodName，Proxy 将根据前端协议自动选择；- 指定权限提供者为 ALL_PERMITTED。

DATABASE_PERMITTED （推荐使用）

```
authority:
  users:
    - user: root@127.0.0.1
      password: root
      admin: true
    - user: sharding
      password: sharding
    - user: test
      password: sharding
  privilege:
    type: DATABASE_PERMITTED
    props:
      user-database-mappings: sharding@%=*, test@%=test_db, test@%=sharding_db
```

说明：- 定义了一个超级用户 root@127.0.0.1 和两个普通用户：sharding 和 test；- 未定义 authenticators 和 authenticationMethodName，Proxy 将根据前端协议自动选择；- 指定权限

提供者为 `DATABASE_PERMITTED`，并授权 `sharding@%` 用户访问所有逻辑库 (`*`)，test 用户仅能访问 test_db 和 sharding_db。

属性配置

背景信息

Apache ShardingSphere 提供了丰富的系统配置属性，用户可通过 `global.yaml` 进行配置。

参数解释

属性配置可以通过 DistSQL#RAL 在线修改。其中支持动态修改的属性立即生效，不支持动态修改的属性在重启后生效。

配置示例

完整配置示例请参考 ShardingSphere 仓库内的 `global.yaml`：https://github.com/apache/shardingsphere/blob/612cd5d8e802d0d712a3a4d89da8fdc048d23879/proxy/bootstrap/src/main/resources/conf/global.yaml#L71-L89

规则配置

背景信息

本节介绍如何进行 ShardingSphere-Proxy 的规则配置。

参数解释

ShardingSphere-Proxy 的规则配置与 ShardingSphere-JDBC 一致，具体规则请参考 ShardingSphere-JDBC 规则配置。

注意事项

与 ShardingSphere-JDBC 不同的是，以下规则需要配置在 ShardingSphere-Proxy 的 `global.yaml` 中：

- SQL 解析

```
sqlParser:
  sqlStatementCache:
    initialCapacity: 2000
    maximumSize: 65535
  parseTreeCache:
    initialCapacity: 128
    maximumSize: 1024
```

- 分布式事务

```
transaction:
  defaultType: XA
  providerType: Atomikos
```

- SQL 翻译

```
sqlTranslator:
  type:
  useOriginalSQLWhenTranslatingFailed:
```

- 联邦查询

```
sqlFederation:
  sqlFederationEnabled: true
  allQueryUseSQLFederation: false
  executionPlanCache:
    initialCapacity: 2000
    maximumSize: 65535
```

数据源配置

背景信息

ShardingSphere-Proxy 支持常见的数据库连接池: HikariCP、C3P0、DBCP（C3P0、DBCP 需要从 shardingsphere-plugin 仓库获取插件）。

可以通过参数 `dataSourceClassName` 指定连接池，当不指定时，默认的的数据库连接池为 HikariCP。

参数解释

```
dataSources: # 数据源配置，可配置多个 <data-source-name>
  <data_source_name>: # 数据源名称
    dataSourceClassName: # 数据源连接池完整类名
    url: # 数据库 URL 连接
    username: # 数据库用户名
    password: # 数据库密码
    # ... 数据库连接池的其它属性
```

配置示例

```
dataSources:
  ds_1:
    url: jdbc:mysql://localhost:3306/ds_1
    username: root
    password:
  ds_2:
    dataSourceClassName: com.mchange.v2.c3p0.ComboPooledDataSource
    url: jdbc:mysql://localhost:3306/ds_2
    username: root
    password:
  ds_3:
    dataSourceClassName: org.apache.commons.dbcp2.BasicDataSource
    url: jdbc:mysql://localhost:3306/ds_3
    username: root
    password:

  # 配置其他数据源
```

9.2.3 DistSQL

本章节将介绍 DistSQL 的详细语法。

定义

DistSQL（Distributed SQL）是 Apache ShardingSphere 特有的操作语言。它与标准 SQL 的使用方式完全一致，用于提供增量功能的 SQL 级别操作能力。

灵活的规则配置和资源管控能力是 Apache ShardingSphere 的特点之一。

在使用 4.x 及其之前版本时，开发者虽然可以像使用原生数据库一样操作数据，但却需要通过本地文件或注册中心配置资源和规则。然而，操作习惯变更，对于运维工程师并不友好。

从 5.x 版本开始，DistSQL（Distributed SQL）让用户可以像操作数据库一样操作 Apache ShardingSphere，使其从面向开发人员的框架和中间件转变为面向运维人员的数据库产品。

相关概念

DistSQL 细分为 RDL、RQL、RAL 和 RUL 四种类型。

RDL

Resource & Rule Definition Language，负责资源和规则的创建、修改和删除。

RQL

Resource & Rule Query Language，负责资源和规则的查询和展现。

RAL

Resource & Rule Administration Language，负责强制路由、熔断、配置导入导出、数据迁移控制等管理功能。

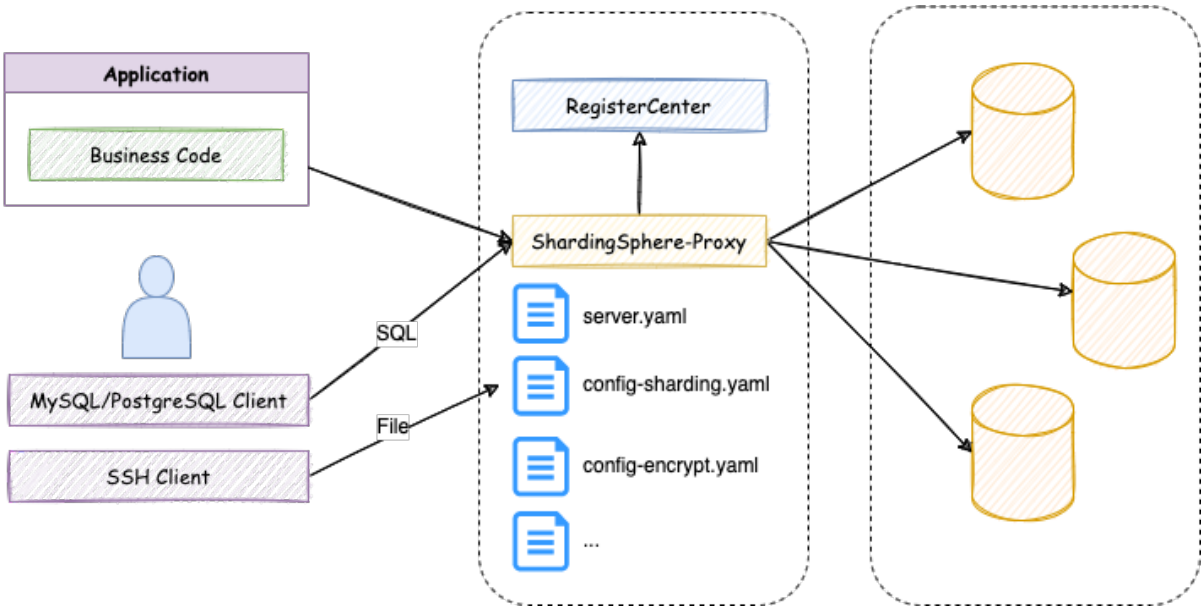
RUL

Resource & Rule Utility Language，负责 SQL 解析、SQL 格式化、执行计划预览等功能。

对系统的影响

之前

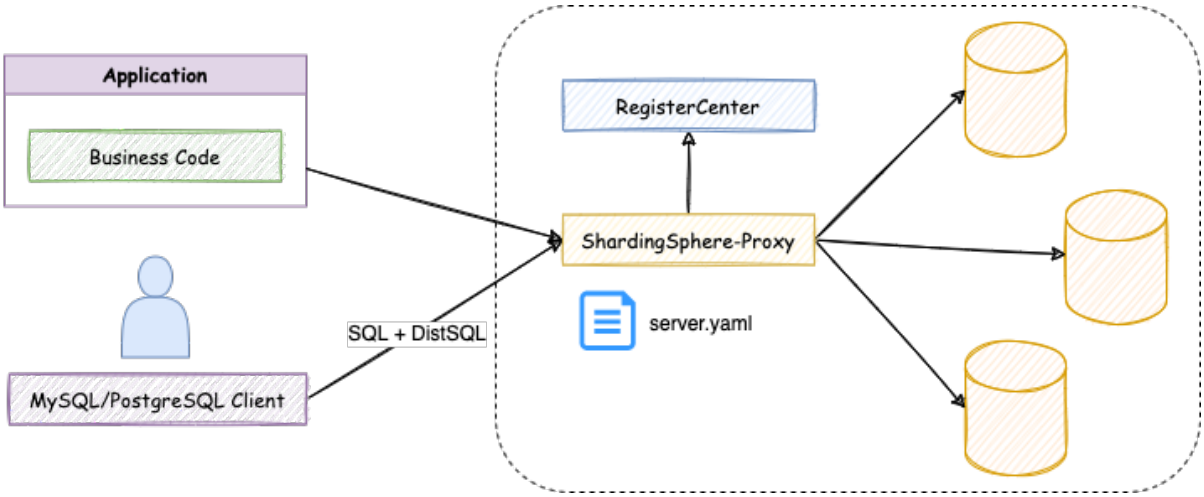
在拥有 DistSQL 以前，用户一边使用 SQL 语句操作数据，一边使用 YAML 文件来管理 ShardingSphere 的配置，如下图：



这时用户不得不面对以下几个问题：- 需要通过不同类型的客户端来操作数据和管理 ShardingSphere 规则；- 多个逻辑库需要多个 YAML 文件；- 修改 YAML 需要文件的编辑权限；- 修改 YAML 后需要重启 ShardingSphere。

之后

随着 DistSQL 的出现，对 ShardingSphere 的操作方式也得到了改变：



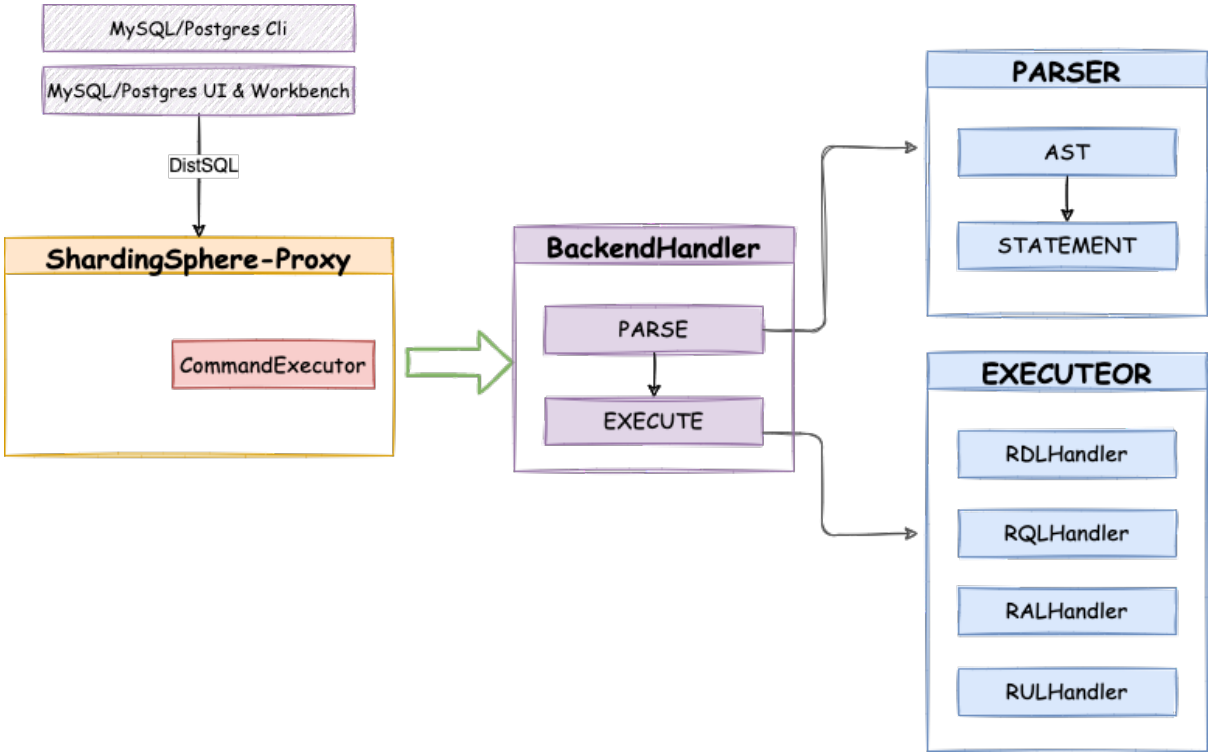
现在，用户的使用体验得到了巨大改善：- 使用相同的客户端来管理数据和 ShardingSphere 配置；- 不再额外创建 YAML 文件，通过 DistSQL 管理逻辑库；- 不再需要文件的编辑权限，通过 DistSQL 来管理配置；- 配置的变更实时生效，无需重启 ShardingSphere。

使用限制

DistSQL 只能用于 ShardingSphere-Proxy，ShardingSphere-JDBC 暂不提供。

原理介绍

与标准 SQL 一样，DistSQL 由 ShardingSphere 的解析引擎进行识别，将输入语句转换为抽象语法树，进而生成各个语法对应的 `Statement`，最后由合适的 `Handler` 进行业务处理。整体流程如下图所示：



相关参考

用户手册：DistSQL

语法

本章节将对 DistSQL 的语法进行详细说明，并以实际的列子介绍 DistSQL 的使用。

语法规则

在 DistSQL 语句中，除关键字外，其余元素的输入格式应符合以下规则。

标识符

1. 标识符代表 SQL 语句中的一个对象，包括：

- 数据库名称
- 表名
- 列名
- 索引名称
- 资源名称
- 规则名称
- 算法名称

2. 标识符中允许使用的字符有：`[a-z,A-Z,0-9,_]`（字母、数字、下划线），且应以字母开头。
3. 当标识符中出现关键字或特殊字符时，使用反引号 (‘) 。

字面量

字面量包括字符串、整数值和布尔值：

- 字符串：是由单引号 (’) 或双引号 (“) 括起来的字符序列；
- 整数值：一般为正整数，如 0-9；

说明：部分 DistSQL 语法允许负值，此时可在数字前加负号（-），如 -1。

- 布尔值：TRUE 或 FALSE，大小写不敏感。

特别说明

- 当指定用户自定义算法类型名称时必须使用 `""` 对算法类型名称进行标注，例如 `NAME="AlgorithmTypeName"`；
- 当指定 ShardingSphere 内置算法类型名称时可以不使用 `""` 对算法类型名称进行标注，例如 `NAME=HASH_MODE`；

RDL 语法

RDL (Resource & Rule Definition Language) 为 Apache ShardingSphere 的资源和规则定义语言。

存储单元定义

本章节将对存储单元定义的语法进行详细说明。

REGISTER STORAGE UNIT

描述

`REGISTER STORAGE UNIT` 语法用于为当前所选逻辑库（DATABASE）注册存储单元。

语法

```
RegisterStorageUnit ::=
  'REGISTER' 'STORAGE' 'UNIT' ifNotExists? storageUnitsDefinition (','
checkPrivileges)?

storageUnitsDefinition ::=
  storageUnitDefinition (',' storageUnitDefinition)*
```

```
storageUnitDefinition ::=
  storageUnitName '(' ('HOST' '=' hostName ',' 'PORT' '=' port ',' 'DB' '=' dbName
| 'URL' '=' url) ',' 'USER' '=' user (',' 'PASSWORD' '=' password)? (','
propertiesDefinition)?')'

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

storageUnitName ::=
  identifier

hostname ::=
  string

port ::=
  int

dbName ::=
  string

url ::=
  string

user ::=
  string

password ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal

checkPrivileges ::=
  'CHECK_PRIVILEGES' '=' privilegeType (',' privilegeType)*

privilegeType ::=
  identifier
```

特别说明

- 注册存储单元前请确认已经在 Proxy 中创建逻辑数据库，并执行 `use` 命令成功选择一个逻辑数据库；
- 确认注册的存储单元是可以正常连接的，否则将不能注册成功；
- `storageUnitName` 区分大小写；
- `storageUnitName` 在当前逻辑库中需要唯一；
- `storageUnitName` 命名只允许使用字母、数字以及 `_`，且必须以字母开头；
- `PROPERTIES` 为可选参数，用于自定义连接池属性，`key` 必须和连接池参数名一致；
- 可通过 `CHECK_PRIVILEGES` 指定注册时校验存储单元用户的权限，`privilegeType` 支持的类型有 `SELECT`、`XA`、`PIPELINE`、`NONE`，缺省值为 `SELECT`，当类型列表中包含 `NONE` 时，跳过权限校验。

示例

- 使用 HOST & PORT 方式注册存储单元

```
REGISTER STORAGE UNIT ds_0 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="db_0",
    USER="root",
    PASSWORD="root"
);
```

- 使用 HOST & PORT 方式注册存储单元并设置连接池属性

```
REGISTER STORAGE UNIT ds_1 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="db_1",
    USER="root",
    PASSWORD="root",
    PROPERTIES("maximumPoolSize"=10)
);
```

- 使用 URL 方式注册存储单元并设置连接池属性

```
REGISTER STORAGE UNIT ds_2 (
    URL="jdbc:mysql://127.0.0.1:3306/db_2?serverTimezone=UTC&useSSL=false&
allowPublicKeyRetrieval=true",
    USER="root",
    PASSWORD="root",
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"=30000)
);
```

- 使用 ifNotExists 子句注册存储单元

```
REGISTER STORAGE UNIT IF NOT EXISTS ds_0 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="db_0",
    USER="root",
    PASSWORD="root"
);
```

- 注册时校验 SELECT、XA 和 PIPELINE 权限

```
REGISTER STORAGE UNIT ds_3 (
    URL="jdbc:mysql://127.0.0.1:3306/db_3?serverTimezone=UTC&useSSL=false&
allowPublicKeyRetrieval=true",
    USER="root",
    PASSWORD="root",
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"=30000)
), CHECK_PRIVILEGES=SELECT,XA,PIPELINE;
```

保留字

REGISTER、STORAGE、UNIT、HOST、PORT、DB、USER、PASSWORD、PROPERTIES、URL、CHECK_PRIVILEGES

相关链接

- 保留字

ALTER STORAGE UNIT

描述

ALTER STORAGE UNIT 语法用于修改当前所选逻辑库（DATABASE）的存储单元。

语法

```
AlterStorageUnit ::=
  'ALTER' 'STORAGE' 'UNIT' storageUnitsDefinition (',' checkPrivileges)?

storageUnitsDefinition ::=
  storageUnitDefinition (',' storageUnitDefinition)*

storageUnitDefinition ::=
  storageUnitName '(' ('HOST' '=' hostName ',' 'PORT' '=' port ',' 'DB' '=' dbName
| 'URL' '=' url) ',' 'USER' '=' user (',' 'PASSWORD' '=' password)? (','
```

```
propertiesDefinition)?')'

storageUnitName ::=
  identifier

hostname ::=
  string

port ::=
  int

dbName ::=
  string

url ::=
  string

user ::=
  string

password ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal

checkPrivileges ::=
  'CHECK_PRIVILEGES' '=' privilegeType (',' privilegeType)*

privilegeType ::=
  identifier
```

补充说明

- 修改存储单元前请确认已经在 Proxy 中创建逻辑数据库，并执行 `use` 命令选择一个逻辑数据库；
- `ALTER STORAGE UNIT` 不允许改变该存储单元关联的真实数据源（通过 host、port 和 db 判断）；
- `ALTER STORAGE UNIT` 会发生连接池的切换，这个操作可能对进行中的业务造成影响，请谨慎使用；
- 请确认修改的存储单元是可以正常连接的，否则将不能修改成功；

- PROPERTIES 为可选参数，用于自定义连接池属性，key 必须和连接池参数名一致;
- 可通过 CHECK_PRIVILEGES 指定注册时校验存储单元用户的权限，privilegeType 支持的类型有 SELECT、XA、PIPELINE、NONE，缺省值为 SELECT，当类型列表中包含 NONE 时，跳过权限校验。

示例

- 使用 HOST & PORT 方式修改存储单元

```
ALTER STORAGE UNIT ds_0 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="db_0",
    USER="root",
    PASSWORD="root"
);
```

- 使用 HOST & PORT 方式修改存储单元并设置连接池属性

```
ALTER STORAGE UNIT ds_1 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="db_1",
    USER="root",
    PASSWORD="root",
    PROPERTIES("maximumPoolSize"=10)
);
```

- 使用 URL 模式修改存储单元并设置连接池属性

```
ALTER STORAGE UNIT ds_2 (
    URL="jdbc:mysql://127.0.0.1:3306/db_2?serverTimezone=UTC&useSSL=false&
allowPublicKeyRetrieval=true",
    USER="root",
    PASSWORD="root",
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"=30000)
);
```

- 修改存储单元时检查 SELECT、XA 和 PIPELINE 权限

```
ALTER STORAGE UNIT ds_2 (
    URL="jdbc:mysql://127.0.0.1:3306/db_2?serverTimezone=UTC&useSSL=false&
allowPublicKeyRetrieval=true",
    USER="root",
    PASSWORD="root",
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"=30000)
), CHECK_PRIVILEGES=SELECT,XA,PIPELINE;
```

保留字

ALTER、STORAGE、UNIT、HOST、PORT、DB、USER、PASSWORD、PROPERTIES、URL、CHECK_PRIVILEGES

相关链接

- 保留字

UNREGISTER STORAGE UNIT

描述

UNREGISTER STORAGE UNIT 语法用于从当前逻辑库中移除存储单元。

语法

```
UnregisterStorageUnit ::=
  'UNREGISTER' 'STORAGE' 'UNIT' ifExists? storageUnitName (',' storageUnitName)*
ignoreTables?

ifExists ::=
  'IF' 'EXISTS'

storageUnitName ::=
  identifier

ignoreTables ::=
  'IGNORE' ('SINGLE')? (',')? ('BROADCAST')? 'TABLES'
```

补充说明

- UNREGISTER STORAGE UNIT 只会移除 Proxy 中的存储单元，不会删除与存储单元对应的真实数据源；
- 无法移除已经被规则使用的存储单元。移除被规则使用的存储单元时会提示 Storage unit are still in used；
- 将要移除的存储单元中仅包含 SINGLE RULE、BROADCAST RULE，且用户确认可以忽略该限制时，可添加 IGNORE SINGLE TABLES、IGNORE BROADCAST TABLES、IGNORE SINGLE, BROADCAST TABLES 关键字移除存储单元；
- ifExists 子句用于避免 Storage unit not exists 错误。

示例

- 移除存储单元

```
UNREGISTER STORAGE UNIT ds_0;
```

- 移除多个存储单元

```
UNREGISTER STORAGE UNIT ds_0, ds_1;
```

- 忽略单表移除存储单元

```
UNREGISTER STORAGE UNIT ds_0 IGNORE SINGLE TABLES;
```

- 忽略广播表移除存储单元

```
UNREGISTER STORAGE UNIT ds_0 IGNORE BROADCAST TABLES;
```

- 忽略单表和广播表移除存储单元

```
UNREGISTER STORAGE UNIT ds_0 IGNORE SINGLE, BROADCAST TABLES;
```

- 使用 `ifExists` 子句移除存储单元

```
UNREGISTER STORAGE UNIT IF EXISTS ds_0;
```

保留字

`UNREGISTER`、`STORAGE`、`UNIT`、`IF`、`EXISTS`、`IGNORE`、`SINGLE`、`BROADCAST`、`TABLES`

相关链接

- 保留字

规则定义

本章节将对规则定义的语法进行详细说明。

分片

本章节将对分片特性的语法进行详细说明。

CREATE SHARDING TABLE RULE

描述

`CREATE SHARDING TABLE RULE` 语法用于为当前所选逻辑库添加分片规则。

语法定义

```
CreateShardingTableRule ::=
  'CREATE' 'SHARDING' 'TABLE' 'RULE' ifNotExists? (tableRuleDefinition |
autoTableRuleDefinition) (',' (tableRuleDefinition | autoTableRuleDefinition))*

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

tableRuleDefinition ::=
  ruleName '(' 'DATANODES' '(' dataNode (',' dataNode)* ')' (','  'DATABASE_
STRATEGY' '(' strategyDefinition ')')? (','  'TABLE_STRATEGY' '('
strategyDefinition ')')? (','  'KEY_GENERATE_STRATEGY' '('
keyGenerateStrategyDefinition ')')? (',' 'AUDIT_STRATEGY' '('
auditStrategyDefinition ')')? ')'

autoTableRuleDefinition ::=
  ruleName '(' 'STORAGE_UNITS' '(' storageUnitName (',' storageUnitName)*  ')' ','
'SHARDING_COLUMN' '=' columnName ',' algorithmDefinition (',' 'KEY_GENERATE_
STRATEGY' '(' keyGenerateStrategyDefinition ')')? (',' 'AUDIT_STRATEGY' '('
auditStrategyDefinition ')')? ')'

strategyDefinition ::=
  'TYPE' '=' strategyType ',' ('SHARDING_COLUMN' | 'SHARDING_COLUMNS') '='
columnName ',' algorithmDefinition

keyGenerateStrategyDefinition ::=
  'KEY_GENERATE_STRATEGY' '(' 'COLUMN' '=' columnName ',' algorithmDefinition ')'

auditStrategyDefinition ::=
  'AUDIT_STRATEGY' '(' algorithmDefinition (',' algorithmDefinition)* ')'

algorithmDefinition ::=
  'TYPE' '(' 'NAME' '=' algorithmType (',' propertiesDefinition)?')'

propertiesDefinition ::=
```

```
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal

ruleName ::=
  identifier

dataNode ::=
  string

storageUnitName ::=
  identifier

columnName ::=
  identifier

strategyType ::=
  string

algorithmType ::=
  string
```

补充说明

- tableRuleDefinition 为标准分片规则定义；autoTableRuleDefinition 为自动分片规则定义。标准分片规则和自动分片规则可参考数据分片；
- 当使用标准分片时：
  - DATANODES 只能使用已经添加到当前逻辑库的资源，且只能使用 INLINE 表达式指定需要的资源；
  - DATABASE_STRATEGY、TABLE_STRATEGY 表示分库和分表策略，均为可选项，未配置时使用默认策略；
  - strategyDefinition 中属性 TYPE 用于指定分片算法的类型，目前仅支持 STANDARD 、COMPLEX。使用 COMPLEX 时需要用 SHARDING_COLUMNS 指定多个分片键。
- 当使用自动分片时：
  - STORAGE_UNITS 只能使用已经添加到当前逻辑库的资源，可通过枚举或 INLINE 表达式指定需要的资源；
  - 只能使用自动分片算法，可参考自动分片算法。
- algorithmType 为分片算法类型，分片算法类型请参考分片算法；

- 自动生成的算法命名规则为 tableName _ strategyType _ algorithmType；
- 自动生成的主键策略命名规则为 tableName _ strategyType；
- KEY_GENERATE_STRATEGY 用于指定主键生成策略，为可选项，关于主键生成策略可参考分布式主键；
- AUDIT_STRATEGY 用于指定分配审计生成策略，为可选项，关于分片审计生成策略可参考分片审计；
- ifNotExists 子句用于避免出现 Duplicate sharding rule 错误。

示例

1. 标准分片规则

```
CREATE SHARDING TABLE RULE t_order_item (
DATANODES("ds_${0..1}.t_order_item_${0..1}"),
DATABASE_STRATEGY(TYPE="standard",SHARDING_COLUMN=user_id,SHARDING_
ALGORITHM(TYPE(NAME="inline",PROPERTIES("algorithm-expression"="ds_${user_id % 2}
"))),
TABLE_STRATEGY(TYPE="standard",SHARDING_COLUMN=order_id,SHARDING_
ALGORITHM(TYPE(NAME="inline",PROPERTIES("algorithm-expression"="t_order_item_$
{order_id % 2}"))),
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME="snowflake")),
AUDIT_STRATEGY (TYPE(NAME="DML_SHARDING_CONDITIONS"),ALLOW_HINT_DISABLE=true)
);
```

2. 自动分片规则

```
CREATE SHARDING TABLE RULE t_order (
STORAGE_UNITS(ds_0,ds_1),
SHARDING_COLUMN=order_id,TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="4")),
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME="snowflake")),
AUDIT_STRATEGY (TYPE(NAME="DML_SHARDING_CONDITIONS"),ALLOW_HINT_DISABLE=true)
);
```

3. 使用 ifNotExists 子句创建分片规则

- 标准分片规则

```
CREATE SHARDING TABLE RULE IF NOT EXISTS t_order_item (
DATANODES("ds_${0..1}.t_order_item_${0..1}"),
DATABASE_STRATEGY(TYPE="standard",SHARDING_COLUMN=user_id,SHARDING_
ALGORITHM(TYPE(NAME="inline",PROPERTIES("algorithm-expression"="ds_${user_id % 2}
"))),
```

```
TABLE_STRATEGY(TYPE="standard",SHARDING_COLUMN=order_id,SHARDING_
ALGORITHM(TYPE(NAME="inline",PROPERTIES("algorithm-expression"="t_order_item_$
{order_id % 2}")))),
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME="snowflake")),
AUDIT_STRATEGY (TYPE(NAME="DML_SHARDING_CONDITIONS"),ALLOW_HINT_DISABLE=true)
);
```

- 自动分片规则

```
CREATE SHARDING TABLE RULE IF NOT EXISTS t_order (
STORAGE_UNITS(ds_0,ds_1),
SHARDING_COLUMN=order_id,TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="4")),
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME="snowflake")),
AUDIT_STRATEGY (TYPE(NAME="DML_SHARDING_CONDITIONS"),ALLOW_HINT_DISABLE=true)
);
```

保留字

CREATE、SHARDING、TABLE、RULE、DATANODES、DATABASE_STRATEGY、TABLE_STRATEGY、KEY_GENERATE_STRATEGY、STORAGE_UNITS、SHARDING_COLUMN、TYPE、SHARDING_COLUMN、KEY_GENERATOR、SHARDING_ALGORITHM、COLUMN、NAME、PROPERTIES、AUDIT_STRATEGY、AUDITORS、ALLOW_HINT_DISABLE

相关链接

- 保留字
- CREATE DEFAULT_SHARDING STRATEGY

ALTER SHARDING TABLE RULE

描述

ALTER SHARDING TABLE RULE 语法用于修改当前所选逻辑库的分片规则。

语法定义

```
AlterShardingTableRule ::=
  'ALTER' 'SHARDING' 'TABLE' 'RULE' (tableRuleDefinition | autoTableRuleDefinition)
(',' (tableRuleDefinition | autoTableRuleDefinition))*

tableRuleDefinition ::=
  ruleName '(' 'DATANODES' '(' dataNode (',' dataNode)* ')' (','  'DATABASE_
STRATEGY' '(' strategyDefinition ')')? (','  'TABLE_STRATEGY' '('
```

```
strategyDefinition ')')? (','  'KEY_GENERATE_STRATEGY' '('
keyGenerateStrategyDefinition ')')? (',' 'AUDIT_STRATEGY' '('
auditStrategyDefinition ')')? ')'

autoTableRuleDefinition ::=
  ruleName '(' 'STORAGE_UNITS' '(' storageUnitName (',' storageUnitName)*  ')' ','
'SHARDING_COLUMN' '=' columnName ',' algorithmDefinition (',' 'KEY_GENERATE_
STRATEGY' '(' keyGenerateStrategyDefinition ')')? (',' 'AUDIT_STRATEGY' '('
auditStrategyDefinition ')')? ')'

strategyDefinition ::=
  'TYPE' '=' strategyType ',' ('SHARDING_COLUMN' | 'SHARDING_COLUMNS') '='
columnName ',' algorithmDefinition

keyGenerateStrategyDefinition ::=
  'KEY_GENERATE_STRATEGY' '(' 'COLUMN' '=' columnName ',' algorithmDefinition ')'

auditStrategyDefinition ::=
  'AUDIT_STRATEGY' '(' algorithmDefinition (',' algorithmDefinition)* ')'

algorithmDefinition ::=
  'TYPE' '(' 'NAME' '=' algorithmType (',' propertiesDefinition)?')'

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal

ruleName ::=
  identifier

dataNode ::=
  string

storageUnitName ::=
  identifier

columnName ::=
  identifier

strategyType ::=
  string

algorithmType ::=
```

```
string
```

补充说明

- tableRuleDefinition 为标准分片规则定义；
- autoTableRuleDefinition 为自动分片规则定义。标准分片规则和自动分片规则可参考数据分片；
- 当使用标准分片时：
  - DATANODES 只能使用已经添加到当前逻辑库的资源，且只能使用 INLINE 表达式指定需要的资源；
  - DATABASE_STRATEGY、TABLE_STRATEGY 表示分库和分表策略，均为可选项，未配置时使用默认策略；
  - strategyDefinition 中属性 TYPE 用于指定分片算法的类型，目前仅支持 STANDARD 、COMPLEX。使用 COMPLEX 时需要用 SHARDING_COLUMNS 指定多个分片键。
- 当使用自动分片时：
  - STORAGE_UNITS 只能使用已经添加到当前逻辑库的资源，可通过枚举或 INLINE 表达式指定需要的资源；
  - 只能使用自动分片算法，可参考自动分片算法。
- algorithmType 为分片算法类型，分片算法类型请参考分片算法；
- 自动生成的算法命名规则为 tableName _ strategyType _ algorithmType；
- 自动生成的主键策略命名规则为 tableName _ strategyType；
- KEY_GENERATE_STRATEGY 用于指定主键生成策略，为可选项，关于主键生成策略可参考分布式主键。
- AUDIT_STRATEGY 用于指定分配审计生成策略，为可选项，关于分片审计生成策略可参考分片审计。

示例

1. 标准分片规则

```
ALTER SHARDING TABLE RULE t_order_item (
DATANODES("ds_${0..3}.t_order_item${0..3}"),
DATABASE_STRATEGY(TYPE="standard",SHARDING_COLUMN=user_id,SHARDING_
ALGORITHM(TYPE(NAME="inline",PROPERTIES("algorithm-expression"="ds_${user_id % 4}
")))),
TABLE_STRATEGY(TYPE="standard",SHARDING_COLUMN=order_id,SHARDING_
ALGORITHM(TYPE(NAME="inline",PROPERTIES("algorithm-expression"="t_order_item_$
{order_id % 4}")))),
```

```
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME="snowflake")),
AUDIT_STRATEGY(TYPE(NAME="dml_sharding_conditions"),ALLOW_HINT_DISABLE=true)
);
```

2. 自动分片规则

```
ALTER SHARDING TABLE RULE t_order (
STORAGE_UNITS(ds_0,ds_1,ds_2,ds_3),
SHARDING_COLUMN=order_id,TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="16")),
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME="snowflake")),
AUDIT_STRATEGY(TYPE(NAME="dml_sharding_conditions"),ALLOW_HINT_DISABLE=true)
);
```

保留字

ALTER、SHARDING、TABLE、RULE、DATANODES、DATABASE_STRATEGY、TABLE_STRATEGY、KEY_GENERATE_STRATEGY、STORAGE_UNITS、SHARDING_COLUMN、TYPE、SHARDING_COLUMN、KEY_GENERATOR、SHARDING_ALGORITHM、COLUMN、NAME、PROPERTIES、AUDIT_STRATEGY、AUDITORS、ALLOW_HINT_DISABLE

相关链接

- 保留字
- ALTER DEFAULT_SHARDING STRATEGY

DROP SHARDING TABLE RULE

描述

DROP SHARDING TABLE RULE 语法用于删除指定逻辑库的指定分片规则。

语法定义

```
DropShardingTableRule ::=
  'DROP' 'SHARDING' 'TABLE' 'RULE' ifExists? ruleName (',' ruleName)*  ('FROM'
databaseName)?

ifExists ::=
  'IF' 'EXISTS'

ruleName ::=
  identifier
```

```
databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected；
- ifExists 子句用于避免 Sharding rule not exists 错误。

示例

- 为指定逻辑库删除多个指定分片规则

```
DROP SHARDING TABLE RULE t_order, t_order_item FROM sharding_db;
```

- 为当前逻辑库删除单个指定分片规则

```
DROP SHARDING TABLE RULE t_order;
```

- 使用 ifExists 子句删除分片规则

```
DROP SHARDING TABLE RULE IF EXISTS t_order;
```

保留字

DROP、SHARDING、TABLE、RULE、FROM

相关链接

- 保留字

CREATE DEFAULT SHARDING STRATEGY

描述

CREATE DEFAULT SHARDING STRATEGY 语法用于创建默认的分片策略。

语法定义

```
CreateDefaultShardingStrategy ::=
  'CREATE' 'DEFAULT' 'SHARDING' ('DATABASE' | 'TABLE') 'STRATEGY' ifNotExists? '('
shardingStrategy ')'

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

shardingStrategy ::=
  'TYPE' '=' strategyType ',' ('SHARDING_COLUMN' '=' columnName | 'SHARDING_COLUMNS
' '=' columnNames) ',' 'SHARDING_ALGORITHM' '=' algorithmDefinition

strategyType ::=
  string

algorithmDefinition ::=
  'TYPE' '(' 'NAME' '=' algorithmType ',' propertiesDefinition ')'

columnNames ::=
  columnName (',' columnName)+

columnName ::=
  identifier

algorithmType ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal
```

补充说明

- 当使用复合分片算法时，需要通过 SHARDING_COLUMNS 指定多个分片键；
- algorithmType 为分片算法类型，详细的分片算法类型信息请参考分片算法；
- ifNotExists 子句用于避免出现 Duplicate default sharding strategy 错误。

示例

- 创建默认分表策略

```
CREATE DEFAULT SHARDING TABLE STRATEGY (
    TYPE="standard", SHARDING_COLUMN=user_id, SHARDING_ALGORITHM(TYPE(NAME=inline,
PROPERTIES("algorithm-expression"="t_order_${user_id % 2}")))
);
```

- 使用 ifNotExists 创建默认分表策略

```
CREATE DEFAULT SHARDING TABLE STRATEGY IF NOT EXISTS (
    TYPE="standard", SHARDING_COLUMN=user_id, SHARDING_ALGORITHM(TYPE(NAME=inline,
PROPERTIES("algorithm-expression"="t_order_${user_id % 2}")))
);
```

保留字

CREATE、DEFAULT、SHARDING、DATABASE、TABLE、STRATEGY、TYPE、SHARDING_COLUMN、SHARDING_COLUMNS、SHARDING_ALGORITHM、NAME、PROPERTIES

相关链接

- 保留字

ALTER DEFAULT SHARDING STRATEGY

描述

ALTER DEFAULT SHARDING STRATEGY 语法用于修改默认的分片策略。

语法定义

```
AlterDefaultShardingStrategy ::=
  'ALTER' 'DEFAULT' 'SHARDING' ('DATABASE' | 'TABLE') 'STRATEGY' '('
shardingStrategy ')'

shardingStrategy ::=
  'TYPE' '=' strategyType ',' ('SHARDING_COLUMN' '=' columnName | 'SHARDING_COLUMNS
' '=' columnNames) ',' 'SHARDING_ALGORITHM' '=' algorithmDefinition

strategyType ::=
  string

algorithmDefinition ::=
  'TYPE' '(' 'NAME' '=' algorithmType ',' propertiesDefinition ')'

columnNames ::=
  columnName (',' columnName)+

columnName ::=
  identifier

algorithmType ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal
```

补充说明

- 当使用复合分片算法时，需要通过 SHARDING_COLUMNS 指定多个分片键；
- algorithmType 为分片算法类型，详细的分片算法类型信息请参考分片算法。

示例

- 修改默认分表策略

```
ALTER DEFAULT SHARDING TABLE STRATEGY (
    TYPE="standard", SHARDING_COLUMN=user_id, SHARDING_ALGORITHM(TYPE(NAME=inline,
PROPERTIES("algorithm-expression"="t_order_${user_id % 2}")))
);
```

保留字

ALTER、DEFAULT、SHARDING、DATABASE、TABLE、STRATEGY、TYPE、SHARDING_COLUMN、SHARDING_COLUMNS、SHARDING_ALGORITHM、NAME、PROPERTIES

相关链接

- 保留字

DROP DEFAULT SHARDING STRATEGY

描述

DROP DEFAULT SHARDING STRATEGY 语法用于删除指定逻辑库的默认分片策略。

语法定义

```
DropDefaultShardingStrategy ::=
  'DROP' 'DEFAULT' 'SHARDING' ('TABLE' | 'DATABASE') 'STRATEGY' ifExists? ('FROM'
databaseName)?

ifExists ::=
  'IF' 'EXISTS'

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected;
- ifExists 子句用于避免 Default sharding strategy not exists 错误。

示例

- 为指定逻辑库删除默认表分片策略

```
DROP DEFAULT SHARDING TABLE STRATEGY FROM sharding_db;
```

- 为当前逻辑库删除默认库分片策略

```
DROP DEFAULT SHARDING DATABASE STRATEGY;
```

- 使用 ifExists 子句删除默认表分片策略

```
DROP DEFAULT SHARDING TABLE STRATEGY IF EXISTS;
```

- 使用 ifExists 子句删除默认库分片策略

```
DROP DEFAULT SHARDING DATABASE STRATEGY IF EXISTS;
```

保留字

DROP、DEFAULT、SHARDING、TABLE、DATABASE、STRATEGY、FROM ### 相关链接

- 保留字

DROP SHARDING KEY GENERATOR

描述

DROP SHARDING KEY GENERATOR 语法用于删除指定逻辑库的指定分片主键生成器。

语法定义

```
DropShardingKeyGenerator ::=
  'DROP' 'SHARDING' 'KEY' 'GENERATOR' ifExists? keyGeneratorName
(keyGeneratorName)* ('FROM' databaseName)?

ifExists ::=
  'IF' 'EXISTS'
```

```
keyGeneratorName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected；
- ifExists 子句用于避免 Sharding key generator not exists 错误。

示例

- 删除指定逻辑库的指定分片主键生成器

```
DROP SHARDING KEY GENERATOR t_order_snowflake FROM sharding_db;
```

- 删除当前逻辑库的指定分片主键生成器

```
DROP SHARDING KEY GENERATOR t_order_snowflake;
```

- 使用 ifExists 子句删除分片主键生成器

```
DROP SHARDING KEY GENERATOR IF EXISTS t_order_snowflake;
```

保留字

DROP、SHARDING、KEY、GENERATOR、FROM

相关链接

- 保留字

DROP SHARDING ALGORITHM

描述

DROP SHARDING ALGORITHM 语法用于删除指定逻辑库的指定分片算法。

语法定义

```
DropShardingAlgorithm ::=
  'DROP' 'SHARDING' 'ALGORITHM' algorithmName ifExists? ('FROM' databaseName)?

ifExists ::=
  'IF' 'EXISTS'

algorithmName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected；
- ifExists 子句用于避免 Sharding algorithm not exists 错误。

示例

- 删除指定逻辑库的指定分片算法

```
DROP SHARDING ALGORITHM t_order_hash_mod FROM sharding_db;
```

- 删除当前逻辑库的指定分片算法

```
DROP SHARDING ALGORITHM t_order_hash_mod;
```

- 使用 ifExists 子句删除分片算法

```
DROP SHARDING ALGORITHM IF EXISTS t_order_hash_mod;
```

保留字

DROP、SHARDING、ALGORITHM、FROM

相关链接

- 保留字

CREATE SHARDING TABLE REFERENCE RULE

描述

`CREATE SHARDING TABLE REFERENCE RULE` 语法用于为分片表创建关联规则。

语法定义

```
CreateShardingTableReferenceRule ::=
  'CREATE' 'SHARDING' 'TABLE' 'REFERENCE' 'RULE' ifNotExists?
referenceRelationshipDefinition  (',' referenceRelationshipDefinition)*

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

referenceRelationshipDefinition ::=
   ruleName '(' tableName (',' tableName)* ')'

ruleName ::=
  identifier

tableName ::=
  identifier
```

补充说明

- 只能为分片表创建关联关系；
- 一张分片表只能具有一个关联关系；
- 关联的分片表应分布在相同的存储单元，并且分片个数相同。例如 `ds_${0..1}.t_order_${0..1}` 与 `ds_${0..1}.t_order_item_${0..1}`；
- 关联的分片表应使用一致的分片算法。例如 `t_order_${order_id % 2}` 与 `t_order_item_${order_item_id % 2}`；
- `ifNotExists` 子句用于避免 `Duplicate sharding table reference rule` 错误。

示例

1. 创建关联关系

```
-- 创建关联关系之前需要先创建分片规则 t_order,t_order_item
CREATE SHARDING TABLE REFERENCE RULE ref_0 (t_order,t_order_item);
```

2. 创建多个关联关系

```
-- 创建关联关系之前需要先创建分片规则 t_order,t_order_item,t_product,t_product_item
CREATE SHARDING TABLE REFERENCE RULE ref_0 (t_order,t_order_item), ref_1 (t_
product,t_product_item);
```

3. 使用 `ifNotExists` 子句创建关联关系

```
CREATE SHARDING TABLE REFERENCE RULE IF NOT EXISTS ref_0 (t_order,t_order_item);
```

保留字

CREATE、SHARDING、TABLE、REFERENCE、RULE

相关链接

- 保留字
- CREATE SHARDING TABLE RULE

ALTER SHARDING TABLE REFERENCE RULE

描述

ALTER SHARDING TABLE REFERENCE RULE 语法用于修改分片表关联关系。

语法定义

```
AlterShardingTableReferenceRule ::=
  'ALTER' 'SHARDING' 'TABLE' 'REFERENCE' 'RULE'  referenceRelationshipDefinition  (
',' referenceRelationshipDefinition)*

referenceRelationshipDefinition ::=
  ruleName '(' tableName (',' tableName)* ')'
```

```
ruleName ::=
  identifier

tableName ::=
  identifier
```

补充说明

- 一张分片表只能具有一个关联关系；
- 关联的分片表应分布在相同的存储单元，并且分片个数相同。例如 `ds_${0..1}.t_order_${0..1}` 与 `ds_${0..1}.t_order_item_${0..1}`；
- 关联的分片表应使用一致的分片算法。例如 `t_order_${order_id % 2}` 与 `t_order_item_${order_item_id % 2}`；

示例

1. 修改关联关系

```
ALTER SHARDING TABLE REFERENCE RULE ref_0 (t_order,t_order_item);
```

2. 修改多个关联关系

```
ALTER SHARDING TABLE REFERENCE RULE ref_0 (t_order,t_order_item), ref_1 (t_product,
t_product_item);
```

保留字

ALTER、SHARDING、TABLE、REFERENCE、RULE

相关链接

- 保留字
- CREATE SHARDING TABLE RULE

DROP SHARDING TABLE REFERENCE RULE

描述

DROP SHARDING TABLE REFERENCE RULE 语法用删除指定的关联规则。

语法定义

```
DropShardingTableReferenceRule ::=
  'DROP' 'SHARDING' 'TABLE' 'REFERENCE' 'RULE' ifExists? ruleName (',' ruleName)*

ifExists ::=
  'IF' 'EXISTS'

ruleName ::=
  identifier
```

补充说明

- ifExists 子句用于避免 Sharding reference rule not exists 错误。

示例

- 删除单个关联规则

```
DROP SHARDING TABLE REFERENCE RULE ref_0;
```

- 删除多个关联规则

```
DROP SHARDING TABLE REFERENCE RULE ref_0, ref_1;
```

- 使用 ifExists 子句删除关联规则

```
DROP SHARDING TABLE REFERENCE RULE IF EXISTS ref_0;
```

保留字

DROP、SHARDING、TABLE、REFERENCE、RULE

相关链接

- 保留字

广播表

本章节将对广播表特性的语法进行详细说明。

CREATE BROADCAST TABLE RULE

描述

`CREATE BROADCAST TABLE RULE` 语法用于为需要广播的表（广播表）创建广播规则。

语法定义

```
CreateBroadcastTableRule ::=
  'CREATE' 'BROADCAST' 'TABLE' 'RULE' ifNotExists? tableName (',' tableName)*

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

tableName ::=
  identifier
```

补充说明

- `tableName` 可使用已经存在的表或者将要创建的表;
- `ifNotExists` 子句用于避免 `Duplicate Broadcast rule` 错误。

示例

创建广播规则

```
-- 将 t_province, t_city 添加到广播规则中
CREATE BROADCAST TABLE RULE t_province, t_city;
```

使用 ifNotExists 子句创建广播规则

```
CREATE BROADCAST TABLE RULE IF NOT EXISTS t_province, t_city;
```

保留字

CREATE、BROADCAST、TABLE、RULE

相关链接

- 保留字

DROP BROADCAST TABLE RULE

描述

DROP BROADCAST TABLE RULE 语法用于为指定的广播表删除广播规则。

语法定义

```
DropBroadcastTableRule ::=
  'DROP' 'BROADCAST' 'TABLE' 'RULE' ifExists? tableName (',' tableName)*

ifExists ::=
  'IF' 'EXISTS'

tableName ::=
  identifier
```

补充说明

- tableName 可使用已经存在的广播规则的表;
- ifExists 子句用于避免 Broadcast rule not exists 错误。

示例

- 为指定广播表删除广播规则

```
DROP BROADCAST TABLE RULE t_province, t_city;
```

- 使用 ifExists 子句为广播表删除广播规则

```
DROP BROADCAST TABLE RULE IF EXISTS t_province, t_city;
```

保留字

DROP、BROADCAST、TABLE、RULE

相关链接

- 保留字

单表

本章节将对单表特性的语法进行详细说明。

LOAD SINGLE TABLE

描述

LOAD SINGLE TABLE 用于加载单表。

语法定义

```
loadSingleTable ::=
  'LOAD' 'SINGLE' 'TABLE' tableDefinition

tableDefinition ::=
  tableIdentifier (',' tableIdentifier)*

tableIdentifier ::=
  '*.*' | '*.*.*' | storageUnitName '.*' | storageUnitName '.*.*' | storageUnitName
'.' schemaName '.*' | storageUnitName '.' tableName | storageUnitName '.'
schemaName '.' tableName

storageUnitName ::=
  identifier
```

```
schemaName ::=
  identifier

tableName ::=
  identifier
```

补充说明

- PostgreSQL 和 OpenGauss 协议下支持指定 schemaName

示例

- 加载指定单表

```
LOAD SINGLE TABLE ds_0.t_single;
```

- 加载指定存储节点中的全部单表

```
LOAD SINGLE TABLE ds_0.*;
```

- 加载全部单表

```
LOAD SINGLE TABLE *.*;
```

保留字

`LOAD`、`SINGLE`、`TABLE`

相关链接

- 保留字

UNLOAD SINGLE TABLE

描述

`UNLOAD SINGLE TABLE` 用于卸载单表。

语法定义

```
unloadSingleTable ::=
  'UNLOAD' 'SINGLE' 'TABLE' tableNames

tableNames ::=
  tableName (',' tableName)*

tableName ::=
  identifier
```

补充说明

- 与加载不同，卸载单表时仅需指定表名

示例

- 卸载指定单表

```
UNLOAD SINGLE TABLE t_single;
```

- 卸载全部单表

```
UNLOAD SINGLE TABLE *;
-- 或
UNLOAD ALL SINGLE TABLES;
```

保留字

`UNLOAD`、`SINGLE`、`TABLE`、`ALL`、`TABLES`

相关链接

- 保留字

SET DEFAULT SINGLE TABLE STORAGE UNIT

描述

`SET DEFAULT SINGLE TABLE STORAGE UNIT` 语法用于设置默认的单表存储单元。

语法定义

```
SetDefaultSingleTableStorageUnit ::=
  'SET' 'DEFAULT' 'SINGLE' 'TABLE' 'STORAGE' 'UNIT' singleTableDefinition

singleTableDefinition ::=
  '=' (storageUnitName | 'RANDOM')

storageUnitName ::=
  identifier
```

补充说明

- STORAGE UNIT 需使用 RDL 管理的存储单元。RANDOM 代表随机储存

示例

- 设置默认的单表存储单元

```
SET DEFAULT SINGLE TABLE STORAGE UNIT = ds_0;
```

- 设置默认的单表存储单元为随机储存

```
SET DEFAULT SINGLE TABLE STORAGE UNIT = RANDOM;
```

保留字

SET、DEFAULT、SINGLE、TABLE、STORAGE、UNIT、RANDOM

相关链接

- 保留字

读写分离

本章节将对读写分离特性的语法进行详细说明。

CREATE READWRITE_SPLITTING RULE

描述

。CREATE READWRITE_SPLITTING RULE 语法用于创建读写分离规则。

语法定义

```
CreateReadwriteSplittingRule ::=
  'CREATE' 'READWRITE_SPLITTING' 'RULE' ifNotExists? readwriteSplittingDefinition (
',' readwriteSplittingDefinition)*

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

readwriteSplittingDefinition ::=
  ruleName '(' dataSourceDefinition (',' transactionalReadQueryStrategyDefinition)?
(',' loadBalancerDefinition)? ')'

dataSourceDefinition ::=
    'WRITE_STORAGE_UNIT' '=' writeStorageUnitName ',' 'READ_STORAGE_UNITS' '('
storageUnitName (',' storageUnitName)* ')'

transactionalReadQueryStrategyDefinition ::=
    'TRANSACTIONAL_READ_QUERY_STRATEGY' '=' transactionalReadQueryStrategyType

loadBalancerDefinition ::=
    'TYPE' '(' 'NAME' '=' algorithmType (',' propertiesDefinition)? ')'

ruleName ::=
  identifier

writeStorageUnitName ::=
  identifier

storageUnitName ::=
  identifier

transactionalReadQueryStrategyType ::=
  string

algorithmType ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'
```

```
key ::=
  string

value ::=
  literal
```

补充说明

- transactionalReadQueryStrategyType 指定事务内读请求路由策略，请参考YAML 配置；
- algorithmType 指定负载均衡算法类型，请参考负载均衡算法；
- 重复的 ruleName 将无法被创建；
- ifNotExists 子句用于避免出现 Duplicate readwrite_splitting rule 错误。

示例

创建读写分离规则

```
CREATE READWRITE_SPLITTING RULE ms_group_0 (
    WRITE_STORAGE_UNIT=write_ds,
    READ_STORAGE_UNITS(read_ds_0,read_ds_1),
    TYPE(NAME="random")
);
```

使用 ifNotExists 子句创建读写分离规则

- 读写分离规则

```
CREATE READWRITE_SPLITTING RULE IF NOT EXISTS ms_group_0 (
    WRITE_STORAGE_UNIT=write_ds,
    READ_STORAGE_UNITS(read_ds_0,read_ds_1),
    TYPE(NAME="random")
);
```

保留字

CREATE、READWRITE_SPLITTING、RULE、WRITE_STORAGE_UNIT、READ_STORAGE_UNITS 、TYPE、NAME、PROPERTIES、TRUE、FALSE

相关链接

- 保留字
- 负载均衡算法

ALTER READWRITE_SPLITTING RULE

描述

`ALTER READWRITE_SPLITTING RULE` 语法用于修改读写分离规则。

语法定义

```
AlterReadwriteSplittingRule ::=
  'ALTER' 'READWRITE_SPLITTING' 'RULE' readwriteSplittingDefinition (','
readwriteSplittingDefinition)*

readwriteSplittingDefinition ::=
  ruleName '(' dataSourceDefinition (',' transactionalReadQueryStrategyDefinition)?
(',' loadBalancerDefinition)? ')'

dataSourceDefinition ::=
    'WRITE_STORAGE_UNIT' '=' writeStorageUnitName ',' 'READ_STORAGE_UNITS' '('
storageUnitName (',' storageUnitName)* ')'

transactionalReadQueryStrategyDefinition ::=
    'TRANSACTIONAL_READ_QUERY_STRATEGY' '=' transactionalReadQueryStrategyType

loadBalancerDefinition ::=
    'TYPE' '(' 'NAME' '=' algorithmType (',' propertiesDefinition)? ')'

ruleName ::=
  identifier

writeStorageUnitName ::=
  identifier

storageUnitName ::=
  identifier

transactionalReadQueryStrategyType ::=
  string

algorithmType ::=
  string
```

```
propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal
```

补充说明

- transactionalReadQueryStrategyType 指定事务内读请求路由策略，请参考YAML 配置；
- algorithmType 指定负载均衡算法类型，请参考负载均衡算法。

示例

修改读写分离规则

```
ALTER READWRITE_SPLITTING RULE ms_group_0 (
    WRITE_STORAGE_UNIT=write_ds,
    READ_STORAGE_UNITS(read_ds_0,read_ds_1),
    TYPE(NAME="random")
);
```

保留字

ALTER、READWRITE_SPLITTING、RULE、WRITE_STORAGE_UNIT、READ_STORAGE_UNITS 、TYPE、NAME、PROPERTIES、TRUE、FALSE

相关链接

- 保留字
- 负载均衡算法

DROP READWRITE_SPLITTING RULE

描述

DROP READWRITE_SPLITTING RULE 语法用于为指定逻辑库删除读写分离。

语法定义

```
DropReadwriteSplittingRule ::=
  'DROP' 'READWRITE_SPLITTING' 'RULE' ifExists? ruleName (',' ruleName)* ('FROM'
databaseName)?

ifExists ::=
  'IF' 'EXISTS'

ruleName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected;
- ifExists 子句用于避免 Readwrite-splitting rule not exists 错误。

示例

- 为指定逻辑库删除读写分离规则

```
DROP READWRITE_SPLITTING RULE ms_group_1 FROM readwrite_splitting_db;
```

- 为当前逻辑库删除读写分离规则

```
DROP READWRITE_SPLITTING RULE ms_group_1;
```

- 使用 ifExists 子句删除读写分离规则

```
DROP READWRITE_SPLITTING RULE IF EXISTS ms_group_1;
```

保留字

DROP、READWRITE_SPLITTING、RULE

相关链接

- 保留字

数据加密

本章节将对数据加密特性的语法进行详细说明。

CREATE ENCRYPT RULE

描述

CREATE ENCRYPT RULE 语法用于创建数据加密规则。

语法定义

```
CreateEncryptRule ::=
  'CREATE' 'ENCRYPT' 'RULE' ifNotExists? encryptDefinition (',' encryptDefinition)*

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

encryptDefinition ::=
  ruleName '(' 'COLUMNS' '(' columnDefinition (',' columnDefinition)*  ')' ')'

columnDefinition ::=
  '(' 'NAME' '=' columnName ',' 'CIPHER' '=' cipherColumnName (',' 'ASSISTED_QUERY'
'=' assistedQueryColumnName)? (',' 'LIKE_QUERY' '=' likeQueryColumnName)? ','
encryptAlgorithmDefinition (',' assistedQueryAlgorithmDefinition)? (','
likeQueryAlgorithmDefinition)? ')'

encryptAlgorithmDefinition ::=
  'ENCRYPT_ALGORITHM' '(' 'TYPE' '(' 'NAME' '=' algorithmType (','
propertiesDefinition)? ')'

assistedQueryAlgorithmDefinition ::=
  'ASSISTED_QUERY_ALGORITHM' '(' 'TYPE' '(' 'NAME' '=' algorithmType (','
propertiesDefinition)? ')'

likeQueryAlgorithmDefinition ::=
  'LIKE_QUERY_ALGORITHM' '(' 'TYPE' '(' 'NAME' '=' algorithmType (','
```

```
propertiesDefinition)? ')'

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

ruleName ::=
  identifier

columnName ::=
  identifier

cipherColumnName ::=
  identifier

assistedQueryColumnName ::=
  identifier

likeQueryColumnName ::=
  identifier

algorithmType ::=
  string

key ::=
  string

value ::=
  literal
```

补充说明

- CIPHER 指定密文数据列，ASSISTED_QUERY 指定辅助查询列，LIKE_QUERY 指定模糊查询列；
- algorithmType 指定加密算法类型，请参考 加密算法；
- 重复的 ruleName 将无法被创建；
- ifNotExists 子句用于避免出现 Duplicate encrypt rule 错误。

示例

创建数据加密规则

```
CREATE ENCRYPT RULE t_encrypt (
COLUMNS(
(NAME=user_id,CIPHER=user_cipher,ASSISTED_QUERY=assisted_query_user,LIKE_
QUERY=like_query_user,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value'=
```

```
'123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_ALGORITHM(TYPE(NAME=
'MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE'))),
(NAME=order_id,CIPHER =order_cipher,ASSISTED_QUERY=assisted_query_order,LIKE_
QUERY=like_query_order,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value
'='123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_
ALGORITHM(TYPE(NAME='MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE')))
)),
t_encrypt_2 (
COLUMNS(
(NAME=user_id,CIPHER=user_cipher,ASSISTED_QUERY=assisted_query_user,LIKE_
QUERY=like_query_user,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value'=
'123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_ALGORITHM(TYPE(NAME=
'MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE'))),
(NAME=order_id, CIPHER=order_cipher,ASSISTED_QUERY=assisted_query_order,LIKE_
QUERY=like_query_order,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value
'='123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_
ALGORITHM(TYPE(NAME='MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE')))
));
```

使用 `ifNotExists` 子句创建数据加密规则

```
CREATE ENCRYPT RULE IF NOT EXISTS t_encrypt (
COLUMNS(
(NAME=user_id,CIPHER=user_cipher,ASSISTED_QUERY=assisted_query_user,LIKE_
QUERY=like_query_user,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value'=
'123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_ALGORITHM(TYPE(NAME=
'MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE'))),
(NAME=order_id,CIPHER =order_cipher,ASSISTED_QUERY=assisted_query_order,LIKE_
QUERY=like_query_order,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value
'='123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_
ALGORITHM(TYPE(NAME='MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE')))
)),
t_encrypt_2 (
COLUMNS(
(NAME=user_id,CIPHER=user_cipher,ASSISTED_QUERY=assisted_query_user,LIKE_
QUERY=like_query_user,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value'=
'123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_ALGORITHM(TYPE(NAME=
'MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE'))),
(NAME=order_id,CIPHER=order_cipher,ASSISTED_QUERY=assisted_query_order,LIKE_
QUERY=like_query_order,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value
'='123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_
ALGORITHM(TYPE(NAME='MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE')))
));
```

保留字

CREATE、ENCRYPT、RULE、COLUMNS、NAME、CIPHER、ASSISTED_QUERY、LIKE_QUERY、ENCRYPT_ALGORITHM、ASSISTED_QUERY_ALGORITHM、LIKE_QUERY_ALGORITHM、TYPE、TRUE、FALSE

相关链接

- 保留字
- 加密算法

ALTER ENCRYPT RULE

说明

ALTER ENCRYPT RULE 语法用于修改加密规则。

语法

```
AlterEncryptRule ::=
  'ALTER' 'ENCRYPT' 'RULE' encryptDefinition (',' encryptDefinition)*

encryptDefinition ::=
  ruleName '(' 'COLUMNS' '(' columnDefinition (',' columnDefinition)*  ')' ')'

columnDefinition ::=
  '(' 'NAME' '=' columnName ',' 'CIPHER' '=' cipherColumnName (',' 'ASSISTED_QUERY'
'=' assistedQueryColumnName)? (',' 'LIKE_QUERY' '=' likeQueryColumnName)? ','
encryptAlgorithmDefinition (',' assistedQueryAlgorithmDefinition)? (','
likeQueryAlgorithmDefinition)? ')'

encryptAlgorithmDefinition ::=
  'ENCRYPT_ALGORITHM' '(' 'TYPE' '(' 'NAME' '=' algorithmType (','
propertiesDefinition)? ')'

assistedQueryAlgorithmDefinition ::=
  'ASSISTED_QUERY_ALGORITHM' '(' 'TYPE' '(' 'NAME' '=' algorithmType (','
propertiesDefinition)? ')'

likeQueryAlgorithmDefinition ::=
  'LIKE_QUERY_ALGORITHM' '(' 'TYPE' '(' 'NAME' '=' algorithmType (','
propertiesDefinition)? ')'

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'
```

```
ruleName ::=
  identifier

columnName ::=
  identifier

cipherColumnName ::=
  identifier

assistedQueryColumnName ::=
  identifier

likeQueryColumnName ::=
  identifier

algorithmType ::=
  string

key ::=
  string

value ::=
  literal
```

补充说明

- CIPHER 指定密文数据列，ASSISTED_QUERY 指定辅助查询列，LIKE_QUERY 指定模糊查询列；
- algorithmType 指定加密算法类型，请参考 加密算法；
- 重复的 ruleName 将无法被创建。

示例

- 修改加密规则

```
ALTER ENCRYPT RULE t_encrypt (
COLUMNS(
(NAME=user_id,CIPHER=user_cipher,ASSISTED_QUERY=assisted_query_user,LIKE_
QUERY=like_query_user,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value'=
'123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_ALGORITHM(TYPE(NAME=
'MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE'))),
(NAME=order_id,CIPHER=order_cipher,ASSISTED_QUERY=assisted_query_order,LIKE_
QUERY=like_query_order,ENCRYPT_ALGORITHM(TYPE(NAME='AES',PROPERTIES('aes-key-value
'='123456abc', 'digest-algorithm-name'='SHA-1'))),ASSISTED_QUERY_
```

```
ALGORITHM(TYPE(NAME='MD5')),LIKE_QUERY_ALGORITHM(TYPE(NAME='CHAR_DIGEST_LIKE')))
));
```

保留字

ALTER、ENCRYPT、RULE、COLUMNS、NAME、CIPHER、ASSISTED_QUERY、LIKE_QUERY、ENCRYPT_ALGORITHM、ASSISTED_QUERY_ALGORITHM、LIKE_QUERY_ALGORITHM、TYPE、TRUE、FALSE

相关链接

- 保留字
- 加密算法

DROP ENCRYPT RULE

说明

DROP ENCRYPT RULE 语法用于删除加密规则。

语法

```
DropEncryptRule ::=
  'DROP' 'ENCRYPT' 'RULE' ifExists? ruleName (',' ruleName)*

ifExists ::=
  'IF' 'EXISTS'

ruleName ::=
  identifier
```

补充说明

- ifExists 子句用于避免 Encrypt rule not exists 错误。

示例

- 删除加密规则

```
DROP ENCRYPT RULE t_encrypt, t_encrypt_2;
```

- 使用 ifExists 删除加密规则

```
DROP ENCRYPT RULE IF EXISTS t_encrypt, t_encrypt_2;
```

保留字

DROP, ENCRYPT, RULE

相关链接

- 保留字

数据脱敏

本章节将对数据脱敏的语法进行详细说明。

CREATE MASK RULE

描述

CREATE MASK RULE 语法用于创建数据脱敏规则。

语法定义

```
CreateMaskRule ::=
  'CREATE' 'MASK' 'RULE' ifNotExists? maskRuleDefinition (',' maskRuleDefinition)*

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

maskRuleDefinition ::=
  ruleName '(' 'COLUMNS' '(' columnDefinition (',' columnDefinition)* ')' ')'

columnDefinition ::=
  '(' 'NAME' '=' columnName ',' maskAlgorithmDefinition ')'

maskAlgorithmDefinition ::=
  'TYPE' '(' 'NAME' '=' algorithmType (',' propertiesDefinition)? ')'
```

```
propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

ruleName ::=
  identifier

columnName ::=
  identifier

algorithmType ::=
  literal

key ::=
  string

value ::=
  literal
```

补充说明

- `algorithmType` 指定数据脱敏算法类型，请参考 数据脱敏算法；
- 重复的 `ruleName` 将无法被创建；
- `ifNotExists` 子句用于避免出现 `Duplicate mask rule` 错误。

示例

创建数据脱敏规则

```
CREATE MASK RULE t_mask (
COLUMNS(
(NAME=phone_number,TYPE(NAME='MASK_FROM_X_TO_Y', PROPERTIES("from-x"=1, "to-y"=2,
"replace-char"="*"))),
(NAME=address,TYPE(NAME='MD5'))
));
```

使用 ifNotExists 子句创建数据脱敏规则

```
CREATE MASK RULE IF NOT EXISTS t_mask (
COLUMNS(
(NAME=phone_number,TYPE(NAME='MASK_FROM_X_TO_Y', PROPERTIES("from-x"=1, "to-y"=2,
"replace-char"="*"))),
(NAME=address,TYPE(NAME='MD5'))
));
```

保留字

CREATE、MASK、RULE、COLUMNS、NAME、TYPE

相关链接

- 保留字
- 数据脱敏算法

ALTER MASK RULE

描述

ALTER MASK RULE 语法用于修改数据脱敏规则。

语法定义

```
AlterMaskRule ::=
  'ALTER' 'MASK' 'RULE' maskRuleDefinition (',' maskRuleDefinition)*

maskRuleDefinition ::=
  ruleName '(' 'COLUMNS' '(' columnDefinition (',' columnDefinition)* ')' ')'

columnDefinition ::=
  '(' 'NAME' '=' columnName ',' maskAlgorithmDefinition ')'

maskAlgorithmDefinition ::=
  'TYPE' '(' 'NAME' '=' algorithmType (',' propertiesDefinition)? ')'

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

ruleName ::=
  identifier
```

```
columnName ::=
  identifier

algorithmType ::=
  literal

key ::=
  string

value ::=
  literal
```

补充说明

- `algorithmType` 指定数据脱敏算法类型，请参考 数据脱敏算法。

示例

修改数据脱敏规则

```
ALTER MASK RULE t_mask (
COLUMNS(
(NAME=phone_number,TYPE(NAME='MASK_FROM_X_TO_Y', PROPERTIES("from-x"=1, "to-y"=2,
"replace-char"="*"))),
(NAME=address,TYPE(NAME='MD5'))
));
```

保留字

`ALTER`、`MASK`、`RULE`、`COLUMNS`、`NAME`、`TYPE`

相关链接

- 保留字
- 数据脱敏算法

DROP MASK RULE

说明

DROP MASK RULE 语法用于删除数据脱敏规则。

语法

```
DropMaskRule ::=
  'DROP' 'MASK' 'RULE' ifExists? ruleName (',' ruleName)*

ifExists ::=
  'IF' 'EXISTS'

ruleName ::=
  identifier
```

补充说明

- ifExists 子句用于避免 Mask rule not exists 错误。

示例

- 删除数据脱敏规则

```
DROP MASK RULE t_mask, t_mask_1;
```

- 使用 ifExists 子句删除数据脱敏规则

```
DROP MASK RULE IF EXISTS t_mask, t_mask_1;
```

保留字

DROP, MASK, RULE

相关链接

- 保留字

影子库压测

本章节将对影子库压测特性的语法进行详细说明。

CREATE SHADOW RULE

描述

`CREATE SHADOW RULE` 语法用于创建影子库压测规则。

语法定义

```
CreateShadowRule ::=
  'CREATE' 'SHADOW' 'RULE' ifNotExists? shadowRuleDefinition (','
shadowRuleDefinition)*

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

shadowRuleDefinition ::=
  ruleName '(' storageUnitMapping shadowTableRule (',' shadowTableRule)* ')'

storageUnitMapping ::=
  'SOURCE' '=' storageUnitName ',' 'SHADOW' '=' storageUnitName

shadowTableRule ::=
  tableName '(' shadowAlgorithm ')'

shadowAlgorithm ::=
  'TYPE' '(' 'NAME' '=' algorithmType ',' propertiesDefinition ')'

ruleName ::=
  identifier

storageUnitName ::=
  identifier

tableName ::=
  identifier

algorithmName ::=
  identifier

algorithmType ::=
  string
```

```
propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal
```

补充说明

- 重复的 ruleName 无法被创建；
- storageUnitMapping 指定源数据库和影子库的映射关系，需使用 RDL 管理的 STORAGE UNIT ，请参考 存储单元；
- shadowAlgorithm 可同时作用于多个 shadowTableRule；
- algorithmName 会根据 ruleName、tableName 和 algorithmType 自动生成；
- algorithmType 目前支持 VALUE_MATCH、REGEX_MATCH 和 SQL_HINT；
- ifNotExists 子句用于避免出现 Duplicate shadow rule 错误。

示例

- 创建影子库压测规则

```
CREATE SHADOW RULE shadow_rule(
  SOURCE=demo_ds,
  SHADOW=demo_ds_shadow,
  t_order(TYPE(NAME="SQL_HINT")),
  t_order_item(TYPE(NAME="VALUE_MATCH", PROPERTIES("operation"="insert","column"=
"user_id", "value"='1')))
);
```

- 使用 ifNotExists 子句创建影子库压测规则

```
CREATE SHADOW RULE IF NOT EXISTS shadow_rule(
  SOURCE=demo_ds,
  SHADOW=demo_ds_shadow,
  t_order(TYPE(NAME="SQL_HINT")),
  t_order_item(TYPE(NAME="VALUE_MATCH", PROPERTIES("operation"="insert","column"=
"user_id", "value"='1')))
);
```

保留字

CREATE、SHADOW、RULE、SOURCE、SHADOW、TYPE、NAME、PROPERTIES

相关链接

- 保留字
- 存储单元

ALTER SHADOW RULE

描述

ALTER SHADOW RULE 语法用于修改影子库压测规则。

语法定义

```
AlterShadowRule ::=
  'ALTER' 'SHADOW' 'RULE' shadowRuleDefinition (',' shadowRuleDefinition)*

shadowRuleDefinition ::=
  ruleName '(' storageUnitMapping shadowTableRule (',' shadowTableRule)* ')'

storageUnitMapping ::=
  'SOURCE' '=' storageUnitName ',' 'SHADOW' '=' storageUnitName

shadowTableRule ::=
  tableName '(' shadowAlgorithm ')'

shadowAlgorithm ::=
  'TYPE' '(' 'NAME' '=' shadowAlgorithmType ',' propertiesDefinition ')'

ruleName ::=
  identifier

storageUnitName ::=
  identifier

tableName ::=
  identifier

algorithmName ::=
  identifier

shadowAlgorithmType ::=
```

```
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal
```

补充说明

- storageUnitMapping 指定源数据库和影子库的映射关系，需使用 RDL 管理的 STORAGE UNIT，请参考 存储单元；
- shadowAlgorithm 可同时作用于多个 shadowTableRule；
- algorithmName 会根据 ruleName、tableName 和 shadowAlgorithmType 自动生成；
- shadowAlgorithmType 目前支持 VALUE_MATCH、REGEX_MATCH 和 SQL_HINT。

示例

- 修改影子库压测规则

```
ALTER SHADOW RULE shadow_rule(
  SOURCE=demo_ds,
  SHADOW=demo_ds_shadow,
  t_order(TYPE(NAME="SQL_HINT")),
  t_order_item(TYPE(NAME="VALUE_MATCH", PROPERTIES("operation"="insert","column"=
"user_id", "value"='1')))
);
```

保留字

ALTER、SHADOW、RULE、SOURCE、SHADOW、TYPE、NAME、PROPERTIES

相关链接

- 保留字
- 存储单元

DROP SHADOW RULE

描述

`DROP SHADOW RULE` 语法用于为指定逻辑库删除影子库压测规则。

语法定义

```
DropShadowRule ::=
  'DROP' 'SHADOW' 'RULE' ifExists? ruleName ('FROM' databaseName)?

ifExists ::=
  'IF' 'EXISTS'

ruleName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`；
- `ifExists` 子句用于避免 `Shadow rule not exists` 错误。

示例

- 为指定数据库删除影子库压测规则

```
DROP SHADOW RULE shadow_rule FROM shadow_db;
```

- 为当前数据库删除影子库压测规则

```
DROP SHADOW RULE shadow_rule;
```

- 使用 `ifExists` 子句删除影子库压测规则

```
DROP SHADOW RULE IF EXISTS shadow_rule;
```

保留字

DROP、SHADOW、RULE、FROM

相关链接

- 保留字

CREATE DEFAULT SHADOW ALGORITHM

描述

CREATE DEFAULT SHADOW ALGORITHM 语法用于创建影子库默认算法规则。

语法定义

```
CreateDefaultShadowAlgorithm ::=
  'CREATE' 'DEFAULT' 'SHADOW' 'ALGORITHM' ifNotExists? shadowAlgorithm

ifNotExists ::=
  'IF' 'NOT' 'EXISTS'

shadowAlgorithm ::=
  'TYPE' '(' 'NAME' '=' algorithmType ',' propertiesDefiinition ')'

algorithmType ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal
```

补充说明

- algorithmType 目前支持 VALUE_MATCH、REGEX_MATCH 和 SQL_HINT;
- ifNotExists 子句用于避免出现 Duplicate default shadow algorithm 错误。

示例

- 创建默认影子库压测算法

```
CREATE DEFAULT SHADOW ALGORITHM TYPE(NAME="SQL_HINT");
```

- 使用 ifNotExists 子句创建默认影子库压测算法

```
CREATE DEFAULT SHADOW ALGORITHM IF NOT EXISTS TYPE(NAME="SQL_HINT");
```

保留字

CREATE、DEFAULT、SHADOW、ALGORITHM、TYPE、NAME、PROPERTIES

相关链接

- 保留字

ALTER DEFAULT SHADOW ALGORITHM

描述

ALTER DEFAULT SHADOW ALGORITHM 语法用于修改影子库默认算法规则。

语法定义

```
AlterDefaultShadowAlgorithm ::=
  'ALTER' 'DEFAULT' 'SHADOW' 'ALGORITHM' shadowAlgorithm

shadowAlgorithm ::=
  'TYPE' '(' 'NAME' '=' algorithmType ',' propertiesDefiinition ')'

algorithmType ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'
```

```
key ::=
  string

value ::=
  literal
```

补充说明

- algorithmType 目前支持 VALUE_MATCH、REGEX_MATCH 和 SQL_HINT。

示例

- 修改默认影子库压测算法

```
ALTER DEFAULT SHADOW ALGORITHM TYPE(NAME="SQL_HINT");
```

保留字

ALTER、DEFAULT、SHADOW、ALGORITHM、TYPE、NAME、PROPERTIES

相关链接

- 保留字

DROP DEFAULT SHADOW ALGORITHM

描述

DROP DEFAULT SHADOW ALGORITHM 语法用于为指定逻辑库删除默认影子库压测算法。

语法定义

```
DropDefaultShadowAlgorithm ::=
  'DROP' 'DEFAULT' 'SHADOW' 'ALGORITHM' ifExists? ('FROM' databaseName)?

ifExists ::=
  'IF' 'EXISTS'

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected;
- ifExists 子句用于避免 Default shadow algorithm not exists 错误。

示例

- 为指定数据库删除默认影子库压测算法

```
DROP DEFAULT SHADOW ALGORITHM FROM shadow_db;
```

- 为当前数据库删除默认影子库压测算法

```
DROP DEFAULT SHADOW ALGORITHM;
```

- 使用 ifExists 子句删除默认影子库压测算法

```
DROP DEFAULT SHADOW ALGORITHM IF EXISTS;
```

保留字

DROP、DEFAULT、SHADOW、ALGORITHM、FROM

相关链接

- 保留字

DROP SHADOW ALGORITHM

描述

DROP SHADOW ALGORITHM 语法用于为指定逻辑库删除影子库压测算法。

语法定义

```
DropShadowAlgorithm ::=
  'DROP' 'SHADOW' 'ALGORITHM' ifExists? algorithmName (',' algorithmName)* ('FROM'
databaseName)?

ifExists ::=
  'IF' 'EXISTS'
```

```
algorithmName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected；
- ifExists 子句用于避免 shadow algorithm not exists 错误。

示例

- 为指定数据库删除多个影子库压测算法

```
DROP SHADOW ALGORITHM shadow_rule_t_order_sql_hint_0, shadow_rule_t_order_item_sql_
hint_0 FROM shadow_db;
```

- 为当前数据库删除单个影子库压测算法

```
DROP SHADOW ALGORITHM shadow_rule_t_order_sql_hint_0;
```

- 使用 ifExists 子句删除影子库压测算法

```
DROP SHADOW ALGORITHM IF EXISTS shadow_rule_t_order_sql_hint_0;
```

保留字

DROP、SHADOW、ALGORITHM、FROM

相关链接

- 保留字

RQL 语法

RQL (Resource & Rule Query Language) 为 Apache ShardingSphere 的资源和规则查询语言。

存储单元查询

本章节将对存储单元查询的语法进行详细说明。

SHOW STORAGE UNITS

描述

`SHOW STORAGE UNITS` 语法用于查询指定逻辑库已经注册的存储单元。

语法

```
ShowStorageUnit ::=
  'SHOW' 'STORAGE' 'UNITS' ('FROM' databaseName)? showLike?

databaseName ::=
  identifier

showLike ::=
  'LIKE' likePattern

likePattern ::=
  string
```

特别说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`；如未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
name	存储单元名称
type	存储单元类型
host	存储单元地址
port	存储单元端口
db	数据库名称
connection_timeout_milliseconds	连接超时时间（毫秒）
idle_timeout_milliseconds	连接最大空闲时间（毫秒）
max_lifetime_milliseconds	连接最大生命周期（毫秒）
max_pool_size	连接池最大连接数
min_pool_size	连接池最小连接数
read_only	只读标识
other_attributes	其他连接参数

示例

- 查询当前逻辑库中的存储单元

```
mysql> SHOW STORAGE UNITS;
+------+-------+-----------+------+------+--------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-------------------------------------------------------------------------
---------------------------------------------------------------------------------
-------------------------------------------------------------------------+
| name | type  | host      | port | db   | connection_timeout_milliseconds | idle_
timeout_milliseconds | max_lifetime_milliseconds | max_pool_size | min_pool_size |
read_only | other_attributes
|
+------+-------+-----------+------+------+--------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-------------------------------------------------------------------------
---------------------------------------------------------------------------------
-------------------------------------------------------------------------+
| ds_1 | MySQL | 127.0.0.1 | 3306 | db1  | 30000                          | 60000
| 2100000                   | 50            | 1             | false     | {
"healthCheckProperties":{},"initializationFailTimeout":1,"validationTimeout":5000,
"keepaliveTime":0,"leakDetectionThreshold":0,"registerMbeans":false,
"allowPoolSuspension":false,"autoCommit":true,"isolateInternalQueries":false} |
| ds_0 | MySQL | 127.0.0.1 | 3306 | db0  | 30000                          | 60000
| 2100000                   | 50            | 1             | false     | {
"healthCheckProperties":{},"initializationFailTimeout":1,"validationTimeout":5000,
"keepaliveTime":0,"leakDetectionThreshold":0,"registerMbeans":false,
"allowPoolSuspension":false,"autoCommit":true,"isolateInternalQueries":false} |
```

```
+------+-------+-----------+------+------+--------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-------------------------------------------------------------------------
-------------------------------------------------------------------------------
-----------------------------------------------------------------------+
2 rows in set (0.01 sec)
```

- 查询指定逻辑库中的存储单元

```
mysql> SHOW STORAGE UNITS FROM sharding_db;
+------+-------+-----------+------+------+--------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-------------------------------------------------------------------------
-------------------------------------------------------------------------------
-----------------------------------------------------------------------+
| name | type  | host      | port | db   | connection_timeout_milliseconds | idle_
timeout_milliseconds | max_lifetime_milliseconds | max_pool_size | min_pool_size |
read_only | other_attributes
|
+------+-------+-----------+------+------+--------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-------------------------------------------------------------------------
-------------------------------------------------------------------------------
-----------------------------------------------------------------------+
| ds_1 | MySQL | 127.0.0.1 | 3306 | db1  | 30000                          | 60000
| 2100000                   | 50            | 1             | false     | {
"healthCheckProperties":{},"initializationFailTimeout":1,"validationTimeout":5000,
"keepaliveTime":0,"leakDetectionThreshold":0,"registerMbeans":false,
"allowPoolSuspension":false,"autoCommit":true,"isolateInternalQueries":false} |
| ds_0 | MySQL | 127.0.0.1 | 3306 | db0  | 30000                          | 60000
| 2100000                   | 50            | 1             | false     | {
"healthCheckProperties":{},"initializationFailTimeout":1,"validationTimeout":5000,
"keepaliveTime":0,"leakDetectionThreshold":0,"registerMbeans":false,
"allowPoolSuspension":false,"autoCommit":true,"isolateInternalQueries":false} |
+------+-------+-----------+------+------+--------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-------------------------------------------------------------------------
-------------------------------------------------------------------------------
-----------------------------------------------------------------------+
2 rows in set (0.01 sec)
```

- 使用 LIKE 子句查询存储单元

```
mysql> SHOW STORAGE UNITS LIKE '%_0';
+------+-------+-----------+------+------+--------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-------------------------------------------------------------------------
-------------------------------------------------------------------------------
-----------------------------------------------------------------------+
```

```
| name | type  | host      | port | db   | connection_timeout_milliseconds | idle_
timeout_milliseconds | max_lifetime_milliseconds | max_pool_size | min_pool_size |
read_only | other_attributes
|
+------+-------+-----------+------+------+---------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-----------------------------------------------------------------------
-----------------------------------------------------------------------------------
-------------------------------------------------------------------------+
| ds_0 | MySQL | 127.0.0.1 | 3306 | db0  | 30000                           | 60000
| 2100000                    | 50            | 1             | false     | {
"healthCheckProperties":{},"initializationFailTimeout":1,"validationTimeout":5000,
"keepaliveTime":0,"leakDetectionThreshold":0,"registerMbeans":false,
"allowPoolSuspension":false,"autoCommit":true,"isolateInternalQueries":false} |
+------+-------+-----------+------+------+---------------------------------+-------
--------------------+---------------------------+---------------+---------------+--
---------+-----------------------------------------------------------------------
-----------------------------------------------------------------------------------
-------------------------------------------------------------------------+
1 rows in set (0.01 sec)
```

保留字

SHOW、STORAGE、UNITS、FROM、LIKE

相关链接

- 保留字

规则查询

本章节将对规则查询的语法进行详细说明。

分片

本章节将对分片特性的语法进行详细说明。

SHOW SHARDING TABLE RULE

描述

`SHOW SHARDING TABLE RULE` 语法用于查询指定逻辑库中的分片规则。

语法

```
ShowShardingTableRule ::=
  'SHOW' 'SHARDING' 'TABLE' ('RULE' tableName | 'RULES') ('FROM' databaseName)?

tableName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
table	逻辑表名
actual_data_nodes	实际的数据节点
actual_data_sources	实际的数据源（通过 RDL 创建的规则时显示）
database_strategy_type	数据库分片策略类型
database_sharding_column	数据库分片键
database_sharding_algorithm_type	数据库分片算法类型
d atabase_sharding_algorithm_props	数据库分片算法参数
table_strategy_type	表分片策略类型
table_sharding_column	表分片键
table_sharding_algorithm_type	表分片算法类型
table_sharding_algorithm_props	表分片算法参数
key_generate_column	分布式主键生成列
key_generator_type	分布式主键生成器类型
key_generator_props	分布式主键生成器参数

示例 - 查询指定逻辑库的分片规则

```
SHOW SHARDING TABLE RULES FROM sharding_db;
```

```
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+-------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
| table        | actual_data_nodes | actual_data_sources | database_strategy_type |
database_sharding_column | database_sharding_algorithm_type | database_sharding_
algorithm_props | table_strategy_type | table_sharding_column | table_sharding_
algorithm_type | table_sharding_algorithm_props | key_generate_column | key_
generator_type | key_generator_props |
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+-------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
| t_order      |                   | ds_0,ds_1           |                       |
|                                  |                                     | mod
| order_id              | mod                           | sharding-count=4
|                     |                   |                     |
| t_order_item |                   | ds_0,ds_1           |                       |
|                                  |                                     | mod
| order_id              | mod                           | sharding-count=4
|                     |                   |                     |
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+-------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
2 rows in set (0.12 sec)
```

- 查询当前逻辑库的分片规则

```
SHOW SHARDING TABLE RULES;
```

```
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+-------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
| table        | actual_data_nodes | actual_data_sources | database_strategy_type |
database_sharding_column | database_sharding_algorithm_type | database_sharding_
algorithm_props | table_strategy_type | table_sharding_column | table_sharding_
algorithm_type | table_sharding_algorithm_props | key_generate_column | key_
generator_type | key_generator_props |
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+-------------------
```

```
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
| t_order      |                   | ds_0,ds_1           |                       |
|                                  |                                     | mod
| order_id              | mod                           | sharding-count=4
|                     |                   |                     |
| t_order_item |                   | ds_0,ds_1           |                       |
|                                  |                                     | mod
| order_id              | mod                           | sharding-count=4
|                     |                   |                     |
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+--------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
2 rows in set (0.12 sec)
```

- 查询指定逻辑表的分片规则

```
SHOW SHARDING TABLE RULE t_order;
```

```
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+--------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
| table        | actual_data_nodes | actual_data_sources | database_strategy_type |
database_sharding_column | database_sharding_algorithm_type | database_sharding_
algorithm_props | table_strategy_type | table_sharding_column | table_sharding_
algorithm_type | table_sharding_algorithm_props | key_generate_column | key_
generator_type | key_generator_props |
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+--------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
| t_order      |                   | ds_0,ds_1           |                       |
|                                  |                                     | mod
| order_id              | mod                           | sharding-count=4
|                     |                   |                     |
+--------------+-------------------+---------------------+-----------------------
+--------------------------+----------------------------------+--------------------
---------------+---------------------+-----------------------+---------------------
----------+-------------------------------+---------------------+-----------------
---+---------------------+
1 row in set (0.12 sec)
```

保留字

CREATE、SHARDING、TABLE、RULE、FROM

相关链接

- 保留字

SHOW SHARDING ALGORITHMS

描述

SHOW SHARDING ALGORITHMS 语法用于查询指定逻辑库的分片算法。

语法

```
ShowShardingAlgorithms::=
  'SHOW' 'SHARDING' 'ALGORITHMS' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分片算法名称
type	分片算法类型
props	分片算法参数

示例

- 查询指定逻辑库的分片算法

```
SHOW SHARDING ALGORITHMS FROM sharding_db;
```

```
mysql> SHOW SHARDING ALGORITHMS FROM sharding_db;
+-------------------------+--------+---------------------------------------------
------+
| name                    | type   | props
|
+-------------------------+--------+---------------------------------------------
------+
| t_order_inline          | INLINE | algorithm-expression=t_order_${order_id % 2}
|
| t_order_item_inline     | INLINE | algorithm-expression=t_order_item_${order_id %
2}   |
+-------------------------+--------+---------------------------------------------
------+
2 rows in set (0.01 sec)
```

- 查询当前逻辑库的分片算法

```
SHOW SHARDING ALGORITHMS;
```

```
mysql> SHOW SHARDING ALGORITHMS;
+-------------------------+--------+---------------------------------------------
------+
| name                    | type   | props
|
+-------------------------+--------+---------------------------------------------
------+
| t_order_inline          | INLINE | algorithm-expression=t_order_${order_id % 2}
|
| t_order_item_inline     | INLINE | algorithm-expression=t_order_item_${order_id %
2}   |
+-------------------------+--------+---------------------------------------------
------+
2 rows in set (0.01 sec)
```

保留字

SHOW、SHARDING、ALGORITHMS、FROM

相关链接

- 保留字

SHOW UNUSED SHARDING ALGORITHMS

描述

SHOW UNUSED SHARDING ALGORITHMS 语法用于查询指定逻辑库未使用的分片算法。

语法

```
ShowShardingAlgorithms::=
  'SHOW' 'UNUSED' 'SHARDING' 'ALGORITHMS' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分片算法名称
type	分片算法类型
props	分片算法参数

示例

- 查询指定逻辑库未使用的分片算法

```
SHOW UNUSED SHARDING ALGORITHMS;
```

```
mysql> SHOW UNUSED SHARDING ALGORITHMS;
+---------------+--------+-------------------------------------------------------+
| name          | type   | props                                                 |
+---------------+--------+-------------------------------------------------------+
| t1_inline     | INLINE | algorithm-expression=t_order_${order_id % 2}          |
+---------------+--------+-------------------------------------------------------+
1 row in set (0.01 sec)
```

保留字

SHOW、UNUSED、SHARDING、ALGORITHMS、FROM

相关链接

- 保留字

SHOW DEFAULT SHARDING STRATEGY

描述

SHOW DEFAULT SHARDING STRATEGY 语法用于查询指定逻辑库的默认分片策略。

语法

```
ShowDefaultShardingStrategy::=
  'SHOW' 'DEFAULT' 'SHARDING' 'STRATEGY' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分片策略范围
type	分片策略类型
sharding_column	分片键
sharding_algorithm_name	分片算法名称
sharding_algorithm_type	分片算法类型
sharding_algorithm_props	分片算法参数

示例

- 查询指定逻辑库的默认分片策略

```
SHOW DEFAULT SHARDING STRATEGY FROM sharding_db;
```

```
mysql> SHOW DEFAULT SHARDING STRATEGY FROM sharding_db;
+----------+----------+-----------------+------------------------+----------------
---------+-----------------------------------------------------+
| name     | type     | sharding_column | sharding_algorithm_name | sharding_
algorithm_type | sharding_algorithm_props                            |
+----------+----------+-----------------+------------------------+----------------
---------+-----------------------------------------------------+
| TABLE    | STANDARD | order_id        | table_inline            | inline
| {algorithm-expression=t_order_item_${order_id % 2}} |
| DATABASE | STANDARD | order_id        | table_inline            | inline
| {algorithm-expression=t_order_item_${order_id % 2}} |
+----------+----------+-----------------+------------------------+----------------
---------+-----------------------------------------------------+
2 rows in set (0.00 sec)
```

- 查询当前逻辑库的默认分片策略

```
SHOW DEFAULT SHARDING STRATEGY;
```

```
mysql> SHOW DEFAULT SHARDING STRATEGY;
+----------+----------+-----------------+------------------------+----------------
---------+-----------------------------------------------------+
| name     | type     | sharding_column | sharding_algorithm_name | sharding_
```

```
algorithm_type | sharding_algorithm_props                          |
+----------+----------+----------------+-------------------------+----------------
---------+-------------------------------------------------------+
| TABLE    | STANDARD | order_id        | table_inline            | inline
| {algorithm-expression=t_order_item_${order_id % 2}} |
| DATABASE | STANDARD | order_id        | table_inline            | inline
| {algorithm-expression=t_order_item_${order_id % 2}} |
+----------+----------+----------------+-------------------------+----------------
---------+-------------------------------------------------------+
2 rows in set (0.00 sec)
```

保留字

SHOW、DEFAULT、SHARDING、STRATEGY、FROM

相关链接

- 保留字

SHOW SHARDING KEY GENERATORS

描述

SHOW SHARDING KEY GENERATORS 语法用于查询指定逻辑库的分布式主键生成器。

语法

```
ShowShardingKeyGenerators::=
  'SHOW' 'SHARDING' 'KEY' 'GENERATORS' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分布式主键生成器名称
type	分布式主键生成器类型
props	分布式主键生成器参数

示例

- 查询指定逻辑库的分布式主键生成器

```
SHOW SHARDING KEY GENERATORS FROM sharding_db;
```

```
mysql> SHOW SHARDING KEY GENERATORS FROM sharding_db;
+-------------------------+-----------+-------+
| name                    | type      | props |
+-------------------------+-----------+-------+
| snowflake_key_generator | snowflake |       |
+-------------------------+-----------+-------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库的分布式主键生成器

```
SHOW SHARDING KEY GENERATORS;
```

```
mysql> SHOW SHARDING KEY GENERATORS;
+-------------------------+-----------+-------+
| name                    | type      | props |
+-------------------------+-----------+-------+
| snowflake_key_generator | snowflake |       |
+-------------------------+-----------+-------+
1 row in set (0.00 sec)
```

保留字

SHOW、SHARDING、KEY、GENERATORS、FROM

相关链接

- 保留字

SHOW UNUSED SHARDING KEY GENERATORS

描述

SHOW UNUSED SHARDING KEY GENERATORS 语法用于查询指定逻辑库中未被使用的分布式主键生成器。

语法

```
ShowUnusedShardingKeyGenerators::=
  'SHOW' 'UNUSED' 'SHARDING' 'KEY' 'GENERATOR' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分布式主键生成器名称
type	分布式主键生成器类型
props	分布式主键生成器参数

示例

- 查询指定逻辑库中未被使用的分布式主键生成器

```
SHOW UNUSED SHARDING KEY GENERATORS FROM sharding_db;
```

```
mysql> SHOW UNUSED SHARDING KEY GENERATORS FROM sharding_db;
+-------------------------+-----------+-------+
| name                    | type      | props |
+-------------------------+-----------+-------+
| snowflake_key_generator | snowflake |       |
+-------------------------+-----------+-------+
1 row in set (0.01 sec)
```

- 查询当前逻辑库中未被使用的分布式主键生成器

```
SHOW UNUSED SHARDING KEY GENERATORS;
```

```
mysql> SHOW UNUSED SHARDING KEY GENERATORS;
+-------------------------+-----------+-------+
| name                    | type      | props |
+-------------------------+-----------+-------+
| snowflake_key_generator | snowflake |       |
+-------------------------+-----------+-------+
1 row in set (0.02 sec)
```

保留字

SHOW、UNUSED、SHARDING、KEY、GENERATORS、FROM

相关链接

- 保留字

SHOW SHARDING AUDITORS

描述

SHOW SHARDING AUDITORS 语法用于查询指定逻辑库中的分片审计器。

语法

```
ShowShardingAuditors::=
  'SHOW' 'SHARDING' 'AUDITORS' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分片审计器名称
type	分片审计算法类型
props	分片审计算法参数

示例

- 查询指定逻辑库中的分片审计器

```
SHOW SHARDING AUDITORS FROM sharding_db;
```

```
mysql> SHOW SHARDING AUDITORS FROM sharding_db;
+-------------------------------+-------------------------+-------+
| name                          | type                    | props |
+-------------------------------+-------------------------+-------+
| sharding_key_required_auditor | dml_sharding_conditions |       |
+-------------------------------+-------------------------+-------+
1 row in set (0.01 sec)
```

- 查询当前逻辑库中的分片审计器

```
SHOW SHARDING AUDITORS;
```

```
mysql> SHOW SHARDING AUDITORS;
+-------------------------------+-------------------------+-------+
| name                          | type                    | props |
+-------------------------------+-------------------------+-------+
| sharding_key_required_auditor | dml_sharding_conditions |       |
```

```
+-------------------------------+-------------------------+-------+
1 row in set (0.00 sec)
```

保留字

SHOW、SHARDING、AUDITORS、FROM

相关链接

- 保留字

SHOW UNUSED SHARDING AUDITORS

描述

SHOW UNUSED SHARDING AUDITORS 语法用于查询指定逻辑库中未被使用的分片审计器。

语法

```
ShowUnusedShardingAuditors::=
  'SHOW' 'SHARDING' 'AUDITOR' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分片审计器名称
type	分片审计算法类型
props	分片审计算法参数

示例

- 查询指定逻辑库中未被使用的分片审计器

```
SHOW UNUSED SHARDING AUDITORS FROM sharding_db;
```

```
mysql> SHOW UNUSED SHARDING AUDITORS FROM sharding_db;
+-------------------------------+-------------------------+-------+
| name                          | type                    | props |
+-------------------------------+-------------------------+-------+
| sharding_key_required_auditor | dml_sharding_conditions |       |
+-------------------------------+-------------------------+-------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中未被使用的的分片审计器

```
SHOW UNUSED SHARDING AUDITORS;
```

```
mysql> SHOW UNUSED SHARDING AUDITORS;
+-------------------------------+-------------------------+-------+
| name                          | type                    | props |
+-------------------------------+-------------------------+-------+
| sharding_key_required_auditor | dml_sharding_conditions |       |
+-------------------------------+-------------------------+-------+
1 row in set (0.00 sec)
```

保留字

SHOW、UNUSED、SHARDING、AUDITORS、FROM

相关链接

- 保留字

SHOW SHARDING TABLE NODES

描述

SHOW SHARDING TABLE NODES 语法用于查询指定逻辑库中的指定分片表的节点分布。

语法

```
ShowShardingTableNode::=
  'SHOW' 'SHARDING' 'TABLE' 'NODES' tableName? ('FROM' databaseName)?

tableName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分片规则名称
nodes	分片节点

示例

- 查询指定逻辑库中指定分片表的节点分布

```
SHOW SHARDING TABLE NODES t_order_item FROM sharding_db;
```

```
mysql> SHOW SHARDING TABLE NODES t_order_item FROM sharding_db;
+--------------+----------------------------------------------------------------------
----------------------------------------+
| name         | nodes
|
+--------------+----------------------------------------------------------------------
----------------------------------------+
| t_order_item | resource_0.t_order_item_0, resource_0.t_order_item_1, resource_1.
t_order_item_0, resource_1.t_order_item_1 |
+--------------+----------------------------------------------------------------------
----------------------------------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中指定分片表的节点分布

```
SHOW SHARDING TABLE NODES t_order_item;
```

```
mysql> SHOW SHARDING TABLE NODES t_order_item;
+--------------+-------------------------------------------------------------------
----------------------------------------+
| name         | nodes
|
+--------------+-------------------------------------------------------------------
----------------------------------------+
| t_order_item | resource_0.t_order_item_0, resource_0.t_order_item_1, resource_1.
t_order_item_0, resource_1.t_order_item_1 |
+--------------+-------------------------------------------------------------------
----------------------------------------+
1 row in set (0.00 sec)
```

- 查询指定逻辑库中所有分片表的节点分布

```
SHOW SHARDING TABLE NODES FROM sharding_db;
```

```
mysql> SHOW SHARDING TABLE NODES FROM sharding_db;
+--------------+-------------------------------------------------------------------
----------------------------------------+
| name         | nodes
|
+--------------+-------------------------------------------------------------------
----------------------------------------+
| t_order_item | resource_0.t_order_item_0, resource_0.t_order_item_1, resource_1.
t_order_item_0, resource_1.t_order_item_1 |
+--------------+-------------------------------------------------------------------
----------------------------------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中所有分片表的节点分布

```
SHOW SHARDING TABLE NODES;
```

```
mysql> SHOW SHARDING TABLE NODES;
+--------------+-------------------------------------------------------------------
----------------------------------------+
| name         | nodes
|
+--------------+-------------------------------------------------------------------
----------------------------------------+
| t_order_item | resource_0.t_order_item_0, resource_0.t_order_item_1, resource_1.
t_order_item_0, resource_1.t_order_item_1 |
+--------------+-------------------------------------------------------------------
----------------------------------------+
1 row in set (0.00 sec)
```

保留字

SHOW、SHARDING、TABLE、NODES、FROM

相关链接

- 保留字

SHOW SHARDING TABLE RULES USED ALGORITHM

描述

SHOW SHARDING TABLE RULES USED ALGORITHM 语法用于查询指定逻辑库中使用指定分片算法的分片规则。

语法

```
ShowShardingTableRulesUsedAlgorithm::=
  'SHOW' 'SHARDING' 'TABLE' 'RULES' 'USED' 'ALGORITHM' algorithmName ('FROM'
databaseName)?

algorithmName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
type	分片规则类型
name	分片规则名称

示例

- 查询指定逻辑库中使用指定分片算法的分片规则

```
SHOW SHARDING TABLE RULES USED ALGORITHM table_inline FROM sharding_db;
```

```
mysql> SHOW SHARDING TABLE RULES USED ALGORITHM table_inline FROM sharding_db;
+-------+--------------+
| type  | name         |
+-------+--------------+
| table | t_order_item |
+-------+--------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中使用指定分片算法的分片规则

```
SHOW SHARDING TABLE RULES USED ALGORITHM table_inline;
```

```
mysql> SHOW SHARDING TABLE RULES USED ALGORITHM table_inline;
+-------+--------------+
| type  | name         |
+-------+--------------+
| table | t_order_item |
+-------+--------------+
1 row in set (0.01 sec)
```

保留字

`SHOW`、`SHARDING`、`TABLE`、`RULES`、`USED`、`ALGORITHM`、`FROM`

相关链接

- 保留字

SHOW SHARDING TABLE RULES USED KEY GENERATOR

描述

`SHOW SHARDING TABLE RULES USED KEY GENERATOR` 语法用于查询指定逻辑库中使用指定分片主键生成器的分片规则。

语法

```
ShowShardingTableRulesUsedKeyGenerator::=
  'SHOW' 'SHARDING' 'TABLE' 'RULES' 'USED' 'KEY' 'GENERATOR' keyGeneratorName (
'FROM' databaseName)?

keyGeneratorName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
type	分片规则类型
name	分片规则名称

示例

- 查询指定逻辑库中使用指定分片主键生成器的分片规则

```
SHOW SHARDING TABLE RULES USED KEY GENERATOR snowflake_key_generator FROM sharding_
db;
```

```
mysql> SHOW SHARDING TABLE RULES USED KEY GENERATOR snowflake_key_generator FROM
sharding_db;
+-------+--------------+
| type  | name         |
+-------+--------------+
| table | t_order_item |
+-------+--------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中使用指定分片主键生成器的分片规则

```
SHOW SHARDING TABLE RULES USED KEY GENERATOR snowflake_key_generator;
```

```
mysql> SHOW SHARDING TABLE RULES USED KEY GENERATOR snowflake_key_generator;
+-------+--------------+
| type  | name         |
+-------+--------------+
| table | t_order_item |
+-------+--------------+
1 row in set (0.01 sec)
```

保留字

SHOW、SHARDING、TABLE、USED、KEY、GENERATOR、FROM

相关链接

- 保留字

SHOW SHARDING TABLE RULES USED AUDITOR

描述

SHOW SHARDING TABLE RULES USED AUDITOR 语法用于查询指定逻辑库中使用指定分片审计器的分片规则。

语法

```
ShowShardingTableRulesUsedAuditor::=
  'SHOW' 'SHARDING' 'TABLE' 'RULES' 'USED' 'AUDITOR' AuditortorName ('FROM'
databaseName)?

AuditortorName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
type	分片规则类型
name	分片规则名称

示例

- 查询指定逻辑库中使用指定分片审计器的分片规则

```
SHOW SHARDING TABLE RULES USED AUDITOR sharding_key_required_auditor FROM sharding_
db;
```

```
mysql> SHOW SHARDING TABLE RULES USED AUDITOR sharding_key_required_auditor FROM
sharding_db;
+-------+---------+
| type  | name    |
+-------+---------+
| table | t_order |
+-------+---------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中使用指定分片审计器的分片规则

```
SHOW SHARDING TABLE RULES USED AUDITOR sharding_key_required_auditor;
```

```
mysql> SHOW SHARDING TABLE RULES USED AUDITOR sharding_key_required_auditor;
+-------+---------+
| type  | name    |
+-------+---------+
| table | t_order |
+-------+---------+
1 row in set (0.00 sec)
```

保留字

SHOW、SHARDING、TABLE、RULES、USED、AUDITOR、FROM

相关链接

- 保留字

SHOW SHARDING TABLE REFERENCE RULE

描述

SHOW SHARDING BINDING TABLE RULE 语法用于查询指定逻辑库中指定分片表关联规则。

语法

```
ShowShardingBindingTableRules::=
  'SHOW' 'SHARDING' 'TABLE' 'REFERENCE' ('RULE' ruleName | 'RULES') ('FROM'
databaseName)?

ruleName ::=
    identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
name	分片表关联规则名称
sharding_table_reference	分片表关联关系

示例

- 查询指定逻辑库中的分片表关联规则

```
SHOW SHARDING TABLE REFERENCE RULES FROM sharding_db;
```

```
mysql> SHOW SHARDING TABLE REFERENCE RULES FROM sharding_db;
+-------+--------------------------+
| name  | sharding_table_reference |
+-------+--------------------------+
| ref_0 | t_a,t_b                  |
| ref_1 | t_c,t_d                  |
+-------+--------------------------+
2 rows in set (0.00 sec)
```

- 查询当前逻辑库中的分片表关联规则

```
SHOW SHARDING TABLE REFERENCE RULES;
```

```
mysql> SHOW SHARDING TABLE REFERENCE RULES;
+-------+--------------------------+
| name  | sharding_table_reference |
+-------+--------------------------+
| ref_0 | t_a,t_b                  |
| ref_1 | t_c,t_d                  |
+-------+--------------------------+
2 rows in set (0.00 sec)
```

- 查询指定逻辑库中的指定分片表关联规则

```
SHOW SHARDING TABLE REFERENCE RULE ref_0 FROM sharding_db;
```

```
mysql> SHOW SHARDING TABLE REFERENCE RULE ref_0 FROM sharding_db;
+-------+--------------------------+
| name  | sharding_table_reference |
+-------+--------------------------+
| ref_0 | t_a,t_b                  |
+-------+--------------------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的分片表关联规则

```
SHOW SHARDING TABLE REFERENCE RULE ref_0;
```

```
mysql> SHOW SHARDING TABLE REFERENCE RULE ref_0;
+-------+--------------------------+
| name  | sharding_table_reference |
+-------+--------------------------+
```

```
| ref_0 | t_a,t_b                  |
+-------+--------------------------+
1 row in set (0.00 sec)
```

保留字

SHOW、SHARDING、TABLE、REFERENCE、RULE、RULES、FROM

相关链接

- 保留字

COUNT SHARDING RULE

描述

COUNT SHARDING RULE 语法用于查询指定逻辑库中的分片规则数量。

语法

```
CountShardingRule::=
  'COUNT' 'SHARDING' 'RULE' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
rule_name	规则类型
database	规则所属逻辑库
count	规则数量

示例

- 查询指定逻辑库中的分片规则数量

```
COUNT SHARDING RULE FROM sharding_db;
```

```
mysql> COUNT SHARDING RULE FROM sharding_db;
+-------------------------+----------------+-------+
| rule_name               | database       | count |
+-------------------------+----------------+-------+
| sharding_table          | sharding_db    | 2     |
| sharding_table_reference | sharding_db    | 2     |
+-------------------------+----------------+-------+
2 rows in set (0.00 sec)
```

- 查询当前逻辑库中的分片规则数量

```
COUNT SHARDING RULE;
```

```
mysql> COUNT SHARDING RULE;
+-------------------------+----------------+-------+
| rule_name               | database       | count |
+-------------------------+----------------+-------+
| sharding_table          | sharding_db    | 2     |
| sharding_table_reference | sharding_db    | 2     |
+-------------------------+----------------+-------+
2 rows in set (0.00 sec)
```

保留字

COUNT、SHARDING、RULE、FROM

相关链接

- 保留字

广播表

本章节将对广播表特性的语法进行详细说明。

SHOW BROADCAST TABLE RULE

描述

`SHOW BROADCAST TABLE RULES` 语法用于查询指定数据库中具有广播规则的表。

语法定义

```
ShowBroadcastTableRule ::=
  'SHOW' 'BROADCAST' 'TABLE' 'RULES' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`；如未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
broadcast_table	广播表名称

示例

- 查询指定数据库中具有广播规则的表

```
SHOW BROADCAST TABLE RULES FROM sharding_db;
```

```
mysql> SHOW BROADCAST TABLE RULES FROM sharding_db;
+-----------------+
| broadcast_table |
+-----------------+
| t_a             |
| t_b             |
| t_c             |
+-----------------+
3 rows in set (0.00 sec)
```

- 查询当前逻辑库中具有广播规则的表

```
SHOW BROADCAST TABLE RULES;
```

```
mysql> SHOW BROADCAST TABLE RULES;
+-----------------+
| broadcast_table |
+-----------------+
| t_a             |
| t_b             |
| t_c             |
+-----------------+
3 rows in set (0.00 sec)
```

保留字

SHOW、BROADCAST、TABLE、RULES

相关链接

- 保留字

COUNT BROADCAST RULE

描述

COUNT BROADCAST RULE 语法用于查询指定逻辑库中的广播表规则数量。

语法

```
CountBroadcastRule::=
  'COUNT' 'BROADCAST' 'RULE' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
rule_name	规则类型
database	规则所属逻辑库
count	规则数量

示例

- 查询指定逻辑库中的广播表规则数量

```
COUNT BROADCAST RULE FROM sharding_db;
```

```
mysql> COUNT BROADCAST RULE FROM sharding_db;
+--------------------------+----------------+-------+
| rule_name                | database       | count |
+--------------------------+----------------+-------+
| broadcast_table          | sharding_db    | 0     |
+--------------------------+----------------+-------+
1 rows in set (0.00 sec)
```

- 查询当前逻辑库中的广播表规则数量

```
COUNT BROADCAST RULE;
```

```
mysql> COUNT BROADCAST RULE;
+--------------------------+----------------+-------+
| rule_name                | database       | count |
+--------------------------+----------------+-------+
| broadcast_table          | sharding_db    | 0     |
+--------------------------+----------------+-------+
1 rows in set (0.00 sec)
```

保留字

COUNT、BROADCAST、RULE、FROM

相关链接

- 保留字

单表

本章节将对单表特性的语法进行详细说明。

SHOW SINGLE TABLE

描述

SHOW SINGLE TABLE 语法用于查询指定逻辑库中的单表。

语法

```
ShowSingleTable::=
  'SHOW' 'SINGLE' ('TABLES' ('LIKE' likeLiteral)?|'TABLE' tableName) ('FROM'
databaseName)?

tableName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
table_name	单表名称
storage_unit_name	单表所在的数据源名称

示例

- 查询指定逻辑库中的指定单表

```
SHOW SINGLE TABLE t_user FROM sharding_db;
```

```
mysql> SHOW SINGLE TABLE t_user FROM sharding_db;
+------------+---------------+
| table_name | resource_name |
+------------+---------------+
| t_user     | ds_0          |
+------------+---------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的指定单表

```
SHOW SINGLE TABLE t_user;
```

```
mysql> SHOW SINGLE TABLE t_user;
+------------+-------------------+
| table_name | storage_unit_name |
+------------+-------------------+
| t_user     | ds_0              |
+------------+-------------------+
1 row in set (0.00 sec)
```

- 查询指定逻辑库中的单表

```
SHOW SINGLE TABLES FROM sharding_db;
```

```
mysql> SHOW SINGLE TABLES FROM sharding_db;
+------------+-------------------+
| table_name | storage_unit_name |
+------------+-------------------+
| t_user     | ds_0              |
+------------+-------------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的单表

```
SHOW SINGLE TABLES;
```

```
mysql> SHOW SINGLE TABLES;
+------------+-------------------+
| table_name | storage_unit_name |
+------------+-------------------+
| t_user     | ds_0              |
+------------+-------------------+
1 row in set (0.00 sec)
```

- 查询指定逻辑库中表名以 `order_5` 结尾的单表

```
SHOW SINGLE TABLES LIKE '%order_5' FROM sharding_db;
```

```
mysql> SHOW SINGLE TABLES LIKE '%order_5' FROM sharding_db;
+------------+-------------------+
| table_name | storage_unit_name |
+------------+-------------------+
| t_order_5  | ds_1              |
+------------+-------------------+
1 row in set (0.11 sec)
```

- 查询当前逻辑库中表名以 `order_5` 结尾的单表

```
SHOW SINGLE TABLES LIKE '%order_5';
```

```
mysql> SHOW SINGLE TABLES LIKE '%order_5';
+------------+-------------------+
| table_name | storage_unit_name |
+------------+-------------------+
| t_order_5  | ds_1              |
+------------+-------------------+
1 row in set (0.11 sec)
```

保留字

SHOW、SINGLE、TABLE、TABLES、LIKE、FROM

相关链接

- 保留字

SHOW DEFAULT SINGLE TABLE STORAGE UNIT

描述

SHOW DEFAULT SINGLE TABLE STORAGE UNIT 语法用于查询指定逻辑库中的存储单元信息。

语法

```
ShowDefaultSingleTableStorageUnit::=
  'SHOW' 'DEFAULT' 'SINGLE' 'TABLE' 'STORAGE' 'UNIT' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
storage_unit_name	存储单元名称

示例

- 查询当前逻辑库中的存储单元信息

```
SHOW DEFAULT SINGLE TABLE STORAGE UNIT
```

```
sql> SHOW DEFAULT SINGLE TABLE STORAGE UNIT;
+-------------------+
| storage_unit_name |
+-------------------+
|  ds_0             |
+-------------------+
1 row in set (0.01 sec)
```

保留字

SHOW、DEFAULT、SINGLE、TABLE、STORAGE、UNIT

相关链接

- 保留字

COUNT SINGLE_TABLE RULE

描述

COUNT SINGLE TABLE 语法用于查询指定逻辑库中的单表个数。

语法

```
CountSingleTable::=
  'COUNT' 'SINGLE' 'TABLE' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
database	单表所在的数据库名称
count	单表个数

示例

- 查询当前逻辑库中的单表规则个数

```
COUNT SINGLE TABLE
```

```
mysql> COUNT SINGLE TABLE;
+----------+--------+
| database | count  |
+----------+--------+
| ds       | 2      |
+----------+--------+
1 row in set (0.02 sec)
```

保留字

COUNT、SINGLE、TABLE、FROM

相关链接

- 保留字

SHOW UNLOADED SINGLE TABLES

描述

SHOW UNLOADED SINGLE TABLES 语法用于查询未加载的单表。

语法

```
showUnloadedSingleTables::=
  'SHOW' 'UNLOADED' 'SINGLE' 'TABLES'
```

返回值说明

列	说明
table_name	单表名称
storage_unit_name	单表所在的存储单元名称

示例

- 查询未加载的单表

```
SHOW UNLOADED SINGLE TABLES;
```

```
mysql> SHOW UNLOADED SINGLE TABLES;
+------------+-------------------+
| table_name | storage_unit_name |
+------------+-------------------+
| t_single   | ds_1              |
+------------+-------------------+
1 row in set (0.01 sec)
```

保留字

SHOW、UNLOADED、SINGLE、TABLES

相关链接

- 保留字

读写分离

本章节将对读写分离特性的语法进行详细说明。

SHOW READWRITE_SPLITTING RULE

描述

`SHOW READWRITE_SPLITTING RULE` 语法用于查询指定逻辑库中的指定读写分离规则。

语法

```
ShowReadWriteSplittingRule::=
  'SHOW' 'READWRITE_SPLITTING' ('RULE' ruleName | 'RULES') ('FROM' databaseName)?

ruleName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
name	读写分离规则名称
write_data_source_name	写数据源名称
read_data_source_names	读数据源名称列表
transactional_read_query_strategy	事务内读请求路由策略
load_balancer_type	负载均衡算法类型
load_balancer_props	负载均衡算法参数

示例

- 查询指定逻辑库中的读写分离规则

```
SHOW READWRITE_SPLITTING RULES FROM readwrite_splitting_db;
```

```
mysql> SHOW READWRITE_SPLITTING RULES FROM readwrite_splitting_db;
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| name       | write_storage_unit_name | read_storage_unit_names | transactional_
read_query_strategy | load_balancer_type | load_balancer_props |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| ms_group_0 | write_ds                | read_ds_0,read_ds_1     | DYNAMIC
| random             |                    |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
1 row in set (0.01 sec)
```

- 查询当前逻辑库中的读写分离规则

```
SHOW READWRITE_SPLITTING RULES;
```

```
mysql> SHOW READWRITE_SPLITTING RULES;
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| name       | write_storage_unit_name | read_storage_unit_names | transactional_
read_query_strategy | load_balancer_type | load_balancer_props |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| ms_group_0 | write_ds                | read_ds_0,read_ds_1     | DYNAMIC
| random             |                    |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
1 row in set (0.01 sec)
```

- 查询指定逻辑库中的指定读写分离规则

```
SHOW READWRITE_SPLITTING RULE ms_group_0 FROM readwrite_splitting_db;
```

```
mysql> SHOW READWRITE_SPLITTING RULE ms_group_0 FROM readwrite_splitting_db;
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| name       | write_storage_unit_name | read_storage_unit_names | transactional_
read_query_strategy | load_balancer_type | load_balancer_props |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| ms_group_0 | write_ds                | read_ds_0,read_ds_1     | DYNAMIC
```

```
| random             |                     |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
1 row in set (0.01 sec)
```

- 查询当前逻辑库中的指定读写分离规则

```
SHOW READWRITE_SPLITTING RULE ms_group_0;
```

```
mysql> SHOW READWRITE_SPLITTING RULE ms_group_0;
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| name       | write_storage_unit_name | read_storage_unit_names | transactional_
read_query_strategy | load_balancer_type | load_balancer_props |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
| ms_group_0 | write_ds                | read_ds_0,read_ds_1     | DYNAMIC
| random             |                     |
+------------+-------------------------+-------------------------+-----------------
------------------+--------------------+---------------------+
1 row in set (0.01 sec)
```

保留字

SHOW、READWRITE_SPLITTING、RULE、RULES、FROM

相关链接

- 保留字

COUNT READWRITE_SPLITTING RULE

描述

`COUNT READWRITE_SPLITTING RULE` 语法用于查询指定逻辑库中的读写分离规则数量。

语法

```
CountReadwriteSplittingRule::=
  'COUNT' 'READWRITE_SPLITTING' 'RULE' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
rule_name	规则类型
database	规则所属逻辑库
count	规则数量

示例

- 查询指定逻辑库中的读写分离规则数量

```
COUNT READWRITE_SPLITTING RULE FROM readwrite_splitting_db;
```

```
mysql> COUNT READWRITE_SPLITTING RULE FROM readwrite_splitting_db;
+---------------------+---------------------------+-------+
| rule_name           | database                  | count |
+---------------------+---------------------------+-------+
| readwrite_splitting | readwrite_splitting_db    | 1     |
+---------------------+---------------------------+-------+
1 row in set (0.02 sec)
```

- 查询当前逻辑库中的读写分离规则数量

```
COUNT READWRITE_SPLITTING RULE;
```

```
mysql> COUNT READWRITE_SPLITTING RULE;
+---------------------+---------------------------+-------+
| rule_name           | database                  | count |
+---------------------+---------------------------+-------+
| readwrite_splitting | readwrite_splitting_db    | 1     |
+---------------------+---------------------------+-------+
1 row in set (0.00 sec)
```

保留字

COUNT、READWRITE_SPLITTING、RULE、FROM

相关链接

- 保留字

数据加密

本章节将对数据加密特性的语法进行详细说明。

SHOW ENCRYPT RULES

描述

SHOW ENCRYPT RULES 语法用于查询指定逻辑库中的数据加密规则。

语法

```
ShowEncryptRule::=
  'SHOW' 'ENCRYPT' ('RULES' | 'TABLE' 'RULE' ruleName) ('FROM' databaseName)?

ruleName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
table	逻辑表名
logic_column	逻辑列名
cipher_column	密文列名
assisted_query_column	辅助查询列名
like_query_column	模糊查询列名
encryptor_type	加密算法类型
encryptor_props	加密算法参数
assisted_query_type	辅助查询算法类型
assisted_query_props	辅助查询算法参数
like_query_type	模糊查询算法类型
like_query_props	模糊查询算法参数

示例

- 查询指定逻辑库中的数据加密规则

```
SHOW ENCRYPT RULES FROM encrypt_db;
```

```
mysql> SHOW ENCRYPT RULES FROM encrypt_db;
+-----------+--------------+---------------+-----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| table     | logic_column | cipher_column | assisted_query_column | like_query_
column | encryptor_type | encryptor_props         | assisted_query_type | assisted_
query_props | like_query_type | like_query_props |
+-----------+--------------+---------------+-----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| t_user    | pwd          | pwd_cipher    |                       |
| AES            | aes-key-value=123456abc, digest-algorithm-name=SHA-1 |
|                      |                 |                  |
| t_encrypt | pwd          | pwd_cipher    |                       |
| AES            | aes-key-value=123456abc, digest-algorithm-name=SHA-1 |
|                      |                 |                  |
+-----------+--------------+---------------+--------------+-----------------------
+------------------+----------------+-------------------------+------------------
```

```
--+----------------------+----------------+------------------+
2 rows in set (0.00 sec)
```

- 查询当前逻辑库中的数据加密规则

```
SHOW ENCRYPT RULES;
```

```
mysql> SHOW ENCRYPT RULES;
+-----------+--------------+---------------+----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| table     | logic_column | cipher_column | assisted_query_column | like_query_
column | encryptor_type | encryptor_props         | assisted_query_type | assisted_
query_props | like_query_type | like_query_props |
+-----------+--------------+---------------+----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| t_user    | pwd          | pwd_cipher    |                      |
| AES            | aes-key-value=123456abc, digest-algorithm-name=SHA-1 |
|                      |                 |                  |
| t_encrypt | pwd          | pwd_cipher    |                      |
| AES            | aes-key-value=123456abc, digest-algorithm-name=SHA-1 |
|                      |                 |                  |
+-----------+--------------+---------------+--------------+----------------------
+------------------+----------------+-------------------------+------------------
--+----------------------+-----------------+------------------+
2 rows in set (0.00 sec)
```

- 查询指定逻辑库中指定的数据加密规则

```
SHOW ENCRYPT TABLE RULE t_encrypt FROM encrypt_db;
```

```
mysql> SHOW ENCRYPT TABLE RULE t_encrypt FROM encrypt_db;
+-----------+--------------+---------------+----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| table     | logic_column | cipher_column | assisted_query_column | like_query_
column | encryptor_type | encryptor_props         | assisted_query_type | assisted_
query_props | like_query_type | like_query_props |
+-----------+--------------+---------------+----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| t_encrypt | pwd          | pwd_cipher    |                      |
| AES            | aes-key-value=123456abc, digest-algorithm-name=SHA-1 |
|                      |                 |                  |
+-----------+--------------+---------------+----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
```

```
1 row in set (0.01 sec)
```

- 查询当前逻辑库中指定的数据加密规则

```
SHOW ENCRYPT TABLE RULE t_encrypt;
```

```
mysql> SHOW ENCRYPT TABLE RULE t_encrypt;
+-----------+--------------+---------------+-----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| table     | logic_column | cipher_column | assisted_query_column | like_query_
column | encryptor_type | encryptor_props         | assisted_query_type | assisted_
query_props | like_query_type | like_query_props |
+-----------+--------------+---------------+-----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
| t_encrypt | pwd          | pwd_cipher    |                       |
| AES            | aes-key-value=123456abc, digest-algorithm-name=SHA-1 |
|                      |                 |                  |
+-----------+--------------+---------------+-----------------------+---------------
----+----------------+-------------------------+---------------------+-------------
---------+-----------------+------------------+
1 row in set (0.01 sec)
```

保留字

SHOW、ENCRYPT、TABLE、RULE、RULES、FROM

相关链接

- 保留字

COUNT ENCRYPT RULE

描述

COUNT ENCRYPT RULE 语法用于查询指定逻辑库中的加密规则数量。

语法

```
CountEncryptRule::=
  'COUNT' 'ENCRYPT' 'RULE' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
rule_name	规则类型
database	规则所属逻辑库
count	规则数量

示例

- 查询指定逻辑库中的加密规则数量

```
COUNT ENCRYPT RULE FROM encrypt_db;
```

```
mysql> COUNT ENCRYPT RULE FROM encrypt_db;
+-----------+---------------+-------+
| rule_name | database      | count |
+-----------+---------------+-------+
| encrypt   | encrypt_db    | 2     |
+-----------+---------------+-------+
1 row in set (0.01 sec)
```

- 查询当前逻辑库中的加密规则数量

```
COUNT ENCRYPT RULE;
```

```
mysql> COUNT ENCRYPT RULE;
+-----------+---------------+-------+
| rule_name | database      | count |
+-----------+---------------+-------+
| encrypt   | encrypt_db    | 2     |
```

```
+-----------+---------------+-------+
1 row in set (0.01 sec)
```

保留字

COUNT、ENCRYPT、RULE、FROM

相关链接

- 保留字

数据脱敏

本章节将对数据脱敏的语法进行详细说明。

SHOW MASK RULES

描述

SHOW MASK RULES 语法用于查询指定逻辑库中的数据脱敏规则。

语法

```
ShowMaskRule::=
  'SHOW' 'MASK' ('RULES' | 'RULE' ruleName) ('FROM' databaseName)?

ruleName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE，如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
table	表名
column	列名
algorithm_type	数据脱敏算法类型
algorithm_props	数据脱敏算法参数

示例

- 查询指定逻辑库中的所有数据脱敏规则

```
SHOW MASK RULES FROM mask_db;
```

```
mysql> SHOW MASK RULES FROM mask_db;
+---------+----------+------------------+--------------------------------+
| table   | column   | algorithm_type   | algorithm_props                |
+---------+----------+------------------+--------------------------------+
| t_mask  | phoneNum | MASK_FROM_X_TO_Y | to-y=2,replace-char=*,from-x=1 |
| t_mask  | address  | MD5              |                                |
| t_order | order_id | MD5              |                                |
| t_user  | user_id  | MASK_FROM_X_TO_Y | to-y=2,replace-char=*,from-x=1 |
+---------+----------+------------------+--------------------------------+
4 rows in set (0.01 sec)
```

- 查询当前逻辑库中的所有数据脱敏规则

```
SHOW MASK RULES;
```

```
mysql> SHOW MASK RULES;
+---------+----------+------------------+--------------------------------+
| table   | column   | algorithm_type   | algorithm_props                |
+---------+----------+------------------+--------------------------------+
| t_mask  | phoneNum | MASK_FROM_X_TO_Y | to-y=2,replace-char=*,from-x=1 |
| t_mask  | address  | MD5              |                                |
| t_order | order_id | MD5              |                                |
| t_user  | user_id  | MASK_FROM_X_TO_Y | to-y=2,replace-char=*,from-x=1 |
+---------+----------+------------------+--------------------------------+
4 rows in set (0.01 sec)
```

- 查询指定逻辑库中的指定数据脱敏算法

```
SHOW MASK RULE t_mask FROM mask_db;
```

```
mysql> SHOW MASK RULE t_mask FROM mask_db;
+--------+--------------+------------------+-------------------------------+
| table  | logic_column | mask_algorithm   | props                         |
+--------+--------------+------------------+-------------------------------+
| t_mask | phoneNum     | MASK_FROM_X_TO_Y | to-y=2,replace-char=*,from-x=1 |
| t_mask | address      | MD5              |                               |
+--------+--------------+------------------+-------------------------------+
2 rows in set (0.00 sec)
```

- 查询当前逻辑库中的指定数据脱敏算法

```
SHOW MASK RULE t_mask;
```

```
mysql> SHOW MASK RULE t_mask;
+--------+--------------+------------------+-------------------------------+
| table  | logic_column | mask_algorithm   | props                         |
+--------+--------------+------------------+-------------------------------+
| t_mask | phoneNum     | MASK_FROM_X_TO_Y | to-y=2,replace-char=*,from-x=1 |
| t_mask | address      | MD5              |                               |
+--------+--------------+------------------+-------------------------------+
2 rows in set (0.00 sec)
```

保留字

SHOW、MASK、RULE、RULES、FROM

相关链接

- 保留字

COUNT MASK RULE

描述

COUNT MASK RULE 语法用于查询指定逻辑库中的数据脱敏规则数量。

语法

```
CountMaskRule::=
  'COUNT' 'MASK' 'RULE' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE，如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
rule_name	规则类型
database	规则所属逻辑库
count	规则数量

示例

- 查询指定逻辑库中的数据脱敏规则数量

```
COUNT MASK RULE FROM mask_db;
```

```
mysql> COUNT MASK RULE FROM mask_db;
+-----------+----------+-------+
| rule_name | database | count |
+-----------+----------+-------+
| mask      | mask_db  | 3     |
+-----------+----------+-------+
1 row in set (0.50 sec)
```

- 查询当前逻辑库中的数据脱敏规则数量

```
COUNT MASK RULE;
```

```
mysql> COUNT MASK RULE;
+-----------+----------+-------+
| rule_name | database | count |
+-----------+----------+-------+
| mask      | mask_db  | 3     |
+-----------+----------+-------+
1 row in set (0.50 sec)
```

保留字

COUNT、MASK、RULE、FROM

相关链接

- 保留字

影子库压测

本章节将对影子库压测特性的语法进行详细说明。

SHOW SHADOW RULE

描述

SHOW SHADOW RULE 语法用于查询指定逻辑库中的影子规则。

语法

```
ShowEncryptRule::=
  'SHOW' 'SHADOW' ('RULES' | 'RULE' shadowRuleName) ('FROM' databaseName)?

shadowRuleName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
rule_name	规则名称
source_name	数据源名称
shadow_name	影子数据源名称
shadow_table	影子表

示例

- 查询指定逻辑库中的指定影子规则

```
SHOW SHADOW RULE shadow_rule FROM shadow_db;
```

```
mysql> SHOW SHADOW RULE shadow_rule FROM shadow_db;
+-------------+-------------+-------------+----------------------+
| rule_name   | source_name | shadow_name | shadow_table         |
+-------------+-------------+-------------+----------------------+
| shadow_rule | ds_0        | ds_1        | t_order_item,t_order |
+-------------+-------------+-------------+----------------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的指定影子规则

```
SHOW SHADOW RULE shadow_rule;
```

```
mysql> SHOW SHADOW RULE shadow_rule;
+-------------+-------------+-------------+----------------------+
| rule_name   | source_name | shadow_name | shadow_table         |
+-------------+-------------+-------------+----------------------+
| shadow_rule | ds_0        | ds_1        | t_order_item,t_order |
+-------------+-------------+-------------+----------------------+
1 row in set (0.01 sec)
```

- 查询指定逻辑库中的影子规则

```
SHOW SHADOW RULES FROM shadow_db;
```

```
mysql> SHOW SHADOW RULES FROM shadow_db;
+-------------+-------------+-------------+----------------------+
| rule_name   | source_name | shadow_name | shadow_table         |
+-------------+-------------+-------------+----------------------+
| shadow_rule | ds_0        | ds_1        | t_order_item,t_order |
+-------------+-------------+-------------+----------------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的影子规则

```
SHOW SHADOW RULES;
```

```
mysql> SHOW SHADOW RULES;
+-------------+-------------+-------------+----------------------+
| rule_name   | source_name | shadow_name | shadow_table         |
+-------------+-------------+-------------+----------------------+
| shadow_rule | ds_0        | ds_1        | t_order_item,t_order |
+-------------+-------------+-------------+----------------------+
```

```
1 row in set (0.00 sec)
```

保留字

SHOW、SHADOW、RULE、RULES、FROM

相关链接

- 保留字

SHOW SHADOW TABLE RULES

描述

SHOW SHADOW TABLE RULES 语法用于查询指定逻辑库中的影子表规则。

语法

```
ShowEncryptRule::=
  'SHOW' 'SHADOW' 'TABLE' 'RULES' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
shadow_table	影子表
shadow_algorithm_name	影子算法名称

示例

- 查询指定逻辑库中的影子表规则

```
SHOW SHADOW TABLE RULES FROM shadow_db;
```

```
mysql> SHOW SHADOW TABLE RULES FROM shadow_db;
+--------------+-------------------------------------------------------+
| shadow_table | shadow_algorithm_name                                 |
+--------------+-------------------------------------------------------+
| t_order_item | shadow_rule_t_order_item_value_match                  |
| t_order      | sql_hint_algorithm,shadow_rule_t_order_regex_match |
+--------------+-------------------------------------------------------+
2 rows in set (0.00 sec)
```

- 查询当前逻辑库中的影子表规则

```
SHOW SHADOW TABLE RULES;
```

```
mysql> SHOW SHADOW TABLE RULES;
+--------------+-------------------------------------------------------+
| shadow_table | shadow_algorithm_name                                 |
+--------------+-------------------------------------------------------+
| t_order_item | shadow_rule_t_order_item_value_match                  |
| t_order      | sql_hint_algorithm,shadow_rule_t_order_regex_match |
+--------------+-------------------------------------------------------+
2 rows in set (0.01 sec)
```

保留字

`SHOW`、`SHADOW`、`TABLE`、`RULES`、`FROM`

相关链接

- 保留字

SHOW SHADOW ALGORITHMS

描述

`SHOW SHADOW ALGORITHMS` 语法用于查询指定逻辑库中的影子算法。

语法

```
ShowEncryptAlgorithm::=
  'SHOW' 'SHADOW' 'ALGORITHMS' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
shadow_algorithm_name	影子算法名称
type	算法类型
props	算法参数
is_default	是否默认

示例

- 查询指定逻辑库中的影子算法

```
SHOW SHADOW ALGORITHMS FROM shadow_db;
```

```
mysql> SHOW SHADOW ALGORITHMS FROM shadow_db;
+-------------------------+-------------+----------------------------------------
+------------+
| shadow_algorithm_name   | type        | props                                  |
is_default |
+-------------------------+-------------+----------------------------------------
+------------+
| user_id_match_algorithm | VALUE_MATCH | column=user_id,operation=insert,value=1 |
false      |
+-------------------------+-------------+----------------------------------------
+------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的影子算法

```
SHOW SHADOW ALGORITHMS;
```

```
mysql> SHOW SHADOW ALGORITHMS;
+-------------------------+-------------+----------------------------------------
+------------+
| shadow_algorithm_name   | type        | props                                  |
is_default |
+-------------------------+-------------+----------------------------------------
+------------+
| user_id_match_algorithm | VALUE_MATCH | column=user_id,operation=insert,value=1 |
false      |
+-------------------------+-------------+----------------------------------------
+------------+
1 row in set (0.00 sec)
```

保留字

SHOW、SHADOW、ALGORITHMS、FROM

相关链接

- 保留字

SHOW DEFAULT SHADOW ALGORITHM

描述

SHOW DEFAULT SHADOW ALGORITHM 语法用于查询指定逻辑库中的默认影子算法。

语法

```
ShowDefaultShadowAlgorithm::=
  'SHOW' 'DEFAULT' 'SHADOW' 'ALGORITHM' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
shadow_algorithm_name	影子算法名称
type	算法类型
props	算法参数

示例

- 查询指定逻辑库中的默认影子算法

```
SHOW DEFAULT SHADOW ALGORITHM FROM shadow_db;
```

```
mysql> SHOW DEFAULT SHADOW ALGORITHM FROM shadow_db;
+-------------------------+-------------+----------------------------------------+
| shadow_algorithm_name   | type        | props                                  |
+-------------------------+-------------+----------------------------------------+
| user_id_match_algorithm | VALUE_MATCH | column=user_id,operation=insert,value=1 |
+-------------------------+-------------+----------------------------------------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的默认影子算法

```
SHOW DEFAULT SHADOW ALGORITHM;
```

```
mysql> SHOW DEFAULT SHADOW ALGORITHM;
+-------------------------+-------------+----------------------------------------+
| shadow_algorithm_name   | type        | props                                  |
+-------------------------+-------------+----------------------------------------+
| user_id_match_algorithm | VALUE_MATCH | column=user_id,operation=insert,value=1 |
+-------------------------+-------------+----------------------------------------+
1 row in set (0.00 sec)
```

保留字

SHOW、DEFAULT、SHADOW、ALGORITHM、FROM

相关链接

- 保留字

COUNT SHADOW RULE

描述

COUNT SHADOW RULE 语法用于查询指定逻辑库中的影子库压测规则数量。

语法

```
CountShadowRule::=
  'COUNT' 'SHADOW' 'RULE' ('FROM' databaseName)?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

返回值说明

列	说明
rule_name	规则类型
database	规则所属逻辑库
count	规则数量

示例

- 查询指定逻辑库中的影子库压测规则数量

```
COUNT SHADOW RULE FROM shadow_db;
```

```
mysql> COUNT SHADOW RULE FROM shadow_db;
+-----------+--------------+-------+
| rule_name | database     | count |
+-----------+--------------+-------+
| shadow    | shadow_db    | 1     |
+-----------+--------------+-------+
1 row in set (0.00 sec)
```

- 查询当前逻辑库中的影子库压测规则数量

```
COUNT SHADOW RULE;
```

```
mysql> COUNT SHADOW RULE;
+-----------+--------------+-------+
| rule_name | database     | count |
+-----------+--------------+-------+
| shadow    | shadow_db    | 1     |
+-----------+--------------+-------+
1 row in set (0.01 sec)
```

保留字

COUNT、SHADOW、RULE、FROM

相关链接

- 保留字

RAL 语法

RAL (Resource & Rule Administration Language) 为 Apache ShardingSphere 的管理语言，负责事务类型切换、弹性伸缩等增量功能的操作。

全局规则

本章节将对全局规则的语法进行详细说明。

SHOW AUTHORITY RULE

描述

`SHOW AUTHORITY RULE` 语法用于查询权限规则配置。### 语法

```
ShowAuthorityRule ::=
  'SHOW' 'AUTHORITY' 'RULE'
```

返回值说明

列	说明
users	用户
provider	权限提供者类型
props	权限参数

示例

- 查询权限规则配置

```
SHOW AUTHORITY RULE;
```

```
mysql> SHOW AUTHORITY RULE;
+--------------------+---------------+-------+
| users              | provider      | props |
+--------------------+---------------+-------+
| root@%; sharding@% | ALL_PERMITTED |       |
+--------------------+---------------+-------+
1 row in set (0.07 sec)
```

保留字

SHOW、AUTHORITY、RULE

相关链接

- 保留字

SHOW TRANSACTION RULE

描述

SHOW TRANSACTION RULE 语法用于查询事务规则配置。### 语法

```
ShowTransactionRule ::=
  'SHOW' 'TRANSACTION' 'RULE'
```

返回值说明

列	说明
default_type	默认事务类型
provider_type	事务提供者类型
props	事务参数

示例

- 查询事务规则配置

```
SHOW TRANSACTION RULE;
```

```
mysql> SHOW TRANSACTION RULE;
+--------------+---------------+-------+
| default_type | provider_type | props |
+--------------+---------------+-------+
| LOCAL        |               |       |
+--------------+---------------+-------+
1 row in set (0.05 sec)
```

保留字

SHOW、TRANSACTION、RULE

相关链接

- 保留字

ALTER TRANSACTION RULE

描述

ALTER TRANSACTION RULE 语法用于修改事务规则。

语法

```
AlterTransactionRule ::=
  'ALTER' 'TRANSACTION' 'RULE' '(' 'DEFAULT' '=' defaultTransactionType ',' 'TYPE'
'(' 'NAME' '=' transactionManager ',' propertiesDefinition ')' ')'

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

defaultTransactionType ::=
  string

transactionManager ::=
  string

key ::=
  string

value ::=
  literal
```

补充说明

- defaultTransactionType 支持 LOCAL、XA、BASE
- transactionManager 支持 Atomikos 和 Narayana

示例

- 修改事务规则配置

```
ALTER TRANSACTION RULE(
  DEFAULT="XA", TYPE(NAME="Narayana")
);
```

保留字

ALTER、TRANSACTION、RULE、DEFAULT、TYPE、NAME、PROPERTIES

相关链接

- 保留字

SHOW SQL_PARSER RULE

描述

SHOW SQL_PARSER RULE 语法用于查询解析引擎规则配置。

语法

```
ShowSqlParserRule ::=
  'SHOW' 'SQL_PARSER' 'RULE'
```

返回值说明

列	说明
parse_tree_cache	语法树缓存
sql_statement_cache	SQL 语句缓存

示例

- 查询解析引擎规则配置

```
SHOW SQL_PARSER RULE;
```

```
mysql> SHOW SQL_PARSER RULE;
+----------------------------------------+----------------------------------------
---+
| parse_tree_cache                       | sql_statement_cache
|
+----------------------------------------+----------------------------------------
---+
| initialCapacity: 128, maximumSize: 1024 | initialCapacity: 2000, maximumSize:
65535 |
+----------------------------------------+----------------------------------------
---+
1 row in set (0.05 sec)
```

保留字

SHOW、SQL_PARSER、RULE

相关链接

- 保留字

ALTER SQL_PARSER RULE

描述

ALTER SQL_PARSER RULE 语法用于修改解析引擎规则配置。

语法

```
AlterSqlParserRule ::=
  'ALTER' 'SQL_PARSER' 'RULE' '(' sqlParserRuleDefinition ')'

sqlParserRuleDefinition ::=
  parseTreeCacheDefinition? (',' sqlStatementCacheDefinition)?

parseTreeCacheDefinition ::=
  'PARSE_TREE_CACHE' '(' cacheOption ')'
```

```
sqlStatementCacheDefinition ::=
  'SQL_STATEMENT_CACHE' '(' cacheOption ')'

cacheOption ::=
  ('INITIAL_CAPACITY' '=' initialCapacity)? (','? 'MAXIMUM_SIZE' '=' maximumSize)?

initialCapacity ::=
  int

maximumSize ::=
  int
```

补充说明

- PARSE_TREE_CACHE：语法树本地缓存配置
- SQL_STATEMENT_CACHE：SQL 语句本地缓存配置项

示例

- 修改 SQL 解析引擎规则

```
ALTER SQL_PARSER RULE (
  PARSE_TREE_CACHE(INITIAL_CAPACITY=128, MAXIMUM_SIZE=1024),
  SQL_STATEMENT_CACHE(INITIAL_CAPACITY=2000, MAXIMUM_SIZE=65535)
);
```

保留字

ALTER、SQL_PARSER、RULE、PARSE_TREE_CACHE、INITIAL_CAPACITY、MAXIMUM_SIZE、SQL_STATEMENT_CACHE

相关链接

- 保留字

SHOW TRAFFIC RULE

描述

SHOW TRAFFIC RULE 语法用于查询指定的双路由规则。### 语法

```
ShowTrafficRule ::=
  'SHOW' 'TRAFFIC' ('RULES' | 'RULE' ruleName)?

ruleName ::=
  identifier
```

补充说明

- 未指定 ruleName 时，默认查询所有双路由规则

返回值说明

列	说明
name	双路由规则名称
labels	计算节点标签
algorithm_type	双路由算法类型
algorithm_props	双路由算法参数
load_balancer_type	负载均衡器类型
load_balancer_props	负载均衡器参数

示例

- 查询指定双路由规则

```
SHOW TRAFFIC RULE sql_match_traffic;
```

```
mysql> SHOW TRAFFIC RULE sql_match_traffic;
+-------------------+--------+----------------+-----------------------------------
-------------------------------------------+--------------------+-----------------
----+
| name              | labels | algorithm_type | algorithm_props
| load_balancer_type | load_balancer_props |
+-------------------+--------+----------------+-----------------------------------
-------------------------------------------+--------------------+-----------------
----+
| sql_match_traffic | OLTP   | SQL_MATCH      | sql=SELECT * FROM t_order WHERE
order_id = 1; UPDATE t_order SET order_id = 5; | RANDOM             |
```

```
|
+-------------------+--------+----------------+-----------------------------------
-------------------------------------------+--------------------+-----------------
----+
1 row in set (0.00 sec)
```

- 查询所有双路由规则

```
SHOW TRAFFIC RULES;
```

```
mysql> SHOW TRAFFIC RULES;
+-------------------+--------+----------------+-----------------------------------
-------------------------------------------+--------------------+-----------------
----+
| name              | labels | algorithm_type | algorithm_props
| load_balancer_type | load_balancer_props |
+-------------------+--------+----------------+-----------------------------------
-------------------------------------------+--------------------+-----------------
----+
| sql_match_traffic | OLTP   | SQL_MATCH      | sql=SELECT * FROM t_order WHERE
order_id = 1; UPDATE t_order SET order_id = 5; | RANDOM             |
|
+-------------------+--------+----------------+-----------------------------------
-------------------------------------------+--------------------+-----------------
----+
1 row in set (0.04 sec)
```

保留字

SHOW、TRAFFIC、RULE、RULES

相关链接

- 保留字

ALTER TRAFFIC RULE

描述

ALTER TRAFFIC RULE 语法用于修改双路由规则。

语法定义

```
AlterTrafficRule ::=
  'ALTER' 'TRAFFIC' 'RULE'  '(' 'LABELS' '(' lableName ')' ','
trafficAlgorithmDefinition ',' loadBalancerDefinition ')'

lableName ::=
  identifier

trafficAlgorithmDefinition ::=
  'TRAFFIC_ALGORITHM' '(' 'TYPE' '(' 'NAME' '=' trafficAlgorithmTypeName (','
propertiesDefinition)? ')' ')'

loadBalancerDefinition ::=
  'LOAD_BALANCER' '(' 'TYPE' '(' 'NAME' '=' loadBalancerName (','
propertiesDefinition)? ')' ')'

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

trafficAlgorithmTypeName ::=
  string

loadBalancerTypeName ::=
  string

key ::=
  string

value ::=
  literal
```

补充说明

- TRAFFIC_ALGORITHM 支持 SQL_MATCH 与 SQL_HINT 两种类型
- LOAD_BALANCER 支持 RANDOM 与 ROUND_ROBIN 两种类型

示例

- 修改双路由规则

```
ALTER TRAFFIC RULE sql_match_traffic (
  LABELS (OLTP),
  TRAFFIC_ALGORITHM(TYPE(NAME="SQL_MATCH",PROPERTIES("sql" = "SELECT * FROM t_order
WHERE order_id = 1; UPDATE t_order SET order_id = 5;"))),
  LOAD_BALANCER(TYPE(NAME="RANDOM")));
```

保留字

ALTER、TRAFFIC、RULE、LABELS、TYPE、NAME、PROPERTIES、TRAFFIC_ALGORITHM、LOAD_BALANCER

相关链接

- 保留字

SHOW SQL_FEDERATION RULE

描述

SHOW SQL_FEDERATION RULE 语法用于查询联邦查询配置。

语法

```
ShowSQLFederationRule ::=
  'SHOW' 'SQL_FEDERATION' 'RULE'
```

返回值说明

列	说明
sql_federation_enabled	是否开启联邦查询
all_query_use_sql_federation	是否全部查询 SQL 使用联邦查询
execution_plan_cache	执行计划缓存

示例

- 查询联邦查询配置

```
SHOW SQL_FEDERATION RULE;
```

```
mysql> show sql_federation rule;
+------------------------+-----------------------------+-------------------------
-----------------+
| sql_federation_enabled | all_query_use_sql_federation | execution_plan_cache
|
+------------------------+-----------------------------+-------------------------
-----------------+
| true                   | false                       | initialCapacity: 2000,
maximumSize: 65535 |
```

```
+------------------------+-----------------------------+-------------------------
-----------------+
1 row in set (0.31 sec)
```

保留字

SHOW、SQL_FEDERATION、RULE

相关链接

- 保留字

ALTER SQL_FEDERATION RULE

描述

ALTER SQL_FEDERATION RULE 语法用于修改联邦查询配置。

语法

```
AlterSQLFederationRule ::=
  'ALTER' 'SQL_FEDERATION' 'RULE' sqlFederationRuleDefinition

sqlFederationRuleDefinition ::=
  '(' sqlFederationEnabled? (','? allQueryUseSQLFederation)? (','?
executionPlanCache)? ')'

sqlFederationEnabled ::=
  'SQL_FEDERATION_ENABLED' '=' boolean_

allQueryUseSQLFederation ::=
  'ALL_QUERY_USE_SQL_FEDERATION' '=' boolean_

executionPlanCache ::=
  'EXECUTION_PLAN_CACHE' '(' cacheOption ')'

cacheOption ::=
  ('INITIAL_CAPACITY' '=' initialCapacity)? (',' 'MAXIMUM_SIZE' '=' maximumSize)?

initialCapacity ::=
  int

maximumSize ::=
  int
```

```
boolean_ ::=
  TRUE | FALSE
```

示例

- 修改联邦查询配置

```
ALTER SQL_FEDERATION RULE (
  SQL_FEDERATION_ENABLED=TRUE,
  ALL_QUERY_USE_SQL_FEDERATION=TRUE,
  EXECUTION_PLAN_CACHE(INITIAL_CAPACITY=1024, MAXIMUM_SIZE=65535)
);
```

保留字

ALTER、SQL_FEDERATION、RULE、SQL_FEDERATION_ENABLED、ALL_QUERY_USE_SQL_FEDERATION、EXECUTION_PLAN_CACHE、INITIAL_CAPACITY、MAXIMUM_SIZE

相关链接

- 保留字

熔断

本章节将对熔断功能的语法进行详细说明。

ALTER READWRITE_SPLITTING RULE ENABLE/DISABLE

描述

ALTER READWRITE_SPLITTING RULE ENABLE/DISABLE 语法用于启用/禁用指定逻辑库中指定读写分离规则中的指定读数据存储单元。

语法定义

```
AlterReadwriteSplittingRule ::=
  'ALTER' 'READWRITE_SPLITTING' 'RULE' groupName ('ENABLE' | 'DISABLE')
storageUnitName 'FROM' databaseName

groupName ::=
  identifier
```

```
storageUnitName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

示例

- 禁用指定逻辑库中指定读写分离规则中的指定读数据存储单元

```
ALTER READWRITE_SPLITTING RULE ms_group_0 DISABLE read_ds_0 FROM sharding_db;
```

- 启用指定逻辑库中指定读写分离规则中的指定读数据存储单元

```
ALTER READWRITE_SPLITTING RULE ms_group_0 ENABLE read_ds_0 FROM sharding_db;
```

- 禁用当前逻辑库中指定读写分离规则中的指定读数据存储单元

```
ALTER READWRITE_SPLITTING RULE ms_group_0 DISABLE read_ds_0;
```

- 启用当前逻辑库中指定读写分离规则中的指定读数据存储单元

```
ALTER READWRITE_SPLITTING RULE ms_group_1 ENABLE read_ds_0;
```

保留字

ALTER、READWRITE_SPLITTING、RULE、ENABLE、DISABLE

相关链接

- 保留字

SHOW STATUS FROM READWRITE_SPLITTING RULE

描述

`SHOW STATUS FROM READWRITE_SPLITTING RULE` 语法用于查询指定逻辑库中指定读写分离规则中读写分离存储单元状态。

语法

```
ShowStatusFromReadwriteSplittingRule ::=
  'SHOW' 'STATUS' 'FROM' 'READWRITE_SPLITTING' ('RULES' | 'RULE' groupName) ('FROM'
databaseName)?

groupName ::=
  identifier

databaseName ::=
  identifier
```

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`。

返回值说明

列	说明
storage_unit	存储单元名称
status	存储单元状态

示例

- 查询指定逻辑库中指定读写分离规则中读写分离存储单元状态

```
SHOW STATUS FROM READWRITE_SPLITTING RULE ms_group_0 FROM sharding_db;
```

```
mysql> SHOW STATUS FROM READWRITE_SPLITTING RULE ms_group_0 FROM sharding_db;
+--------------+----------+
| storage_unit | status   |
+--------------+----------+
| ds_0         | disabled |
```

```
+--------------+----------+
1 rows in set (0.01 sec)
```

- 查询指定逻辑库中所有读写分离存储单元状态

```
SHOW STATUS FROM READWRITE_SPLITTING RULES FROM sharding_db;
```

```
mysql> SHOW STATUS FROM READWRITE_SPLITTING RULES FROM sharding_db;
+--------------+----------+
| storage_unit | status   |
+--------------+----------+
| ds_0         | disabled |
+--------------+----------+
1 rows in set (0.00 sec)
```

- 查询当前逻辑库中指定读写分离规则中读写分离存储单元状态

```
SHOW STATUS FROM READWRITE_SPLITTING RULE ms_group_0;
```

```
mysql> SHOW STATUS FROM READWRITE_SPLITTING RULE ms_group_0;
+--------------+----------+
| storage_unit | status   |
+--------------+----------+
| ds_0         | disabled |
+--------------+----------+
1 rows in set (0.01 sec)
```

- 查询当前逻辑库中所有读写分离存储单元状态

```
mysql> SHOW STATUS FROM READWRITE_SPLITTING RULES;
```

```
mysql> SHOW STATUS FROM READWRITE_SPLITTING RULES;
+--------------+----------+
| storage_unit | status   |
+--------------+----------+
| ds_0         | disabled |
+--------------+----------+
1 rows in set (0.01 sec)
```

保留字

SHOW、STATUS、FROM、READWRITE_SPLITTING、RULE、RULES

相关链接

- 保留字

SHOW COMPUTE NODES

描述

SHOW COMPUTE NODES 语法用于查询计算节点信息。

语法

```
ShowComputeNodes ::=
  'SHOW' 'COMPUTE' 'NODES'
```

返回值说明

列	说明
instance_id	实例 id
instance_type	实例类型
host	主机地址
port	端口号
status	实例状态
mode_type	模式类型
worker_id	worker id
labels	标签
version	版本
database_name	当前连接逻辑库（仅对 JDBC 生效）

示例

```
mysql> SHOW COMPUTE NODES;
+--------------------------------------+---------------+------------+------+-------
-+------------+-----------+--------+----------+--------------+
| instance_id                          | instance_type | host       | port | status
| mode_type  | worker_id | labels | version  | database_name |
+--------------------------------------+---------------+------------+------+-------
-+------------+-----------+--------+----------+--------------+
| 3e84d33e-cb97-42f2-b6ce-f78fea0ded89 | PROXY         | 127.0.0.1  | 3307 | OK
| Cluster    | -1        |        | 5.4.2    | logic_db     |
+--------------------------------------+---------------+------------+------+-------
-+------------+-----------+--------+----------+--------------+
1 row in set (0.01 sec)
```

保留字

SHOW、COMPUTE、NODES

相关链接

- 保留字

ENABLE/DISABLE COMPUTE NODE

描述

ENABLE/DISABLE COMPUTE NODE 语法用于启用/禁用指定 proxy 实例。

语法定义

```
EnableDisableComputeNode ::=
  ('ENABLE' | 'DISABLE') 'COMPUTE' 'NODE' instanceId

instanceId ::=
  string
```

补充说明

- `instanceId` 需要通过 SHOW COMPUTE NODES 语法查询得到
- 不可禁用当前正在使用的 proxy 实例

示例

- 禁用指定 proxy 实例

```
DISABLE COMPUTE NODE '734bb086-b15d-4af0-be87-2372d8b6a0cd';
```

- 启用指定 proxy 实例

```
ENABLE COMPUTE NODE '734bb086-b15d-4af0-be87-2372d8b6a0cd';
```

保留字

`ENABLE`、`DISABLE`、`COMPUTE`、`NODE`

相关链接

- 保留字
- SHOW COMPUTE NODES

LABEL|RELABEL COMPUTE NODE

描述

`LABEL|RELABEL COMPUTE NODE` 语法用于为 `PROXY` 实例添加标签。

语法

```
LableRelabelComputeNodes ::=
  ('LABEL' | 'RELABEL') 'COMPUTE' 'NODE' instance_id 'WITH' labelName

instance_id ::=
  string

labelName ::=
  identifier
```

补充说明

- instance_id 需要通过 SHOW COMPUTE NODES 语法查询获得
- RELABEL 用于为 PROXY 实例修改标签

示例

- 为 PROXY 实例添加标签

```
LABEL COMPUTE NODE "0699e636-ade9-4681-b37a-65240c584bb3" WITH label_1;
```

- 为 PROXY 实例修改标签

```
RELABEL COMPUTE NODE "0699e636-ade9-4681-b37a-65240c584bb3" WITH label_2;
```

保留字

LABEL、RELABEL、COMPUTE、NODE 、WITH

相关链接

- 保留字
- SHOW COMPUTE NODES

UNLABEL COMPUTE NODE

描述

UNLABEL COMPUTE NODE 语法用于为 PROXY 实例去除指定标签。

语法

```
UnlabelComputeNode ::=
  'UNLABEL' 'COMPUTE' 'NODE' instance_id 'WITH' labelName

instance_id ::=
  string

labelName ::=
  identifier
```

补充说明

- `instance_id` 需要通过 SHOW COMPUTE NODES 语法查询获得

示例

- 为 PROXY 实例去除指定标签

```
UNLABEL COMPUTE NODE "0699e636-ade9-4681-b37a-65240c584bb3" WITH label_1;
```

保留字

`UNLABEL`、`COMPUTE`、`NODE`、`WITH`

相关链接

- 保留字
- SHOW COMPUTE NODES

数据迁移

本章节将对数据迁移功能的语法进行详细说明。

SHOW MIGRATION RULE

描述

`SHOW MIGRATION RULE` 语法用于查询数据迁移规则。### 语法

```
ShowMigrationRule ::=
  'SHOW' 'MIGRATION' 'RULE'
```

返回值说明

列	说明
read	数据读取配置
write	数据写入配置
stream_channel	数据通道

示例

- 查询数据迁移规则

```
SHOW MIGRATION RULE;
```

```
mysql> SHOW MIGRATION RULE;
+-------------------------------------------------------------+-------------------
------------------+-------------------------------------------------------+
| read                                                        | write
| stream_channel                                        |
+-------------------------------------------------------------+-------------------
------------------+-------------------------------------------------------+
| {"workerThread":20,"batchSize":1000,"shardingSize":10000000} | {"workerThread
":20,"batchSize":1000} | {"type":"MEMORY","props":{"block-queue-size":"2000"}} |
+-------------------------------------------------------------+-------------------
------------------+-------------------------------------------------------+
1 row in set (0.01 sec)
```

保留字

SHOW、MIGRATION、RULE

相关链接

- 保留字

ALTER MIGRATION RULE

描述

ALTER MIGRATION RULE 语法用于修改数据迁移规则。### 语法

```
AlterMigrationRule ::=
  'ALTER' 'MIGRATION' 'RULE' ('(' (readConfiguration ',')?  (writeConfiguration  ',
')? (dataChannel)? ')')?

readConfiguration ::=
  'READ' '(' ('WORKER_THREAD' '=' workerThreadPoolSize ',')? ('BATCH_SIZE' '='
batchSize ',')? ('SHARDING_SIZE' '=' shardingSize ',')? (rateLimiter)? ')'

writeConfiguration ::=
  'WRITE' '(' ('WORKER_THREAD' '=' workerThreadPoolSize ',')? ('BATCH_SIZE' '='
batchSize ',')? ('SHARDING_SIZE' '=' shardingSize ',')? (rateLimiter)? ')'
```

```
dataChannel ::=
  'STREAM_CHANNEL' '(' 'TYPE' '(' 'NAME' '=' algorithmName ',' propertiesDefinition
')' ')'

workerThreadPoolSize ::=
  int

batchSize ::=
  int

shardingSize ::=
  int

rateLimiter ::=
  'RATE_LIMITER' '(' 'TYPE' '(' 'NAME' '=' algorithmName ',' propertiesDefinition
')' ')'

algorithmName ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' key '=' value (',' key '=' value)* ')'

key ::=
  string

value ::=
  literal
```

示例

```
ALTER MIGRATION RULE (
  READ( WORKER_THREAD=20, BATCH_SIZE=1000, SHARDING_SIZE=10000000, RATE_LIMITER
(TYPE(NAME='QPS',PROPERTIES('qps'='500')))),
  WRITE( WORKER_THREAD=20, BATCH_SIZE=1000, RATE_LIMITER (TYPE(NAME='TPS',
PROPERTIES('tps'='2000')))),
  STREAM_CHANNEL ( TYPE(NAME='MEMORY',PROPERTIES('block-queue-size'='2000')))
  );
```

保留字

ALTER、MIGRATION、RULE、READ、WRITE、WORKER_THREAD、BATCH_SIZE、SHARDING_SIZE、STREAM_CHANNEL、TYPE、NAME、PROPERTIES

相关链接

- 保留字

REGISTER MIGRATION SOURCE STORAGE UNIT

描述

REGISTER MIGRATION SOURCE STORAGE UNIT 语法用于为当前连接注册数据迁移源存储单元。

语法

```
RegisterStorageUnit ::=
  'REGISTER' 'MIGRATION' 'SOURCE' 'STORAGE' 'UNIT' storageUnitDefinition (','
storageUnitDefinition)*

storageUnitDefinition ::=
  StorageUnitName '('  'URL' '=' url ',' 'USER' '=' user (',' 'PASSWORD' '='
password)? (',' propertiesDefinition)?')'

storageUnitName ::=
  identifier

url ::=
  string

user ::=
  string

password ::=
  string

propertiesDefinition ::=
  'PROPERTIES' '(' ( key  '=' value ) ( ',' key  '=' value )* ')'

key ::=
  string

value ::=
  literal
```

特别说明

- 确认注册的数据迁移源存储单元是可以正常连接的，否则将不能注册成功；
- storageUnitName 区分大小写；
- storageUnitName 在当前连接中需要唯一；
- storageUnitName 命名只允许使用字母、数字以及 _ ，且必须以字母开头；
- poolProperty 用于自定义连接池参数，key 必须和连接池参数名一致，value 支持 int 和 String 类型；
- 当 password 包含特殊字符时，建议使用 string 形式；例如 password@123 的 string 形式为 "password@123"；
- 数据迁移源存储单元暂时仅支持使用 URL 注册，暂时不支持使用 HOST 和 PORT。

示例

- 注册数据迁移源存储单元

```
REGISTER MIGRATION SOURCE STORAGE UNIT ds_0 (
    URL="jdbc:mysql://127.0.0.1:3306/migration_ds_0?serverTimezone=UTC&useSSL=false
",
    USER="root",
    PASSWORD="123456"
);
```

- 注册数据迁移源存储单元并设置连接池参数

```
REGISTER MIGRATION SOURCE STORAGE UNIT ds_0 (
    URL="jdbc:mysql://127.0.0.1:3306/migration_ds_0?serverTimezone=UTC&useSSL=false
",
    USER="root",
    PASSWORD="123456",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);
```

保留字

REGISTER、MIGRATION、SOURCE、STORAGE、UNIT、USER、PASSWORD、PROPERTIES、URL

相关链接

- 保留字

UNREGISTER MIGRATION SOURCE STORAGE UNIT

描述

UNREGISTER MIGRATION SOURCE STORAGE UNIT 语法用于从当前逻辑库中移除存储单元。

语法

```
UnregisterMigrationSourceStorageUnit ::=
  'UNREGISTER' 'MIGRATION' 'SOURCE' 'STORAGE' 'UNIT' storageUnitName  (','
storageUnitName)*

storageUnitName ::=
  identifier
```

补充说明

- UNREGISTER MIGRATION SOURCE STORAGE UNIT 只会移除 Proxy 中的存储单元, 不会删除与存储单元对应的真实数据源;

示例

- 移除数据迁移源存储单元

```
UNREGISTER MIGRATION SOURCE STORAGE UNIT ds_0;
```

- 移除多个数据迁移源存储单元

```
UNREGISTER MIGRATION SOURCE STORAGE UNIT ds_0, ds_1;
```

保留字

UNREGISTER、MIGRATION、SOURCE、STORAGE、UNIT

相关链接

- 保留字

SHOW MIGRATION SOURCE STORAGE UNITS

描述

`SHOW MIGRATION SOURCE STORAGE UNITS` 语法用于查询已经注册的数据迁移源存储单元。

语法

```
ShowStorageUnit ::=
  'SHOW' 'MIGRATION' 'SOURCE' 'STORAGE' 'UNITS'
```

返回值说明

列	说明
name	存储单元名称
type	存储单元类型
host	存储单元地址
port	存储单元端口
db	数据库名称
attribute	存储单元参数

示例

- 查询指定逻辑库中未被使用的存储单元

```
SHOW MIGRATION SOURCE STORAGE UNITS;
```

```
mysql> SHOW MIGRATION SOURCE STORAGE UNITS;
+------+-------+-----------+------+----------------+-----------------------------
--+---------------------------+---------------------------+---------------+--------
-------+-----------+------------------+
| name | type  | host      | port | db             | connection_timeout_
milliseconds | idle_timeout_milliseconds | max_lifetime_milliseconds | max_pool_
size | min_pool_size | read_only | other_attributes |
+------+-------+-----------+------+----------------+-----------------------------
--+---------------------------+---------------------------+---------------+--------
-------+-----------+------------------+
| ds_1 | MySQL | 127.0.0.1 | 3306 | migration_ds_0 |
```

```
|                           |                           |               |
|           |                  |
+------+-------+-----------+------+----------------+-------------------------------
--+---------------------------+---------------------------+---------------+--------
-------+-----------+------------------+
1 row in set (0.01 sec)
```

保留字

SHOW、MIGRATION、SOURCE、STORAGE、UNITS

相关链接

- 保留字

MIGRATE TABLE INTO

描述

MIGRATE TABLE INTO 语法用于将表从源端迁移到目标端。

语法

```
MigrateTableInto ::=
  'MIGRATE' 'TABLE' migrationSource '.' tableName 'INTO' (databaseName '.')?
tableName

migrationSource ::=
  identifier

databaseName ::=
  identifier

tableName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

示例

- 将表从源端迁移到当前逻辑库

```
MIGRATE TABLE ds_0.t_order INTO t_order;
```

- 将表从源端迁移到指定逻辑库

```
MIGRATE TABLE ds_0.t_order INTO sharding_db.t_order;
```

保留字

MIGRATE、TABLE、INTO

相关链接

- 保留字

SHOW MIGRATION LIST

描述

SHOW MIGRATION LIST 语法用于查询数据迁移作业列表。

语法

```
ShowMigrationList ::=
  'SHOW' 'MIGRATION' 'LIST'
```

返回值说明

列	说明
id	数据迁移作业 ID
tables	迁移表
active	数据迁移作业状态
create_time	数据迁移作业创建时间
stop_time	数据迁移作业停止时间
job_item_count	数据迁移作业分片数量
job_sharding_nodes	数据迁移作业分片运行节点

示例

- 查询数据迁移作业列表

```
SHOW MIGRATION LIST;
```

```
mysql> SHOW MIGRATION LIST;
+------------------------------------------+---------------------+--------+------
---------------+-----------+----------------+--------------------+
| id                                       | tables              | active |
create_time         | stop_time | job_item_count | job_sharding_nodes |
+------------------------------------------+---------------------+--------+------
---------------+-----------+----------------+--------------------+
| j0102p00001d029afca1fd960d567fed6cddc9b4a2 | source_ds.t_order   | true   | 2022-
10-31 18:18:24 |           | 1              | 10.7.5.76@-@27808  |
+------------------------------------------+---------------------+--------+------
---------------+-----------+----------------+--------------------+
4 rows in set (0.06 sec)
```

保留字

SHOW、MIGRATION、LIST

相关链接

- 保留字

SHOW MIGRATION STATUS

描述

`SHOW MIGRATION STATUS` 语法用于查询指定数据迁移作业的详细情况。

语法

```
ShowMigrationStatus ::=
  'SHOW' 'MIGRATION' 'STATUS' migrationJobId

migrationJobId ::=
  string
```

补充说明

- `migrationJobId` 需要通过 `SHOW MIGRATION LIST` 语法查询获得

返回值说明

列	说明
item	数据迁移作业分片编号
data source	数据迁移源
status	数据迁移作业状态
processed_records_count	处理数据行数
inventory_finished_percentage	数据迁移作业完成度
incremental_idle_seconds	增量闲置时间
error_message	错误信息提示

示例

- 查询指定数据迁移作业的详细情况

```
SHOW MIGRATION STATUS 'j010180026753ef0e25d3932d94d1673ba551';
```

```
mysql> SHOW MIGRATION STATUS 'j010180026753ef0e25d3932d94d1673ba551';
+------+-------------+--------------------------+--------+------------------------
+------------------------------+--------------------------+---------------+
| item | data_source | status                   | active | processed_records_count
| inventory_finished_percentage | incremental_idle_seconds | error_message |
+------+-------------+--------------------------+--------+------------------------
+------------------------------+--------------------------+---------------+
```

```
| 0    | ds_1        | EXECUTE_INCREMENTAL_TASK | true    | 6
| 100                            | 25                        |                |
+------+-------------+--------------------------+---------+-------------------------
+------------------------------+--------------------------+---------------+
1 row in set (0.01 sec)
```

保留字

SHOW、MIGRATION、STATUS

相关链接

- 保留字

SHOW MIGRATION CHECK ALGORITHM

描述

SHOW MIGRATION CHECK ALGORITHM 语法用于查询数据迁移一致性校验算法。### 语法

```
ShowMigrationCheckAlgorithm ::=
  'SHOW' 'MIGRATION' 'CHECK' 'ALGORITHMS'
```

返回值说明

列	说明
type	一致性校验算法类型
supported_database_types	支持数据库类型
description	说明

示例

- 查询数据迁移一致性校验算法

```
SHOW MIGRATION CHECK ALGORITHMS;
```

```
mysql> SHOW MIGRATION CHECK ALGORITHMS;
+-------------+-------------------------------------------------------------+-----
------------------------+
| type        | supported_database_types                                    |
description                |
```

```
+-------------+-----------------------------------------------------------------+-----
-----------------------+
| CRC32_MATCH | MySQL                                                           |
Match CRC32 of records.    |
| DATA_MATCH  | SQL92,MySQL,MariaDB,PostgreSQL,openGauss,Oracle,SQLServer,H2 |
Match raw data of records. |
+-------------+-----------------------------------------------------------------+-----
-----------------------+
2 rows in set (0.03 sec)
```

保留字

SHOW、MIGRATION、CHECK、ALGORITHMS

相关链接

- 保留字

CHECK MIGRATION BY

描述

CHECK MIGRATION BY 语法用于校验数据迁移作业中的数据一致性。

语法

```
ShowMigrationList ::=
  'CHECK' 'MIGRATION' migrationJobId 'BY' 'TYPE' '(' 'NAME' '='
migrationCheckAlgorithmType ')'

migrationJobId ::=
  string

migrationCheckAlgorithmType ::=
  string
```

补充说明

- `migrationJobId` 需要通过 SHOW MIGRATION LIST 语法查询获得
- `migrationCheckAlgorithmType` 需要通过 SHOW MIGRATION CHECK ALGORITHMS 语法查询获得

示例

- 校验数据迁移作业中数据一致性

```
CHECK MIGRATION 'j01016e501b498ed1bdb2c373a2e85e2529a6' BY TYPE (NAME='CRC32_MATCH
');
```

保留字

`CHECK`、`MIGRATION`、`BY`、`TYPE`

相关链接

- 保留字
- SHOW MIGRATION LIST
- SHOW MIGRATION CHECK ALGORITHMS

SHOW MIGRATION CHECK STATUS

描述

`SHOW MIGRATION CHECK STATUS` 语法用于查询指定数据迁移作业的数据校验情况。

语法

```
ShowMigrationCheckStatus ::=
  'SHOW' 'MIGRATION' 'CHECK' 'STATUS' migrationJobId

migrationJobId ::=
  string
```

补充说明

- `migrationJobId` 需要通过 SHOW MIGRATION LIST 语法查询获得

返回值说明

列	说明
tables	校验表
result	校验结果
finished_percentage	校验完成度
remaining_seconds	剩余时间
check_begin_time	校验开始时间
check_end_time	校验结束时间
error_message	错误信息提示

示例

- 查询指定数据迁移作业的数据校验情况

```
SHOW MIGRATION CHECK STATUS 'j010180026753ef0e25d3932d94d1673ba551';
```

```
mysql> SHOW MIGRATION CHECK STATUS 'j010180026753ef0e25d3932d94d1673ba551';
+---------+--------+---------------------+------------------+---------------------
----+-------------------------+------------------+---------------+
| tables  | result | finished_percentage | remaining_seconds | check_begin_time
| check_end_time          | duration_seconds | error_message |
+---------+--------+---------------------+------------------+---------------------
----+-------------------------+------------------+---------------+
| t_order | true   | 100                 | 0                 | 2022-11-01 17:57:39.
940 | 2022-11-01 17:57:40.587 | 0                |               |
+---------+--------+---------------------+------------------+---------------------
----+-------------------------+------------------+---------------+
1 row in set (0.01 sec)
```

保留字

SHOW、MIGRATION、CHECK、STATUS

相关链接

- 保留字
- SHOW MIGRATION LIST

START MIGRATION CHECK

描述

`START MIGRATION CHECK` 语法用于开始指定数据迁移作业的数据校验。

语法

```
StartMigrationCheck ::=
  'START' 'MIGRATION' 'CHECK' migrationJobId

migrationJobId ::=
  string
```

补充说明

- `migrationJobId` 需要通过 SHOW MIGRATION LIST 语法查询获得

示例

- 开始指定数据迁移作业的数据校验

```
START MIGRATION CHECK 'j010180026753ef0e25d3932d94d1673ba551';
```

保留字

`START`、`MIGRATION`、`CHECK`

相关链接

- 保留字
- SHOW MIGRATION LIST

STOP MIGRATION CHECK

描述

`STOP MIGRATION CHECK` 语法用于停止指定数据迁移作业的数据校验。

语法

```
StopMigrationCheck ::=
  'STOP' 'MIGRATION' 'CHECK' migrationJobId

migrationJobId ::=
  string
```

补充说明

- `migrationJobId` 需要通过 SHOW MIGRATION LIST 语法查询获得

示例

- 停止指定数据迁移作业的数据校验

```
STOP MIGRATION CHECK 'j010180026753ef0e25d3932d94d1673ba551';
```

保留字

`STOP`、`MIGRATION`、`CHECK`

相关链接

- 保留字
- SHOW MIGRATION LIST

START MIGRATION

描述

`START MIGRATION` 语法用于开始指定的数据迁移作业。

语法

```
StartMigration ::=
  'START' 'MIGRATION'  migrationJobId

migrationJobId ::=
  string
```

补充说明

- `migrationJobId` 需要通过 SHOW MIGRATION LIST 语法查询获得

示例

- 停止指定的数据迁移作业

```
START MIGRATION 'j010180026753ef0e25d3932d94d1673ba551';
```

保留字

`START`、`MIGRATION`

相关链接

- 保留字
- SHOW MIGRATION LIST

STOP MIGRATION

描述

`STOP MIGRATION` 语法用于停止指定的数据迁移作业。

语法

```
StopMigration ::=
  'STOP' 'MIGRATION'  migrationJobId

migrationJobId ::=
  string
```

补充说明

- `migrationJobId` 需要通过 SHOW MIGRATION LIST 语法查询获得

示例

- 停止指定的数据迁移作业

```
STOP MIGRATION 'j010180026753ef0e25d3932d94d1673ba551';
```

保留字

`STOP`、`MIGRATION`

相关链接

- 保留字
- SHOW MIGRATION LIST

COMMIT MIGRATION

描述

`COMMIT MIGRATION` 语法用于完成指定的数据迁移作业。

语法

```
CommitMigration ::=
  'COMMIT' 'MIGRATION'  migrationJobId

migrationJobId ::=
  string
```

补充说明

- `migrationJobId` 需要通过 SHOW MIGRATION LIST 语法查询获得

示例

- 完成指定的数据迁移作业

```
COMMIT MIGRATION 'j010180026753ef0e25d3932d94d1673ba551';
```

保留字

COMMIT、MIGRATION

相关链接

- 保留字
- SHOW MIGRATION LIST

ROLLBACK MIGRATION

描述

ROLLBACK MIGRATION 语法用于撤销指定的数据迁移作业。

语法

```
RollbackMigration ::=
  'ROLLBACK' 'MIGRATION'  migrationJobId

migrationJobId ::=
  string
```

补充说明

- migrationJobId 需要通过 SHOW MIGRATION LIST 语法查询获得
- 该语句执行后会清理目标端

示例

- 撤销指定的数据迁移作业

```
ROLLBACK MIGRATION 'j010180026753ef0e25d3932d94d1673ba551';
```

保留字

ROLLBACK、MIGRATION

相关链接

- 保留字
- SHOW MIGRATION LIST

插件

本章节将对插件相关语法进行详细说明。

SHOW PLUGINS

描述

SHOW PLUGINS OF interfaceClass 语法用于查询指定接口的全部实现。

语法

```
showPluginImplementations ::=
  'SHOW' 'PLUGINS' 'OF' interfaceClass

interfaceClass ::=
  string
```

返回值说明

列	说明
type	类型
type_aliases	类型别名
description	描述

示例

- 查询 org.apache.shardingsphere.sharding.spi.ShardingAlgorithm 接口的所有实现类

```
SHOW PLUGINS OF 'org.apache.shardingsphere.sharding.spi.ShardingAlgorithm'
```

```
SHOW PLUGINS OF 'org.apache.shardingsphere.sharding.spi.ShardingAlgorithm';
+----------------+--------------+-------------+
| type           | type_aliases | description |
+----------------+--------------+-------------+
| MOD            |              |             |
| HASH_MOD       |              |             |
| VOLUME_RANGE   |              |             |
| BOUNDARY_RANGE |              |             |
| AUTO_INTERVAL  |              |             |
| INTERVAL       |              |             |
| CLASS_BASED    |              |             |
| INLINE         |              |             |
| COMPLEX_INLINE |              |             |
| HINT_INLINE    |              |             |
+----------------+--------------+-------------+
10 rows in set (0.52 sec)
```

补充说明

针对一些常用的接口，ShardingSphere 提供了语法糖，可以简化操作，目前已提供的插件查询语法糖如下：

- 查询 org.apache.shardingsphere.sharding.spi.ShardingAlgorithm 接口实现：SHOW SHARDING ALGORITHM PLUGINS
- 查询 org.apache.shardingsphere.infra.algorithm.loadbalancer.core.LoadBalanceAlgorithm 接口实现：SHOW LOAD BALANCE ALGORITHM PLUGINS
- 查询 org.apache.shardingsphere.encrypt.spi.EncryptAlgorithm 接口实现：SHOW ENCRYPT ALGORITHM PLUGINS
- 查询 org.apache.shardingsphere.mask.spi.MaskAlgorithm 接口实现：SHOW MASK ALGORITHM PLUGINS
- 查询 org.apache.shardingsphere.shadow.spi.ShadowAlgorithm 接口实现：SHOW SHADOW ALGORITHM PLUGINS
- 查询 org.apache.shardingsphere.keygen.core.algorithm.KeyGenerateAlgorithm 接口实现：SHOW KEY GENERATE ALGORITHM PLUGINS

保留字

SHOW、PLUGINS、OF

相关链接

- 保留字

SHOW SHARDING ALGORITHM PLUGINS

描述

SHOW SHARDING ALGORITHM PLUGINS 语法用于查询 org.apache.shardingsphere.sharding.spi.ShardingAlgorithm 接口的所有实现类。

语法

```
showShardingAlgorithmPlugins ::=
  'SHOW' 'SHARDING' 'ALGORITHM' 'PLUGINS'
```

返回值说明

列	说明
type	类型
type_aliases	类型别名
description	描述

示例

- 查询 org.apache.shardingsphere.sharding.spi.ShardingAlgorithm 接口的所有实现类

```
SHOW SHARDING ALGORITHM PLUGINS
```

```
SHOW SHARDING ALGORITHM PLUGINS;
+----------------+--------------+-------------+
| type           | type_aliases | description |
+----------------+--------------+-------------+
| MOD            |              |             |
| HASH_MOD       |              |             |
| VOLUME_RANGE   |              |             |
```

```
| BOUNDARY_RANGE |              |             |
| AUTO_INTERVAL  |              |             |
| INTERVAL       |              |             |
| CLASS_BASED    |              |             |
| INLINE         |              |             |
| COMPLEX_INLINE |              |             |
| HINT_INLINE    |              |             |
+----------------+--------------+-------------+
10 rows in set (0.27 sec)
```

保留字

SHOW、SHARDING、ALGORITHM、PLUGINS

相关链接

- 保留字

SHOW LOAD BALANCE ALGORITHM PLUGINS

描述

SHOW LOAD BALANCE ALGORITHM PLUGINS 语法用于查询 org.apache.shardingsphere.infra.algorithm.loadbalancer.core.LoadBalanceAlgorithm 接口的所有实现类。

语法

```
showLoadBalanceAlgorithmPlugins ::=
  'SHOW' 'LOAD' 'BALANCE' 'ALGORITHM' 'PLUGINS'
```

返回值说明

列	说明
type	类型
type_aliases	类型别名
description	描述

示例

- 查 询 org.apache.shardingsphere.infra.algorithm.loadbalancer.core.LoadBalanceAlgorithm 接口的所有实现类

```
SHOW LOAD BALANCE ALGORITHM PLUGINS
```

```
SHOW LOAD BALANCE ALGORITHM PLUGINS;
+-------------+--------------+-------------+
| type        | type_aliases | description |
+-------------+--------------+-------------+
| ROUND_ROBIN |              |             |
| RANDOM      |              |             |
| WEIGHT      |              |             |
+-------------+--------------+-------------+
3 rows in set (0.03 sec)
```

保留字

SHOW、LOAD、BALANCE、ALGORITHM、PLUGINS

相关链接

- 保留字

SHOW ENCRYPT ALGORITHM PLUGINS

描述

SHOW ENCRYPT ALGORITHM PLUGINS 语法用于查询 org.apache.shardingsphere.encrypt.spi.EncryptAlgorithm 接口的所有实现类。

语法

```
showEncryptAlgorithmPlugins ::=
  'SHOW' 'ENCRYPT' 'ALGORITHM' 'PLUGINS'
```

返回值说明

列	说明
type	类型
type_aliases	类型别名
description	描述

示例

- 查询 `org.apache.shardingsphere.encrypt.spi.EncryptAlgorithm` 接口的所有实现类

```
SHOW ENCRYPT ALGORITHM PLUGINS
```

```
SHOW ENCRYPT ALGORITHM PLUGINS;
+------+--------------+-------------+
| type | type_aliases | description |
+------+--------------+-------------+
| AES  |              |             |
| MD5  |              |             |
+------+--------------+-------------+
2 rows in set (0.06 sec)
```

保留字

`SHOW`、`ENCRYPT`、`ALGORITHM`、`PLUGINS`

相关链接

- 保留字

SHOW MASK ALGORITHM PLUGINS

描述

`SHOW MASK ALGORITHM PLUGINS` 语法用于查询 `org.apache.shardingsphere.mask.spi.MaskAlgorithm` 接口的所有实现类。

语法

```
showMaskAlgorithmPlugins ::=
  'SHOW' 'MASK' 'ALGORITHM' 'PLUGINS'
```

返回值说明

列	说明
type	类型
type_aliases	类型别名
description	描述

示例

- 查询 `org.apache.shardingsphere.mask.spi.MaskAlgorithm` 接口的所有实现类

```
SHOW MASK ALGORITHM PLUGINS
```

```
SHOW MASK ALGORITHM PLUGINS;
+------------------------------+--------------+-------------+
| type                         | type_aliases | description |
+------------------------------+--------------+-------------+
| MD5                          |              |             |
| KEEP_FIRST_N_LAST_M          |              |             |
| KEEP_FROM_X_TO_Y             |              |             |
| MASK_AFTER_SPECIAL_CHARS     |              |             |
| MASK_BEFORE_SPECIAL_CHARS    |              |             |
| MASK_FIRST_N_LAST_M          |              |             |
| MASK_FROM_X_TO_Y             |              |             |
| GENERIC_TABLE_RANDOM_REPLACE |              |             |
+------------------------------+--------------+-------------+
8 rows in set (0.13 sec)
```

保留字

SHOW、MASK、ALGORITHM、PLUGINS

相关链接

- 保留字

SHOW SHADOW ALGORITHM PLUGINS

描述

`SHOW SHADOW ALGORITHM PLUGINS` 语法用于查询 `org.apache.shardingsphere.shadow.spi.ShadowAlgorithm` 接口的所有实现类。

语法

```
showShadowAlgorithmPlugins ::=
  'SHOW' 'SHADOW' 'ALGORITHM' 'PLUGINS'
```

返回值说明

列	说明
type	类型
type_aliases	类型别名
description	描述

示例

- 查询 `org.apache.shardingsphere.shadow.spi.ShadowAlgorithm` 接口的所有实现类

```
SHOW SHADOW ALGORITHM PLUGINS
```

```
SHOW SHADOW ALGORITHM PLUGINS;
+-------------+--------------+-------------+
| type        | type_aliases | description |
+-------------+--------------+-------------+
| SQL_HINT    |              |             |
| REGEX_MATCH |              |             |
| VALUE_MATCH |              |             |
+-------------+--------------+-------------+
3 rows in set (0.37 sec)
```

保留字

SHOW、SHADOW、ALGORITHM、PLUGINS

相关链接

- 保留字

SHOW KEY GENERATE ALGORITHM PLUGINS

描述

SHOW KEY GENERATE ALGORITHM PLUGINS 语法用于查询 org.apache.shardingsphere.keygen.core.algorithm.KeyGenerateAlgorithm 接口的所有实现类。

语法

```
showKeyGenerateAlgorithmPlugins ::=
  'SHOW' 'KEY' 'GENERATE' 'ALGORITHM' 'PLUGINS'
```

返回值说明

列	说明
type	类型
type_aliases	类型别名
description	描述

示例

- 查询 org.apache.shardingsphere.keygen.core.algorithm.KeyGenerateAlgorithm 接口的所有实现类

```
SHOW KEY GENERATE ALGORITHM PLUGINS
```

```
SHOW KEY GENERATE ALGORITHM PLUGINS;
+-----------+--------------+-------------+
| type      | type_aliases | description |
+-----------+--------------+-------------+
| UUID      |              |             |
| SNOWFLAKE |              |             |
```

```
+-----------+--------------+-------------+
2 rows in set (0.05 sec)
```

保留字

SHOW、KEY、GENERATE、ALGORITHM、PLUGINS

相关链接

- 保留字

SHOW COMPUTE NODE INFO

描述

SHOW COMPUTE NODE INFO 语法用于查询当前 proxy 实例信息。

语法

```
ShowComputeNodeInfo ::=
  'SHOW' 'COMPUTE' 'NODE' 'INFO'
```

返回值说明

列	说明
instance_id	proxy 实例编号
host	主机地址
port	端口号
status	proxy 实例状态
mode_type	proxy 实例模式
worker_id	worker id
labels	标签
version	版本

示例

- 查询当前 proxy 实例信息

```
SHOW COMPUTE NODE INFO;
```

```
mysql> SHOW COMPUTE NODE INFO;
mysql> SHOW COMPUTE NODES;
+--------------------------------------+---------------+------------+------+-------
-+------------+-----------+--------+----------+
| instance_id                          | instance_type | host       | port | status
| mode_type  | worker_id | labels | version  |
+--------------------------------------+---------------+------------+------+-------
-+------------+-----------+--------+----------+
| 3e84d33e-cb97-42f2-b6ce-f78fea0ded89 | PROXY         | 127.0.0.1  | 3307 | OK
| Cluster    | -1        |        | 5.4.2    |
+--------------------------------------+---------------+------------+------+-------
-+------------+-----------+--------+----------+
1 row in set (0.01 sec)
```

保留字

SHOW、COMPUTE、NODE、INFO

相关链接

- 保留字

SHOW COMPUTE NODE MODE

描述

SHOW COMPUTE NODE MODE 语法用于查询当前 proxy 的模式配置信息。

语法

```
ShowComputeNodeMode ::=
  'SHOW' 'COMPUTE' 'NODE' 'MODE'
```

返回值说明

列	说明
type	proxy 模式类型
repository	proxy 持久化仓库类型
props	proxy 持久化仓库属性参数

示例

- 查询当前 proxy 实例模式配置信息

```
SHOW COMPUTE NODE MODE;
```

```
mysql> SHOW COMPUTE NODE MODE;
+---------+------------+-------------------------------------------------------------
-----------------------------------------------------------------------------------
--------------------------+
| type    | repository | props
|
+---------+------------+-------------------------------------------------------------
-----------------------------------------------------------------------------------
--------------------------+
| Cluster | ZooKeeper  | {"operationTimeoutMilliseconds":500,"timeToLiveSeconds
":60,"maxRetries":3,"namespace":"governance_ds","server-lists":"localhost:2181",
"retryIntervalMilliseconds":500} |
+---------+------------+-------------------------------------------------------------
-----------------------------------------------------------------------------------
--------------------------+
1 row in set (0.00 sec)
```

保留字

SHOW、COMPUTE、NODE、MODE

相关链接

- 保留字

SET DIST VARIABLE

描述

SET DIST VARIABLE 语法用于设置系统变量。

语法

```
SetDistVariable ::=
  'SET' 'DIST' 'VARIABLE' (proxyPropertyName '=' proxyPropertyValue | 'agent_
plugins_enabled' '=' agentPluginsEnabled)

proxyPropertyName ::=
  identifier

proxyPropertyValue ::=
  literal

agentPluginsEnabled ::=
  boolean
```

补充说明

- proxy_property_name 为 PROXY 的属性配置，需使用下划线命名
- agent_plugins_enabled 为 agent 插件的启用状态，默认值 FALSE

示例

- 设置 Proxy 属性配置

```
SET DIST VARIABLE sql_show = true;
```

- 设置 agent 插件启用状态

```
SET DIST VARIABLE agent_plugins_enabled = TRUE;
```

保留字

SET、DIST、VARIABLE

相关链接

- 保留字

SHOW DIST VARIABLE

描述

SHOW DIST VARIABLE 语法用于查询 PROXY 系统变量配置。

语法

```
ShowDistVariable ::=
  'SHOW' 'DIST' ('VARIABLES' ('LIKE' likePattern)?| 'VARIABLE' 'WHERE' 'NAME' '='
variableName)

likePattern ::=
  string

variableName ::=
  identifier
```

返回值说明

列	说明
variable_name	系统变量名称
variable_value	系统变量值

补充说明

- 未指定 vairableName 时，默认查询所有 PROXY 系统变量配置

示例

- 查询所有 PROXY 系统变量配置

```
SHOW DIST VARIABLES;
```

```
mysql> SHOW DIST VARIABLES;
+----------------------------------------+-----------------+
| variable_name                          | variable_value  |
+----------------------------------------+-----------------+
| agent_plugins_enabled                  | true            |
| cached_connections                     | 0               |
| cdc_server_port                        | 33071           |
| check_table_metadata_enabled           | false           |
| kernel_executor_size                   | 0               |
| load_table_metadata_batch_size         | 1000            |
| max_connections_size_per_query         | 1               |
| proxy_backend_query_fetch_size         | -1              |
| proxy_default_port                     | 3307            |
| proxy_frontend_database_protocol_type  |                 |
| proxy_frontend_executor_size           | 0               |
| proxy_frontend_flush_threshold         | 128             |
| proxy_frontend_max_connections         | 0               |
| proxy_frontend_ssl_cipher              |                 |
| proxy_frontend_ssl_enabled             | false           |
| proxy_frontend_ssl_version             | TLSv1.2,TLSv1.3 |
| proxy_meta_data_collector_enabled      | false           |
| proxy_netty_backlog                    | 1024            |
| sql_show                               | false           |
| sql_simple                             | false           |
| system_schema_metadata_assembly_enabled | true            |
+----------------------------------------+-----------------+
21 rows in set (0.01 sec)
```

- 查询指定 PROXY 系统变量配置

```
SHOW DIST VARIABLE WHERE NAME = sql_show;
```

```
mysql> SHOW DIST VARIABLE WHERE NAME = sql_show;
+---------------+----------------+
| variable_name | variable_value |
+---------------+----------------+
| sql_show      | false          |
+---------------+----------------+
1 row in set (0.00 sec)
```

保留字

SHOW、DIST、VARIABLE、VARIABLES、NAME

相关链接

- 保留字

REFRESH TABLE METADATA

描述

REFRESH TABLE METADATA 语法用于刷新表元数据。

语法

```
RefreshTableMetadata ::=
  'REFRESH' 'TABLE' 'METADATA' (tableName | tableName 'FROM' 'STORAGE' 'UNIT'
storageUnitName ('SCHEMA' schemaName)?)?

tableName ::=
  identifier

storageUnitName ::=
  identifier

schemaName ::=
  identifier
```

补充说明

- 未指定 tableName 和 storageUnitName 时，默认刷新所有表的元数据；
- 刷新元数据需要使用 DATABASE 如果未使用 DATABASE 则会提示 No database selected；
- 如果 SCHEMA 中不存在表，则会删除该 SCHEMA。

示例

- 刷新指定存储单元中指定 SCHEMA 中指定表的元数据

```
REFRESH TABLE METADATA t_order FROM STORAGE UNIT ds_1 SCHEMA db_schema;
```

- 刷新指定存储单元中指定 SCHEMA 中所有表的元数据

```
REFRESH TABLE METADATA FROM STORAGE UNIT ds_1 SCHEMA db_schema;
```

- 刷新指定存储单元中指定表的元数据

```
REFRESH TABLE METADATA t_order FROM STORAGE UNIT ds_1;
```

- 刷新指定表的元数据

```
REFRESH TABLE METADATA t_order;
```

- 刷新所有表的元数据

```
REFRESH TABLE METADATA;
```

保留字

REFRESH、TABLE、METADATA、FROM、STORAGE、UNIT

相关链接

- 保留字

REFRESH DATABASE METADATA

描述

REFRESH DATABASE METADATA 语法用于刷新本地逻辑库元数据。

语法

```
RefreshDatabaseMetadata ::=
  'FORCE'? 'REFRESH' 'DATABASE' 'METADATA' databaseName?

databaseName ::=
  identifier
```

补充说明

- 未指定 databaseName 时，默认刷新所有逻辑库的元数据
- 刷新元数据使用 FORCE 时，将会从本地获取最新元数据，并写入到治理中心，未开启 FORCE 则会从治理中心拉取最新配置

示例

- 刷新指定逻辑库的元数据

```
REFRESH DATABASE METADATA sharding_db;
```

- 刷新所有逻辑库的元数据

```
REFRESH DATABASE METADATA;
```

- 强制刷新所有逻辑库的元数据

```
FORCE REFRESH DATABASE METADATA;
```

保留字

FORCE、REFRESH、DATABASE、METADATA

相关链接

- 保留字

SHOW TABLE METADATA

描述

SHOW TABLE METADATA 语法用于查询表的元数据。

语法

```
ShowTableMetadata ::=
  'SHOW' 'TABLE' 'METADATA' tableName (',' tableName)* ('FROM' databaseName)?

tableName ::=
  identifier

databaseName ::=
  identifier
```

返回值说明

列	说明
schema_name	逻辑库名称
table_name	表名称
type	元数据类型
name	元数据名称

补充说明

- 未指定 `databaseName` 时，默认是当前使用的 `DATABASE`。如果也未使用 `DATABASE` 则会提示 `No database selected`。

示例

- 查询指定逻辑库中多个表的元数据

```
SHOW TABLE METADATA t_order, t_order_1 FROM sharding_db;
```

```
mysql> SHOW TABLE METADATA t_order, t_order_1 FROM sharding_db;
+-------------------+------------+--------+----------+
| schema_name       | table_name | type   | name     |
+-------------------+------------+--------+----------+
| sharding_db       | t_order_1  | COLUMN | order_id |
| sharding_db       | t_order_1  | COLUMN | user_id  |
| sharding_db       | t_order_1  | COLUMN | status   |
| sharding_db       | t_order_1  | INDEX  | PRIMARY  |
| sharding_db       | t_order    | COLUMN | order_id |
| sharding_db       | t_order    | COLUMN | user_id  |
| sharding_db       | t_order    | COLUMN | status   |
| sharding_db       | t_order    | INDEX  | PRIMARY  |
+-------------------+------------+--------+----------+
8 rows in set (0.01 sec)
```

- 查询指定逻辑库中单个表的元数据

```
SHOW TABLE METADATA t_order FROM sharding_db;
```

```
mysql> SHOW TABLE METADATA t_order FROM sharding_db;
+-------------------+------------+--------+----------+
| schema_name       | table_name | type   | name     |
+-------------------+------------+--------+----------+
```

```
| sharding_db       | t_order    | COLUMN | order_id |
| sharding_db       | t_order    | COLUMN | user_id  |
| sharding_db       | t_order    | COLUMN | status   |
| sharding_db       | t_order    | INDEX  | PRIMARY  |
+-------------------+------------+--------+----------+
4 rows in set (0.00 sec)
```

- 查询当前逻辑库中多个表的元数据

```
SHOW TABLE METADATA t_order, t_order_1;
```

```
mysql> SHOW TABLE METADATA t_order, t_order_1;
+-------------------+------------+--------+----------+
| schema_name       | table_name | type   | name     |
+-------------------+------------+--------+----------+
| sharding_db       | t_order_1  | COLUMN | order_id |
| sharding_db       | t_order_1  | COLUMN | user_id  |
| sharding_db       | t_order_1  | COLUMN | status   |
| sharding_db       | t_order_1  | INDEX  | PRIMARY  |
| sharding_db       | t_order    | COLUMN | order_id |
| sharding_db       | t_order    | COLUMN | user_id  |
| sharding_db       | t_order    | COLUMN | status   |
| sharding_db       | t_order    | INDEX  | PRIMARY  |
+-------------------+------------+--------+----------+
8 rows in set (0.00 sec)
```

- 查询当前逻辑库中单个表的元数据

```
SHOW TABLE METADATA t_order;
```

```
mysql> SHOW TABLE METADATA t_order;
+-------------------+------------+--------+----------+
| schema_name       | table_name | type   | name     |
+-------------------+------------+--------+----------+
| sharding_db       | t_order    | COLUMN | order_id |
| sharding_db       | t_order    | COLUMN | user_id  |
| sharding_db       | t_order    | COLUMN | status   |
| sharding_db       | t_order    | INDEX  | PRIMARY  |
+-------------------+------------+--------+----------+
4 rows in set (0.01 sec)
```

保留字

SHOW、TABLE、METADATA、FROM

相关链接

- 保留字

SHOW RULES USED STORAGE UNIT

描述

SHOW RULES USED STORAGE UNIT 语法用于查询指定逻辑库中使用指定存储单元的规则。

语法

```
ShowRulesUsedStorageUnit ::=
  'SHOW' 'RULES' 'USED' 'STORAGE' 'UNIT' storageUnitName ('FROM' databaseName)?

storageUnitName ::=
  identifier

databaseName ::=
  identifier
```

返回值说明

列	说明
type	规则类型
name	规则名称

补充说明

- 未指定 databaseName 时，默认是当前使用的 DATABASE。如果也未使用 DATABASE 则会提示 No database selected。

示例

- 查询指定逻辑库中使用指定存储单元的规则

```
SHOW RULES USED STORAGE UNIT ds_1 FROM sharding_db;
```

```
mysql> SHOW RULES USED STORAGE UNIT ds_1 FROM sharding_db;
+---------------------+------------+
| type                | name       |
+---------------------+------------+
| readwrite_splitting | ms_group_0 |
| readwrite_splitting | ms_group_0 |
+---------------------+------------+
2 rows in set (0.01 sec)
```

- 查询当前逻辑库中使用指定存储单元的规则

```
SHOW RULES USED STORAGE UNIT ds_1;
```

```
mysql> SHOW RULES USED STORAGE UNIT ds_1;
+---------------------+------------+
| type                | name       |
+---------------------+------------+
| readwrite_splitting | ms_group_0 |
| readwrite_splitting | ms_group_0 |
+---------------------+------------+
2 rows in set (0.01 sec)
```

保留字

SHOW、RULES、USED、STORAGE、UNIT、FROM

相关链接

- 保留字

EXPORT DATABASE CONFIGURATION

描述

EXPORT DATABASE CONFIGURATION 语法用于将逻辑库中的存储单元和规则配置导出为 YAML 格式。

语法

```
ExportDatabaseConfiguration ::=
  'EXPORT' 'DATABASE' 'CONFIGURATION' ('FROM' databaseName)? ('TO' 'FILE'
filePath)?

databaseName ::=
  identifier

filePath ::=
  string
```

补充说明

- 未指定 `databaseName` 时，导出当前使用的逻辑库；如果也未使用逻辑库则提示 `No database selected`；
- 未指定 `filePath` 时，会将导出的信息通过结果集输出；
- 指定 `filePath` 时，会自动创建文件，若文件已存在，会被覆盖。

示例

- 导出当前逻辑库的配置信息

```
mysql> EXPORT DATABASE CONFIGURATION;
+-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| result
|
+-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| databaseName: sharding_db
dataSources:
  ds_1:
    password: 123456
    url: jdbc:mysql://127.0.0.1:3306/db0
```

```
    username: root
    minPoolSize: 1
    connectionTimeoutMilliseconds: 30000
    maxLifetimeMilliseconds: 2100000
    readOnly: false
    idleTimeoutMilliseconds: 60000
    maxPoolSize: 50
  ds_2:
    password: 123456
    url: jdbc:mysql://127.0.0.1:3306/db1
    username: root
    minPoolSize: 1
    connectionTimeoutMilliseconds: 30000
    maxLifetimeMilliseconds: 2100000
    readOnly: false
    idleTimeoutMilliseconds: 60000
    maxPoolSize: 50
rules:
 |
+-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

- 导出指定逻辑库的配置信息，并输出到文件

```
mysql> EXPORT DATABASE CONFIGURATION FROM sharding_db TO FILE '/xxx/config_
sharding_db.yaml';
+-----------------------------------------------------------------------+
| result                                                                |
+-----------------------------------------------------------------------+
| Successfully exported to: '/xxx/config_sharding_db.yaml'  |
+-----------------------------------------------------------------------+
1 row in set (0.02 sec)
```

保留字

EXPORT、DATABASE、CONFIGURATION、FROM、TO、FILE

相关链接

- 保留字

IMPORT DATABASE CONFIGURATION

描述

IMPORT DATABASE CONFIGURATION 语法用于从 YAML 中的配置导入逻辑库。

语法

```
ImportDatabaseConfiguration ::=
  'IMPORT' 'DATABASE' 'CONFIGURATION' 'FROM' 'FILE' filePath

filePath ::=
  string
```

补充说明

- 当元数据中已存在同名逻辑库时，无法导入；
- 当 YAML 中 databaseName 为空时，无法导入；
- 当 YAML 中 dataSources 为空时，只导入空的逻辑库。

示例

```
IMPORT DATABASE CONFIGURATION FROM FILE "/xxx/config_sharding_db.yaml";
```

保留字

IMPORT、DATABASE、CONFIGURATION、FROM、FILE

相关链接

- 保留字

CONVERT YAML CONFIGURATION

描述

`CONVERT YAML CONFIGURATION` 语法用于将 `YAML` 配置转换为对应的 DistSQL RDL 语句。

语法

```
convertYamlConfiguration ::=
  'CONVERT' 'YAML' 'CONFIGURATION' 'FROM' 'FILE' filePath

filePath ::=
  string
```

补充说明

- `CONVERT YAML CONFIGURATION` 语法仅读取 YAML 文件并将配置转换为 DistSQL 语句，不会影响当前元数据；
- 当 YAML 中 `dataSources` 为空时，不会进行 `rules` 的转换。

示例

```
mysql> CONVERT YAML CONFIGURATION FROM FILE '/xxx/config_sharding_db.yaml';
+-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------+
| dist_sql
|
+-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------------
-----------------------------------------------------+
| CREATE DATABASE sharding_db;
USE sharding_db;

REGISTER STORAGE UNIT ds_0 (
URL='jdbc:mysql://127.0.0.1:3306/demo_ds_0?serverTimezone=UTC&useSSL=false&
allowPublicKeyRetrieval=true',
USER='root',
PASSWORD='123456',
PROPERTIES('maxPoolSize'='10')
), ds_1 (
URL='jdbc:mysql://127.0.0.1:3306/demo_ds_1?serverTimezone=UTC&useSSL=false&
allowPublicKeyRetrieval=true',
USER='root',
PASSWORD='123456',
PROPERTIES('maxPoolSize'='10')
);

CREATE SHARDING TABLE RULE t_order (
STORAGE_UNITS(ds_0,ds_1),
SHARDING_COLUMN=order_id,
TYPE(NAME='mod', PROPERTIES('sharding-count'='4')),
KEY_GENERATE_STRATEGY(COLUMN=order_id, TYPE(NAME='snowflake'))
);

|
+---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
-----------------------------------------------------+
1 row in set (0.02 sec)
```

保留字

CONVERT、YAML、CONFIGURATION、FROM、FILE

相关链接

- 保留字

LOCK CLUSTER WITH

描述

LOCK CLUSTER WITH 语法用于向 CLUSTER 施加特定算法的锁。

语法

```
LockClusterWith ::=
  'LOCK' 'CLUSTER' 'WITH' lockStrategy ('TIMEOUT' timeoutMillis)?

timeoutmillis ::=
  long
```

补充说明

- 当 CLUSTER 已经处于被锁状态时，无法重复加锁，否则会抛出异常。
- lockStrategy 当前支持两种锁策略，分别是排他锁 WRITE 与读写锁 READ_WRITE。
- timeoutMillis 用于表明尝试加锁的超时时间，其单位为毫秒，未指定时，默认为 3000 毫秒。

示例

- 采用排他锁锁定 CLUSTER，不设置超时时间

```
LOCK CLUSTER WITH WRITE;
```

- 采用读写锁锁定 CLUSTER，并设置超时时间为 2000 毫秒

```
LOCK CLUSTER WITH READ_WRITE TIMEOUT 2000;
```

保留字

LOCK、CLUSTER、WITH

相关链接

- 保留字

UNLOCK CLUSTER

描述

UNLOCK CLUSTER 语法用于解除通过 LOCK CLUSTER WITH 语句施加在 CLUSTER 上的锁。

语法

```
UnlockCluster ::=
  'UNLOCK' 'CLUSTER' ('TIMEOUT' timeoutMillis)?

timeoutmillis ::=
  long
```

补充说明

- 当 CLUSTER 不处于被锁状态时，无法解除锁，否则会抛出异常。
- timeoutMillis 表明尝试解锁的超时时间，其单位为毫秒，未指定时，默认为 3000 毫秒。

示例

- 解锁 CLUSTER，不设置超时时间。

```
UNLOCK CLUSTER;
```

- 解锁 CLUSTER，并设置超时时间为 2000 毫秒。

```
UNLOCK CLUSTER TIMEOUT 2000;
```

保留字

UNLOCK、CLUSTER

相关链接

- 保留字

RUL 语法

RUL (Resource Utility Language) 为 Apache ShardingSphere 的工具类语言，提供 SQL 解析、SQL 格式化、执行计划预览等功能。

PARSE SQL

描述

PARSE SQL 语法用于解析 SQL 并输出抽象语法树。

语法

```
ParseSql ::=
  'PARSE' sqlStatement
```

返回值说明

列	说明
parsed_statement	解析 SQL 语句类型
parsed_statement_detail	解析 SQL 语句细节

示例

- 解析 SQL 并输出抽象语法树

```
PARSE SELECT * FROM t_order;
```

```
mysql> PARSE SELECT * FROM t_order;
+----------------------+-------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------+
| parsed_statement     | parsed_statement_detail
|
+----------------------+------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------+
| SelectStatement      | {"projections":{"startIndex":7,"stopIndex":7,"projections
":[{"startIndex":7,"stopIndex":7}],"distinctRow":false},"from":{"tableName":{
"startIndex":14,"stopIndex":20,"identifier":{"value":"t_order","quoteCharacter":
"NONE"}}},"parameterCount":0,"parameterMarkerSegments":[],"commentSegments":[]} |
+----------------------+------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------+
1 row in set (0.01 sec)
```

保留字

PARSE

相关链接

- 保留字

FORMAT SQL

描述

FORMAT SQL 语法用于解析并输出格式化后的 SQL 语句。

语法

```
FormatSql ::=
  'FORMAT' sqlStatement
```

返回值说明

列	说明
formatted_result	格式化后的 SQL 语句

示例

- 解析并输出格式化后的 SQL 语句

```
FORMAT SELECT * FROM t_order;
```

```
mysql> FORMAT SELECT * FROM t_order;
+-------------------------+
| formatted_result        |
+-------------------------+
| SELECT *
FROM t_order; |
+-------------------------+
1 row in set (0.00 sec)
```

保留字

FORMAT

相关链接

- 保留字

PREVIEW SQL

描述

PREVIEW SQL 语法用于预览 SQL 执行计划。

语法

```
PreviewSql ::=
  'PREVIEW' sqlStatement
```

返回值说明

列	说明
data_source_name	存储单元名称
actual_sql	实际执行 SQL 语句

示例

- 预览 SQL 执行计划

```
PREVIEW SELECT * FROM t_order;
```

```
mysql> PREVIEW SELECT * FROM t_order;
+------------------+-----------------------+
| data_source_name | actual_sql            |
+------------------+-----------------------+
| su_1             | SELECT * FROM t_order |
+------------------+-----------------------+
1 row in set (0.18 sec)
```

保留字

PREVIEW

相关链接

- 保留字

保留字

RDL

基础保留字

CREATE、ALTER、DROP、TABLE、RULE、TYPE、NAME、PROPERTIES、TRUE、FALSE、IF、NOT、EXISTS

储存单元定义

REGISTER、UNREGISTER、STORAGE、UNIT、HOST、PORT、DB、USER、PASSWORD、PROPERTIES、URL、IGNORE、SINGLE、TABLES

规则定义

分片

DEFAULT、SHARDING、BROADCAST、REFERENCE、DATABASE、STRATEGY、RULES、ALGORITHM、DATANODES、DATABASE_STRATEGY、TABLE_STRATEGY、KEY_GENERATE_STRATEGY、RESOURCES、SHARDING_COLUMN、KEY 、GENERATOR、SHARDING_COLUMNS、KEY_GENERATOR、SHARDING_ALGORITHM、COLUMN、AUDIT_STRATEGY、AUDITORS 、ALLOW_HINT_DISABLE

广播表

BROADCAST

单表

SET、DEFAULT、SINGLE、STORAGE、UNIT、RANDOM

读写分离

READWRITE_SPLITTING、WRITE_STORAGE_UNIT、READ_STORAGE_UNITS 、AUTO_AWARE_RESOURCE

数据加密

ENCRYPT、COLUMNS、CIPHER、ENCRYPT_ALGORITHM

数据库发现

DB_DISCOVERY、STORAGE_UNITS、HEARTBEAT

影子压测

SHADOW、DEFAULT、SOURCE、SHADOW

数据脱敏

MASK、COLUMNS

RQL

基础保留字

SHOW、COUNT、DEFAULT、RULE、RULES、TABLE、DATABASE、FROM、UNUSED、USED

储存单元查询

STORAGE、UNIT、UNITS

规则查询

分片

DEFAULT、SHARDING、BROADCAST、REFERENCE、STRATEGY、ALGORITHM、ALGORITHMS、AUDITORS、KEY、GENERATOR、GENERATORS、AUDITOR、AUDITORS、NODES

单表

SINGLE、STORAGE、UNIT

读写分离

READWRITE_SPLITTING

数据加密

ENCRYPT

数据库发现

DB_DISCOVERY、TYPES、HEARTBEATS

影子压测

SHADOW、ALGORITHMS

数据脱敏

MASK

RAL

ALTER、READWRITE_SPLITTING、RULE、RULES、FROM、ENABLE、DISABLE、SHOW、COMPUTE、NODES、NODE 、STATUS、LABEL、RELABEL、WITH、UNLABEL、AUTHORITY、TRANSACTION、SQL_PARSER、DEFAULT、TYPE 、NAME、PROPERTIES、PARSE_TREE_CACHE、INITIAL_CAPACITY、MAXIMUM_SIZE、CONCURRENCY_LEVEL、SQL_STATEMENT_CACHE、TRAFFIC、TRAFFIC_ALGORITHM、LOAD_BALANCER、CREATE 、DATABASE_VALUE、TABLE_VALUE、CLEAR、MIGRATION、READ、WRITE、WORKER_THREAD、BATCH_SIZE、SHARDING_SIZE 、STREAM_CHANNEL、REGISTER、URL、UNREGISTER、UNITS、INTO、LIST、CHECK、BY、STOP、START、ROLLBACK 、COMMIT、INFO、MODE、DIST、VARIABLE、VARIABLES、WHERE、DROPSET、SET、HINT、SOURCE、ADD 、SHARDING、STORAGE、UNIT、USER、PASSWORD、REFRESH、METADATA、TABLE、DATABASE、GOVERNANCE、CENTER 、EXPORT、CONFIGURATION、TO、FILE、IMPORT、USED, IMPLEMENTATIONS, OF, KEY, GENERATE, ALGORITHM 、QUERY、LOAD、BALANCE, FEDERATION, SQL_FEDERATION_ENABLED, ALL_QUERY_USE_SQL_FEDERATION 、EXECUTION_PLAN_CACHE

RUL

`PARSE`、`FORMAT`、`PREVIEW`

补充说明

- 上述保留字大小写不敏感

使用

本章节将结合 DistSQL 的语法，并以实战的形式分别介绍如何使用 DistSQL 管理分布式数据库下的资源和规则。

前置工作

以 MySQL 为例，其他数据库可直接替换。

1. 启动 MySQL 服务；
2. 创建待注册资源的 MySQL 数据库；
3. 在 MySQL 中为 ShardingSphere-Proxy 创建一个拥有创建权限的角色或者用户；
4. 启动 ZooKeeper 服务；
5. 添加 `mode` 和 `authentication` 配置参数到 `global.yaml`；
6. 启动 ShardingSphere-Proxy；
7. 通过应用程序或终端连接到 ShardingSphere-Proxy；

创建数据库

1. 创建逻辑库。

```
CREATE DATABASE foo_db;
```

2. 使用新创建的逻辑库。

```
USE foo_db;
```

资源操作

详见具体规则示例。

规则操作

详见具体规则示例。

注意事项

1. 当前, DROP DATABASE 只会移除 逻辑的分布式数据库，不会删除用户真实的数据库；
2. DROP TABLE 会将逻辑分片表和数据库中真实的表全部删除；
3. CREATE DATABASE 只会创建 逻辑的分布式数据库，所以需要用户提前创建好真实的数据库。

数据分片

存储单元操作

```
REGISTER STORAGE UNIT ds_0 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_1",
    USER="root",
    PASSWORD="root"
),ds_1 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_2",
    USER="root",
    PASSWORD="root"
);
```

规则操作

- 创建分片规则

```
CREATE SHARDING TABLE RULE t_order(
STORAGE_UNITS(ds_0,ds_1),
SHARDING_COLUMN=order_id,
TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="4")),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME="snowflake"))
);
```

- 创建切分表

```
CREATE TABLE `t_order` (
  `order_id` int NOT NULL,
  `user_id` int NOT NULL,
  `status` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

- 删除切分表

```
DROP TABLE t_order;
```

- 删除分片规则

```
DROP SHARDING TABLE RULE t_order;
```

- 移除数据源

```
UNREGISTER STORAGE UNIT ds_0, ds_1;
```

- 删除分布式数据库

```
DROP DATABASE foo_db;
```

读写分离

存储单元操作

```
REGISTER STORAGE UNIT write_ds (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_0",
    USER="root",
    PASSWORD="root"
),read_ds (
    HOST="127.0.0.1",
    PORT=3307,
    DB="ds_0",
    USER="root",
    PASSWORD="root"
);
```

规则操作

- 创建读写分离规则

```
CREATE READWRITE_SPLITTING RULE group_0 (
WRITE_STORAGE_UNIT=write_ds,
READ_STORAGE_UNITS(read_ds),
TYPE(NAME="random")
);
```

- 修改读写分离规则

```
ALTER READWRITE_SPLITTING RULE group_0 (
WRITE_STORAGE_UNIT=write_ds,
READ_STORAGE_UNITS(read_ds),
TYPE(NAME="random",PROPERTIES("read_weight"="2:0"))
);
```

- 删除读写分离规则

```
DROP READWRITE_SPLITTING RULE group_0;
```

- 移除数据源

```
UNREGISTER STORAGE UNIT write_ds,read_ds;
```

- 删除分布式数据库

```
DROP DATABASE readwrite_splitting_db;
```

数据加密

存储单元操作

```
REGISTER STORAGE UNIT ds_0 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_0",
    USER="root",
    PASSWORD="root"
);
```

规则操作

- 创建加密规则

```
CREATE ENCRYPT RULE t_encrypt (
    COLUMNS(
        (NAME=user_id,CIPHER=user_cipher,ENCRYPT_ALGORITHM(TYPE(NAME='AES',
PROPERTIES('aes-key-value'='123456abc', 'digest-algorithm-name'='SHA-1')))),
        (NAME=order_id,CIPHER =order_cipher,ENCRYPT_ALGORITHM(TYPE(NAME='AES',
PROPERTIES('aes-key-value'='123456abc', 'digest-algorithm-name'='SHA-1'))))
));
```

- 创建加密表

```
CREATE TABLE `t_encrypt` (
    `id` int(11) NOT NULL,
    `user_id` varchar(45) DEFAULT NULL,
    `order_id` varchar(45) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

- 修改加密规则

```
ALTER ENCRYPT RULE t_encrypt (
    COLUMNS(
        (NAME=user_id,CIPHER=user_cipher,ENCRYPT_ALGORITHM(TYPE(NAME='AES',
PROPERTIES('aes-key-value'='123456abc', 'digest-algorithm-name'='SHA-1'))))
));
```

- 删除加密规则

```
DROP ENCRYPT RULE t_encrypt;
```

- 移除数据源

```
UNREGISTER STORAGE UNIT ds_0;
```

- 删除分布式数据库

```
DROP DATABASE encrypt_db;
```

数据脱敏

存储单元操作

```
REGISTER STORAGE UNIT ds_0 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_0",
    USER="root",
    PASSWORD="root"
);
```

规则操作

- 创建数据脱敏规则

```
CREATE MASK RULE t_mask (
    COLUMNS(
        (NAME=phone_number,TYPE(NAME='MASK_FROM_X_TO_Y', PROPERTIES("from-x"=1,
"to-y"=2, "replace-char"="*"))),
        (NAME=address,TYPE(NAME='MD5'))
));
```

- 创建数据脱敏表

```
CREATE TABLE `t_mask` (
    `id` int(11) NOT NULL,
    `user_id` varchar(45) DEFAULT NULL,
    `phone_number` varchar(45) DEFAULT NULL,
    `address` varchar(45) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

- 修改数据脱敏规则

```
ALTER MASK RULE t_mask (
    COLUMNS(
        (NAME=user_id,TYPE(NAME='MD5'))
));
```

- 删除数据脱敏规则

```
DROP MASK RULE t_mask;
```

- 移除数据源

```
UNREGISTER STORAGE UNIT ds_0;
```

- 删除分布式数据库

```
DROP DATABASE mask_db;
```

影子库压测

存储单元操作

```
REGISTER STORAGE UNIT ds_0 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_0",
    USER="root",
    PASSWORD="root"
),ds_1 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_1",
    USER="root",
    PASSWORD="root"
),ds_2 (
    HOST="127.0.0.1",
    PORT=3306,
    DB="ds_2",
    USER="root",
    PASSWORD="root"
);
```

规则操作

- 创建影子库压测规则

```
CREATE SHADOW RULE group_0(
SOURCE=ds_0,
SHADOW=ds_1,
t_order(TYPE(NAME="SQL_HINT"),TYPE(NAME="REGEX_MATCH", PROPERTIES("operation"=
"insert","column"="user_id", "regex"='[1]'))),
t_order_item(TYPE(NAME="SQL_HINT")));
```

- 修改影子库压测规则

```
ALTER SHADOW RULE group_0(
SOURCE=ds_0,
```

```
SHADOW=ds_2,
t_order_item(TYPE(NAME="SQL_HINT")));
```

- 删除影子库压测规则

```
DROP SHADOW RULE group_0;
```

- 移除数据源

```
UNREGISTER STORAGE UNIT ds_0,ds_1,ds_2;
```

9. 删除分布式数据库

```
DROP DATABASE foo_db;
```

9.2.4 数据迁移

简介

ShardingSphere 可以提供给用户通用的数据迁移解决方案。

于 **4.1.0** 开始向用户提供。

运行部署

背景信息

对于使用单数据库运行的系统来说，如何安全简单地将数据迁移至水平分片的数据库上，一直以来都是一个迫切的需求。

前提条件

- Proxy 采用纯 JAVA 开发，JDK 建议 1.8 或以上版本。
- 数据迁移使用集群模式，目前支持 ZooKeeper 作为注册中心。

操作步骤

1. 获取 ShardingSphere-Proxy。详情请参见 proxy 启动手册。
2. 修改配置文件 `conf/global.yaml`，详情请参见模式配置。

目前 `mode` 必须是 `Cluster`，需要提前启动对应的注册中心。

配置示例：

```
mode:
  type: Cluster
  repository:
    type: ZooKeeper
    props:
      namespace: governance_ds
      server-lists: localhost:2181
      retryIntervalMilliseconds: 500
      timeToLiveSeconds: 60
      maxRetries: 3
      operationTimeoutMilliseconds: 500
```

3. 引入 JDBC 驱动。

proxy 已包含 PostgreSQL JDBC 和 openGauss JDBC 驱动。

如果后端连接以下数据库，请下载相应 JDBC 驱动 jar 包，并将其放入 `${shardingsphere-proxy}/ext-lib` 目录。

数据库	JDBC 驱动
MySQL	mysql-connector-j-8.3.0.jar

如果是异构迁移，源端支持范围更广的数据库。JDBC 驱动处理方式同上。

4. 启动 ShardingSphere-Proxy：

```
sh bin/start.sh
```

5. 查看 proxy 日志 `logs/stdout.log`，看到日志中出现：

```
[INFO ] [main] o.a.s.p.frontend.ShardingSphereProxy - ShardingSphere-Proxy start
success
```

确认启动成功。

6. 按需配置迁移

6.1. 查询配置。

```
SHOW MIGRATION RULE;
```

默认配置如下：

```
+-------------------------------------------------------------+-------------------
-------------------+-------------------------------------------------------+
| read                                                        | write
| stream_channel                                        |
+-------------------------------------------------------------+-------------------
-------------------+-------------------------------------------------------+
| {"workerThread":20,"batchSize":1000,"shardingSize":10000000} | {"workerThread
```

```
":20,"batchSize":1000} | {"type":"MEMORY","props":{"block-queue-size":"2000"}} |
+-----------------------------------------------------------------+-------------------
-------------------+-------------------------------------------------------+
```

6.2. 修改配置（可选）。

因 migration rule 具有默认值，无需创建，仅提供 ALTER 语句。

完整配置 DistSQL 示例：

```
ALTER MIGRATION RULE (
READ(
  WORKER_THREAD=20,
  BATCH_SIZE=1000,
  SHARDING_SIZE=10000000,
  RATE_LIMITER (TYPE(NAME='QPS',PROPERTIES('qps'='500')))
),
WRITE(
  WORKER_THREAD=20,
  BATCH_SIZE=1000,
  RATE_LIMITER (TYPE(NAME='TPS',PROPERTIES('tps'='2000')))
),
STREAM_CHANNEL (TYPE(NAME='MEMORY',PROPERTIES('block-queue-size'='2000')))
);
```

配置项说明：

```
ALTER MIGRATION RULE (
READ( -- 数据读取配置。如果不配置则部分参数默认生效。
  WORKER_THREAD=20, -- 从源端摄取全量数据的线程池大小。如果不配置则使用默认值。
  BATCH_SIZE=1000, -- 一次查询操作返回的最大记录数。如果不配置则使用默认值。
  SHARDING_SIZE=10000000, -- 全量数据分片大小。如果不配置则使用默认值。
  RATE_LIMITER ( -- 限流算法。如果不配置则不限流。
  TYPE( -- 算法类型。可选项：QPS
  NAME='QPS',
  PROPERTIES( -- 算法属性
  'qps'='500'
  )))
),
WRITE( -- 数据写入配置。如果不配置则部分参数默认生效。
  WORKER_THREAD=20, -- 数据写入到目标端的线程池大小。如果不配置则使用默认值。
  BATCH_SIZE=1000, -- 一次批量写入操作的最大记录数。如果不配置则使用默认值。
  RATE_LIMITER ( -- 限流算法。如果不配置则不限流。
  TYPE( -- 算法类型。可选项：TPS
  NAME='TPS',
  PROPERTIES( -- 算法属性
  'tps'='2000'
  )))
),
STREAM_CHANNEL ( -- 数据通道，连接生产者和消费者，用于 read 和 write 环节。如果不配置则默认使
```

```
用 MEMORY 类型。
TYPE( -- 算法类型。可选项: MEMORY
NAME='MEMORY',
PROPERTIES( -- 算法属性
'block-queue-size'='2000' -- 属性: 阻塞队列大小
)))
);
```

使用手册

MySQL 使用手册

环境要求

支持的 MySQL 版本：5.1.15 ~ 8.0.x。

权限要求

1. 源端开启 `binlog`

MySQL 5.7 `my.cnf` 示例配置：

```
[mysqld]
server-id=1
log-bin=mysql-bin
binlog-format=row
binlog-row-image=full
max_connections=600
```

执行以下命令，确认是否有开启 binlog：

```
show variables like '%log_bin%';
show variables like '%binlog%';
```

如以下显示，则说明 binlog 已开启

```
+-----------------------------------------+---------------------------------------+
| Variable_name                           | Value                                 |
+-----------------------------------------+---------------------------------------+
| log_bin                                 | ON                                    |
| binlog_format                           | ROW                                   |
| binlog_row_image                        | FULL                                  |
+-----------------------------------------+---------------------------------------+
```

2. 赋予源端 MySQL 账号 replication 相关权限。

执行以下命令，查看该用户是否有迁移权限：

```
SHOW GRANTS FOR 'migration_user';
```

示例结果:

```
+-----------------------------------------------------------------------------+
|Grants for ${username}@${host}                                               |
+-----------------------------------------------------------------------------+
|GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO ${username}@${host}    |
|.......                                                                      |
+-----------------------------------------------------------------------------+
```

3. 赋予 MySQL 账号 DDL DML 权限

源端账号需要具备查询权限。示例:

```
GRANT SELECT ON migration_ds_0.* TO `migration_user`@`%`;
```

目标端账号需要具备增删改查等权限。示例:

```
GRANT CREATE, DROP, INDEX, SELECT, INSERT, UPDATE, DELETE ON *.* TO `migration_
user`@`%`;
```

详情请参见 MySQL GRANT

完整流程示例

前提条件

1. 在 MySQL 已准备好源端库、表、数据。

```
DROP DATABASE IF EXISTS migration_ds_0;
CREATE DATABASE migration_ds_0 DEFAULT CHARSET utf8;

USE migration_ds_0;

CREATE TABLE t_order (order_id INT NOT NULL, user_id INT NOT NULL, status
VARCHAR(45) NULL, PRIMARY KEY (order_id));

INSERT INTO t_order (order_id, user_id, status) VALUES (1,2,'ok'),(2,4,'ok'),(3,6,
'ok'),(4,1,'ok'),(5,3,'ok'),(6,5,'ok');
```

2. 在 MySQL 准备目标端库。

```
DROP DATABASE IF EXISTS migration_ds_10;
CREATE DATABASE migration_ds_10 DEFAULT CHARSET utf8;

DROP DATABASE IF EXISTS migration_ds_11;
CREATE DATABASE migration_ds_11 DEFAULT CHARSET utf8;
```

```
DROP DATABASE IF EXISTS migration_ds_12;
CREATE DATABASE migration_ds_12 DEFAULT CHARSET utf8;
```

操作步骤

1. 在 proxy 新建逻辑数据库并配置好存储单元和规则。

```
CREATE DATABASE sharding_db;

USE sharding_db

REGISTER STORAGE UNIT ds_2 (
    URL="jdbc:mysql://127.0.0.1:3306/migration_ds_10?serverTimezone=UTC&
useSSL=false",
    USER="root",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
), ds_3 (
    URL="jdbc:mysql://127.0.0.1:3306/migration_ds_11?serverTimezone=UTC&
useSSL=false",
    USER="root",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
), ds_4 (
    URL="jdbc:mysql://127.0.0.1:3306/migration_ds_12?serverTimezone=UTC&
useSSL=false",
    USER="root",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);

CREATE SHARDING TABLE RULE t_order(
STORAGE_UNITS(ds_2,ds_3,ds_4),
SHARDING_COLUMN=order_id,
TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="6")),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME="snowflake"))
);
```

如果是迁移到异构数据库，那目前需要在 proxy 执行建表语句。

2. 在 proxy 配置源端存储单元。

```
REGISTER MIGRATION SOURCE STORAGE UNIT ds_0 (
    URL="jdbc:mysql://127.0.0.1:3306/migration_ds_0?serverTimezone=UTC&useSSL=false
",
    USER="root",
    PASSWORD="root",
```

```
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);
```

3. 启动数据迁移。

```
MIGRATE TABLE ds_0.t_order INTO t_order;
```

或者指定目标端逻辑库:

```
MIGRATE TABLE ds_0.t_order INTO sharding_db.t_order;
```

4. 查看数据迁移作业列表。

```
SHOW MIGRATION LIST;
```

示例结果:

```
+-------------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
| id                                        | tables       | job_item_count |
active | create_time         | stop_time |
+-------------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
| j0102p00002333dcb3d9db141cef14bed6fbf1ab54 | ds_0.t_order | 1              | true
| 2023-09-20 14:41:32 | NULL      |
+-------------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
```

5. 查看数据迁移详情。

```
SHOW MIGRATION STATUS 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

示例结果:

```
+------+-------------+--------------+-------------------------+--------+----------
---------------+-------------------------------+-------------------------+--------
-------+
| item | data_source | tables       | status                  | active |
processed_records_count | inventory_finished_percentage | incremental_idle_seconds
| error_message |
+------+-------------+--------------+-------------------------+--------+----------
---------------+-------------------------------+-------------------------+--------
-------+
| 0    | ds_0        | ds_0.t_order | EXECUTE_INCREMENTAL_TASK | true   | 6
| 100                           |                          |               |
+------+-------------+--------------+-------------------------+--------+----------
---------------+-------------------------------+-------------------------+--------
-------+
```

6. 执行数据一致性校验。

```
CHECK MIGRATION 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54' BY TYPE (NAME='DATA_
MATCH');
```

数据一致性校验算法类型来自:

```
SHOW MIGRATION CHECK ALGORITHMS;
```

示例结果:

```
+-------------+--------------+-------------------------------------------------
---------+---------------------------+
| type        | type_aliases | supported_database_types
| description                |
+-------------+--------------+-------------------------------------------------
---------+---------------------------+
| CRC32_MATCH |              | MySQL,MariaDB,H2
| Match CRC32 of records.    |
| DATA_MATCH  |              | SQL92,MySQL,PostgreSQL,openGauss,Oracle,SQLServer,
MariaDB,H2 | Match raw data of records. |
+-------------+--------------+-------------------------------------------------
---------+---------------------------+
```

目标端开启数据加密的情况需要使用 DATA_MATCH。

异构迁移需要使用 DATA_MATCH。

查询数据一致性校验进度:

```
SHOW MIGRATION CHECK STATUS 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

示例结果:

```
+--------------+--------+---------------------+--------+--------------------------
----+----------------------------+-------------------------+---------------------
----+-------------------------+------------------+----------------+----------------
-+----------------+
| tables       | result | check_failed_tables | active | inventory_finished_
percentage | inventory_remaining_seconds | incremental_idle_seconds | check_begin_
time        | check_end_time          | duration_seconds | algorithm_type |
algorithm_props | error_message |
+--------------+--------+---------------------+--------+--------------------------
----+----------------------------+-------------------------+---------------------
----+-------------------------+------------------+----------------+----------------
-+----------------+
| ds_0.t_order | true   |                     | false  | 100
| 0                          |                          | 2023-09-20 14:45:31.992
| 2023-09-20 14:45:33.519 | 1                | DATA_MATCH     |                 |
|
+--------------+--------+---------------------+--------+--------------------------
----+----------------------------+-------------------------+---------------------
```

```
----+-------------------------+------------------+----------------+----------------
-+----------------+
```

7. 完成作业。

```
COMMIT MIGRATION 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

更多 DistSQL 请参见 RAL # 数据迁移。

PostgreSQL 使用手册

环境要求

支持的 PostgreSQL 版本：9.4 或以上版本。

权限要求

1. 源端开启 test_decoding。
2. 源端调整 WAL 配置。

`postgresql.conf` 示例配置：

```
wal_level = logical
max_wal_senders = 10
max_replication_slots = 10
wal_sender_timeout = 0
max_connections = 600
```

详情请参见 Write Ahead Log 和 Replication。

3. 赋予源端 PostgreSQL 账号 replication 权限。

`pg_hba.conf` 示例配置：

```
host replication repl_acct 0.0.0.0/0 md5
```

详情请参见 The pg_hba.conf File。

4. 赋予源端 PostgreSQL 账号 DDL DML 权限。

如果使用非超级管理员账号进行迁移，要求该账号在迁移时用到的数据库上，具备 CREATE 和 CONNECT 的权限。

示例：

```
GRANT CREATE, CONNECT ON DATABASE migration_ds_0 TO migration_user;
```

还需要账号对迁移的表和 schema 具备访问权限，以 test schema 下的 t_order 表为例。

```
\c migration_ds_0

GRANT USAGE ON SCHEMA test TO GROUP migration_user;
GRANT SELECT ON TABLE test.t_order TO migration_user;
```

PostgreSQL 有 OWNER 的概念，如果是数据库，SCHEMA，表的 OWNER，则可以省略对应的授权步骤。

详情请参见 PostgreSQL GRANT

完整流程示例

前提条件

1. 在 PostgreSQL 已准备好源端库、表、数据。

```
DROP DATABASE IF EXISTS migration_ds_0;
CREATE DATABASE migration_ds_0;

\c migration_ds_0

CREATE TABLE t_order (order_id INT NOT NULL, user_id INT NOT NULL, status
VARCHAR(45) NULL, PRIMARY KEY (order_id));

INSERT INTO t_order (order_id, user_id, status) VALUES (1,2,'ok'),(2,4,'ok'),(3,6,
'ok'),(4,1,'ok'),(5,3,'ok'),(6,5,'ok');
```

2. 在 PostgreSQL 准备目标端库。

```
DROP DATABASE IF EXISTS migration_ds_10;
CREATE DATABASE migration_ds_10;

DROP DATABASE IF EXISTS migration_ds_11;
CREATE DATABASE migration_ds_11;

DROP DATABASE IF EXISTS migration_ds_12;
CREATE DATABASE migration_ds_12;
```

操作步骤

1. 在 proxy 新建逻辑数据库并配置好存储单元和规则。

```
CREATE DATABASE sharding_db;

\c sharding_db

REGISTER STORAGE UNIT ds_2 (
    URL="jdbc:postgresql://127.0.0.1:5432/migration_ds_10",
```

```
    USER="postgres",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
), ds_3 (
    URL="jdbc:postgresql://127.0.0.1:5432/migration_ds_11",
    USER="postgres",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
), ds_4 (
    URL="jdbc:postgresql://127.0.0.1:5432/migration_ds_12",
    USER="postgres",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);

CREATE SHARDING TABLE RULE t_order(
STORAGE_UNITS(ds_2,ds_3,ds_4),
SHARDING_COLUMN=order_id,
TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="6")),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME="snowflake"))
);
```

如果是迁移到异构数据库，那目前需要在 proxy 执行建表语句。

2. 在 proxy 配置源端存储单元。

```
REGISTER MIGRATION SOURCE STORAGE UNIT ds_0 (
    URL="jdbc:postgresql://127.0.0.1:5432/migration_ds_0",
    USER="postgres",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);
```

3. 启动数据迁移。

```
MIGRATE TABLE ds_0.t_order INTO t_order;
```

或者指定目标端逻辑库：

```
MIGRATE TABLE ds_0.t_order INTO sharding_db.t_order;
```

也可以指定源端 schema：

```
MIGRATE TABLE ds_0.public.t_order INTO sharding_db.t_order;
```

4. 查看数据迁移作业列表。

```
SHOW MIGRATION LIST;
```

示例结果：

```
+----------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
| id                                     | tables       | job_item_count |
active | create_time         | stop_time |
+----------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
| j0102p00002333dcb3d9db141cef14bed6fbf1ab54 | ds_0.t_order | 1              | true
| 2023-09-20 14:41:32 | NULL      |
+----------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
```

5. 查看数据迁移详情。

```
SHOW MIGRATION STATUS 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

示例结果:

```
+------+-------------+--------------+-------------------------+--------+----------
---------------+------------------------------+-------------------------+--------
-------+
| item | data_source | tables       | status                  | active |
processed_records_count | inventory_finished_percentage | incremental_idle_seconds
| error_message |
+------+-------------+--------------+-------------------------+--------+----------
---------------+------------------------------+-------------------------+--------
-------+
| 0    | ds_0        | ds_0.t_order | EXECUTE_INCREMENTAL_TASK | true   | 6
| 100                          |                         |               |
+------+-------------+--------------+-------------------------+--------+----------
---------------+------------------------------+-------------------------+--------
-------+
```

6. 执行数据一致性校验。

```
CHECK MIGRATION 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

查询数据一致性校验进度:

```
SHOW MIGRATION CHECK STATUS 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

示例结果:

```
+--------------+--------+---------------------+--------+-------------------------
----+----------------------------+-------------------------+---------------------
----+-------------------------+------------------+----------------+----------------
-+----------------+
| tables       | result | check_failed_tables | active | inventory_finished_
percentage | inventory_remaining_seconds | incremental_idle_seconds | check_begin_
time        | check_end_time          | duration_seconds | algorithm_type |
```

```
algorithm_props | error_message |
+--------------+--------+--------------------+--------+-------------------------
----+----------------------------+-------------------------+--------------------
----+-------------------------+------------------+----------------+----------------
-+----------------+
| ds_0.t_order | true   |                    | false  | 100
| 0                          |                         | 2023-09-20 14:45:31.992
| 2023-09-20 14:45:33.519 | 1                | DATA_MATCH     |                 |
|
+--------------+--------+--------------------+--------+-------------------------
----+----------------------------+-------------------------+--------------------
----+-------------------------+------------------+----------------+----------------
-+----------------+
```

7. 完成作业。

```
COMMIT MIGRATION 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

更多 DistSQL 请参见 RAL # 数据迁移。

openGauss 使用手册

环境要求

支持的 openGauss 版本：2.0.1 ~ 3.0.0。

权限要求

1. 调整源端 WAL 配置。

`postgresql.conf` 示例配置：

```
wal_level = logical
max_wal_senders = 10
max_replication_slots = 10
wal_sender_timeout = 0
max_connections = 600
```

详情请参见 Write Ahead Log 和 Replication。

2. 赋予源端 openGauss 账号 replication 权限。

`pg_hba.conf` 示例配置：

```
host replication repl_acct 0.0.0.0/0 md5
```

详情请参见 Configuring Client Access Authentication 和 Example: Logic Replication Code。

3. 赋予 openGauss 账号 DDL DML 权限。

如果使用非超级管理员账号进行迁移，要求该账号在迁移时用到的数据库上，具备 CREATE 和 CONNECT 的权限。

示例：

```
GRANT CREATE, CONNECT ON DATABASE migration_ds_0 TO migration_user;
```

还需要账号对迁移的表和 schema 具备访问权限，以 test schema 下的 t_order 表为例。

```
\c migration_ds_0

GRANT USAGE ON SCHEMA test TO GROUP migration_user;
GRANT SELECT ON TABLE test.t_order TO migration_user;
```

openGauss 有 OWNER 的概念，如果是数据库，SCHEMA，表的 OWNER，则可以省略对应的授权步骤。

openGauss 不允许普通账户在 public schema 下操作。所以如果迁移的表在 public schema 下，需要额外授权。

```
GRANT ALL PRIVILEGES TO migration_user;
```

详情请参见 openGauss GRANT

完整流程示例

前提条件

1. 准备好源端库、表、数据。

1.1. 同构数据库。

```
DROP DATABASE IF EXISTS migration_ds_0;
CREATE DATABASE migration_ds_0;

\c migration_ds_0

CREATE TABLE t_order (order_id INT NOT NULL, user_id INT NOT NULL, status
VARCHAR(45) NULL, PRIMARY KEY (order_id));

INSERT INTO t_order (order_id, user_id, status) VALUES (1,2,'ok'),(2,4,'ok'),(3,6,
'ok'),(4,1,'ok'),(5,3,'ok'),(6,5,'ok');
```

1.2. 异构数据库。

MySQL 示例：

```
DROP DATABASE IF EXISTS migration_ds_0;
CREATE DATABASE migration_ds_0 DEFAULT CHARSET utf8;

USE migration_ds_0;
```

```
CREATE TABLE t_order (order_id INT NOT NULL, user_id INT NOT NULL, status
VARCHAR(45) NULL, PRIMARY KEY (order_id));

INSERT INTO t_order (order_id, user_id, status) VALUES (1,2,'ok'),(2,4,'ok'),(3,6,
'ok'),(4,1,'ok'),(5,3,'ok'),(6,5,'ok');
```

2. 在 openGauss 准备目标端库。

```
DROP DATABASE IF EXISTS migration_ds_10;
CREATE DATABASE migration_ds_10;

DROP DATABASE IF EXISTS migration_ds_11;
CREATE DATABASE migration_ds_11;

DROP DATABASE IF EXISTS migration_ds_12;
CREATE DATABASE migration_ds_12;
```

操作步骤

1. 在 proxy 新建逻辑数据库并配置好存储单元和规则。

1.1. 创建逻辑库。

```
CREATE DATABASE sharding_db;

\c sharding_db
```

1.2. 注册存储单元。

```
REGISTER STORAGE UNIT ds_2 (
    URL="jdbc:opengauss://127.0.0.1:5432/migration_ds_10",
    USER="gaussdb",
    PASSWORD="Root@123",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
), ds_3 (
    URL="jdbc:opengauss://127.0.0.1:5432/migration_ds_11",
    USER="gaussdb",
    PASSWORD="Root@123",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
), ds_4 (
    URL="jdbc:opengauss://127.0.0.1:5432/migration_ds_12",
    USER="gaussdb",
    PASSWORD="Root@123",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);
```

1.3. 创建分片规则。

```
CREATE SHARDING TABLE RULE t_order(
STORAGE_UNITS(ds_2,ds_3,ds_4),
SHARDING_COLUMN=order_id,
TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="6")),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME="snowflake"))
);
```

1.4. 创建目标端表。

如果是迁移到异构数据库，那目前需要在 proxy 执行建表语句。

```
CREATE TABLE t_order (order_id INT NOT NULL, user_id INT NOT NULL, status
VARCHAR(45) NULL, PRIMARY KEY (order_id));
```

2. 在 proxy 配置源端存储单元。

2.1. 同构数据库。

```
REGISTER MIGRATION SOURCE STORAGE UNIT ds_0 (
    URL="jdbc:opengauss://127.0.0.1:5432/migration_ds_0",
    USER="gaussdb",
    PASSWORD="Root@123",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);
```

2.2. 异构数据库。

MySQL 示例：

```
REGISTER MIGRATION SOURCE STORAGE UNIT ds_0 (
    URL="jdbc:mysql://127.0.0.1:3306/migration_ds_0?serverTimezone=UTC&useSSL=false
",
    USER="root",
    PASSWORD="root",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);
```

3. 启动数据迁移。

```
MIGRATE TABLE ds_0.t_order INTO t_order;
```

或者指定目标端逻辑库：

```
MIGRATE TABLE ds_0.t_order INTO sharding_db.t_order;
```

也可以指定源端 schema：

```
MIGRATE TABLE ds_0.public.t_order INTO sharding_db.t_order;
```

4. 查看数据迁移作业列表。

```
SHOW MIGRATION LIST;
```

示例结果:

```
+------------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
| id                                       | tables       | job_item_count |
active | create_time         | stop_time |
+------------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
| j0102p00002333dcb3d9db141cef14bed6fbf1ab54 | ds_0.t_order | 1              | true
| 2023-09-20 14:41:32 | NULL      |
+------------------------------------------+--------------+----------------+-----
---+---------------------+-----------+
```

5. 查看数据迁移详情。

```
SHOW MIGRATION STATUS 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

示例结果:

```
+------+-------------+--------------+-------------------------+--------+----------
---------------+------------------------------+-------------------------+--------
-------+
| item | data_source | tables       | status                  | active |
processed_records_count | inventory_finished_percentage | incremental_idle_seconds
| error_message |
+------+-------------+--------------+-------------------------+--------+----------
---------------+------------------------------+-------------------------+--------
-------+
| 0    | ds_0        | ds_0.t_order | EXECUTE_INCREMENTAL_TASK | true   | 6
| 100                          |                         |               |
+------+-------------+--------------+-------------------------+--------+----------
---------------+------------------------------+-------------------------+--------
-------+
```

6. 执行数据一致性校验。

```
CHECK MIGRATION 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

查询数据一致性校验进度:

```
SHOW MIGRATION CHECK STATUS 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

示例结果:

```
+--------------+--------+---------------------+--------+-------------------------
----+-----------------------------+-------------------------+---------------------
----+-------------------------+------------------+----------------+----------------
```

```
-+---------------+
| tables       | result | check_failed_tables | active | inventory_finished_
percentage | inventory_remaining_seconds | incremental_idle_seconds | check_begin_
time        | check_end_time          | duration_seconds | algorithm_type |
algorithm_props | error_message |
+--------------+--------+---------------------+--------+--------------------------
----+----------------------------+-------------------------+---------------------
----+-------------------------+------------------+----------------+----------------
-+---------------+
| ds_0.t_order | true   |                     | false  | 100
| 0                          |                          | 2023-09-20 14:45:31.992
| 2023-09-20 14:45:33.519 | 1                | DATA_MATCH     |                 |
|
+--------------+--------+---------------------+--------+--------------------------
----+----------------------------+-------------------------+---------------------
----+-------------------------+------------------+----------------+----------------
-+---------------+
```

7. 完成作业。

```
COMMIT MIGRATION 'j0102p00002333dcb3d9db141cef14bed6fbf1ab54';
```

更多 DistSQL 请参见 RAL # 数据迁移。

9.2.5 可观察性

Agent

源码编译

从 Github 下载 Apache ShardingSphere 源码，对源码进行编译，操作命令如下。

```
git clone --depth 1 https://github.com/apache/shardingsphere.git
cd shardingsphere
mvn clean install -DskipITs -DskipTests -Prelease,default-dep
```

Agent 制品 `distribution/agent/target/apache-shardingsphere-${latest.release.version}-shardingsphere-agent-bin.tar.gz`

Proxy 制品 `distribution/proxy/target/apache-shardingsphere-${latest.release.version}-shardingsphere-proxy-bin.tar.gz`

目录说明

创建 agent 目录，解压 agent 二进制包到 agent 目录。

```
mkdir agent
tar -zxvf apache-shardingsphere-${latest.release.version}-shardingsphere-agent-bin.
tar.gz -C agent
cd agent
tree
├── LICENSE
├── NOTICE
├── conf
│   └── agent.yaml
├── plugins
│   ├── lib
│   │   ├── shardingsphere-agent-metrics-core-${latest.release.version}.jar
│   │   └── shardingsphere-agent-plugin-core-${latest.release.version}.jar
│   ├── logging
│   │   └── shardingsphere-agent-logging-file-${latest.release.version}.jar
│   ├── metrics
│   │   └── shardingsphere-agent-metrics-prometheus-${latest.release.version}.jar
│   └── tracing
│       ├── shardingsphere-agent-tracing-opentelemetry-${latest.release.version}.
jar
└── shardingsphere-agent-${latest.release.version}.jar
```

配置说明

`conf/agent.yaml` 用于管理 agent 配置。内置插件包括 File、Prometheus、OpenTelemetry。

```
plugins:
#  logging:
#    File:
#      props:
#        level: "INFO"
#  metrics:
#    Prometheus:
#      host:  "localhost"
#      port: 9090
#      props:
#        jvm-information-collector-enabled: "true"
#  tracing:
#    OpenTelemetry:
#      props:
#        otel.service.name: "shardingsphere"
#        otel.traces.exporter: "jaeger"
#        otel.exporter.otlp.traces.endpoint: "http://localhost:14250"
#        otel.traces.sampler: "always_on"
```

插件说明

File

目前 File 插件只有构建元数据耗时日志输出，暂无其他日志输出。

Prometheus

用于暴露监控指标

- 参数说明

名称	说明
host	主机
port	端口
jvm-information-collector-enabled	是否采集 JVM 指标信息

OpenTelemetry

OpenTelemetry 可以导出 tracing 数据到 Jaeger，Zipkin。

- 参数说明

名称	说明
otel.service.name	服务名称
otel.traces.exporter	traces exporter
otel.exporter.otlp.traces.endpoint	traces endpoint
otel.traces.sampler	traces sampler

参数参考 OpenTelemetry SDK Autoconfigure

使用方式

启动 ShardingSphere-Proxy

```
tar -zxvf apache-shardingsphere-${latest.release.version}-shardingsphere-proxy-bin.
tar.gz
cd apache-shardingsphere-${latest.release.version}-shardingsphere-proxy-bin
./bin/start.sh -g
```

正常启动后，可以在 ShardingSphere-Proxy 日志中找到 plugin 的加载信息，访问 Proxy 后，可以通过配置的监控地址查看到 `Metric` 和 `Tracing` 的数据。

Metrics

指标名称	指标类型	指标描述
build_info	GAUGE	构建信息
parsed_sql_total	COUNTER	按类型（INSERT、UPDATE、DELETE、SELECT、DDL、DCL、DAL、TCL、RQL、RDL、RAL、RUL）分类的解析总数
routed_sql_total	COUNTER	按类型（INSERT、UPDATE、DELETE、SELECT）分类的路由总数
routed_result_total	COUNTER	路由结果总数 (数据源路由结果、表路由结果)
proxy_state	GAUGE	ShardingSphere-Proxy 状态信息。0 表示正常状态；1 表示熔断状态；2 锁定状态
proxy_meta_data_info	GAUGE	ShardingSphere-Proxy 元数据信息,database_count:逻辑库数量,storage_unit_count：存储节点数量
proxy_current_connections	GAUGE	ShardingSphere-Proxy 的当前连接数
proxy_requests_total	COUNTER	ShardingSphere-Proxy 的接受请求总数
proxy_transactions_total	COUNTER	ShardingSphere-Proxy 的事务总数，按 commit，rollback 分类
proxy_execute_latency_millis	HISTOGRAM	ShardingSphere-Proxy 的执行耗时毫秒直方图
proxy_execute_errors_total	COUNTER	ShardingSphere-Proxy 的执行异常总数

9.2.6 可选插件

ShardingSphere 默认情况下仅包含核心 SPI 的实现，在 Git Source 存在一部分包含第三方依赖的 SPI 实现的插件未包含在内。可在 https://central.sonatype.com/ 进行检索。

所有插件对应的 SPI 和 SPI 的已有实现类均可在 https://shardingsphere.apache.org/document/current/cn/dev-manual/ 检索。

下以 `groupId:artifactId` 的表现形式列出 ShardingSphere-Proxy 所有的内置插件。

- `org.apache.shardingsphere:shardingsphere-cluster-mode-repository-etcd`,集群模式配置信息持久化定义的 etcd 实现
- `org.apache.shardingsphere:shardingsphere-cluster-mode-repository-zookeeper`,集群模式配置信息持久化定义的 zookeeper 实现
- `org.apache.shardingsphere:shardingsphere-jdbc`，JDBC 模块
- `org.apache.shardingsphere:shardingsphere-db-protocol-core`，数据库协议核心

- `org.apache.shardingsphere:shardingsphere-mysql-protocol`,数据库协议的 MySQL 实现
- `org.apache.shardingsphere:shardingsphere-postgresql-protocol`，数据库协议的 PostgreSQL 实现
- `org.apache.shardingsphere:shardingsphere-opengauss-protocol`，数据库协议的 OpenGauss 实现
- `org.apache.shardingsphere:shardingsphere-proxy-frontend-core`，用于 ShardingSphere-Proxy 解析与适配访问数据库的协议
- `org.apache.shardingsphere:shardingsphere-proxy-frontend-mysql`，用于 ShardingSphere-Proxy 解析与适配访问数据库的协议的 MySQL 实现
- `org.apache.shardingsphere:shardingsphere-proxy-frontend-postgresql`，用于 ShardingSphere-Proxy 解析与适配访问数据库的协议的 PostgreSQL 实现
- `org.apache.shardingsphere:shardingsphere-proxy-frontend-opengauss`，用于 ShardingSphere-Proxy 解析与适配访问数据库的协议的 openGauss 实现
- `org.apache.shardingsphere:shardingsphere-proxy-backend-core`，ShardingSphere Proxy 的后端核心模块
- `org.apache.shardingsphere:shardingsphere-standalone-mode-core`，单机模式配置信息持久化定义核心

对于核心的 `org.apache.shardingsphere:shardingsphere-jdbc`，其内置插件参考ShardingSphere-JDBC 可选插件。

如果 ShardingSphere-Proxy 需要使用可选插件，需要在 Maven Central 下载包含其 SPI 实现的 JAR 和其依赖的 JAR。

下以 `groupId:artifactId` 的表现形式列出所有的可选插件。

- 单机模式配置信息持久化定义
  - `org.apache.shardingsphere:shardingsphere-standalone-mode-repository-jdbc`，基于 JDBC 的持久化
- XA 分布式事务管理器
  - `org.apache.shardingsphere:shardingsphere-transaction-xa-narayana`，基于 Narayana 的 XA 分布式事务管理器
- 行表达式
  - `org.apache.shardingsphere:shardingsphere-infra-expr-espresso`，基于 GraalVM Truffle 的 Espresso 实现的使用 Groovy 语法的行表达式
- 数据库类型识别
  - `org.apache.shardingsphere:shardingsphere-infra-database-hive`，对 Hive 的 jdbcURL 的识别适配，元数据加载实现
  - `org.apache.shardingsphere:shardingsphere-infra-database-presto`，对 Presto 的 jdbcURL 的识别适配，元数据加载实现

- SQL 解析
  - `org.apache.shardingsphere:shardingsphere-parser-sql-clickhouse`,SQL 解析的 ClickHouse 方言实现
  - `org.apache.shardingsphere:shardingsphere-parser-sql-hive`，SQL 解析的 Hive 方言实现

除了以上可选插件外，ShardingSphere 社区开发者还贡献了大量的插件实现，可以在 ShardingSphere Plugin 仓库中查看插件的使用说明，ShardingSphere Plugin 仓库中的插件会和 ShardingSphere 保持相同的发布节奏，可以手动打包安装到 ShardingSphere 中。

Seata AT 模式事务

背景信息

ShardingSphere Proxy 或 GraalVM Native Image 形态的 ShardingSphere Proxy Native 默认情况下不提供对 Seata 的 AT 模式的支持。两者对 Seata 的 AT 模式的支持均位于可选模块中。

本节依然受到 ShardingSphere JDBC 一侧的 Seata 事务 的已记录内容的限制，但有所不同，

1. 若用户采用混合部署架构使用 ShardingSphere JDBC，此情景不直接与 ShardingSphere Proxy 交互，因此与本文无关。本文仅讨论业务项目不使用 ShardingSphere JDBC 的场景
2. Seata Client 只存在于 ShardingSphere Proxy 中，业务项目并不需要依赖 Seata Client
3. 业务项目的 R2DBC DataSource 可以正常连接至开启 Seata 集成的 ShardingSphere Proxy
4. 对于开启 Seata 集成的 ShardingSphere Proxy，无法通过建立 跨服务的事务传播的操作，传播事务到其他使用 Seata 集成的 ShardingSphere Proxy 实例或其他使用 Seata 集成的微服务。用户如果有这种需求，应考虑为 ShardingSphere 提交 PR
5. ShardingSphere JDBC 对 Seata 的 TCC 模式建立的假设，在 ShardingSphere Proxy 上失效

下文以使用 Seata Client 2.3.0 的 ShardingSphere Proxy 为例讨论。

操作步骤

1. 确认 Seata Client 的 JAR 和依赖列表
2. 启动 Seata Server
3. 为所涉及的真实数据库创建 `undo_log` 表
4. 创建包含 Seata Client 和 Seata 集成模块的 ShardingSphere Proxy
5. 为 ShardingSphere Proxy 添加 Seata 配置

配置示例

确认 Seata Client 的 JAR 和依赖列表

对于已安装 SDKMAN! 的 Ubuntu 22.04.4，可以以如下命令确认 Seata Client 的所有 compile scope 的依赖，

```
sdk install java 23-open
sdk use java 23-open
sdk install maven 3.9.9
sdk use maven 3.9.9
mvn dependency:get -Dartifact=org.apache.seata:seata-all:2.3.0
mvn -f ~/.m2/repository/org/apache/seata/seata-all/2.3.0/seata-all-2.3.0.pom
dependency:tree | grep -v ':provided' | grep -v ':runtime'
```

与 org.apache.shardingsphere:shardingsphere-proxy-distribution 的 pom.xml 对比，不难发现有差异列表为，

```
org.apache.seata:seata-all:jar:2.3.0
org.springframework:spring-context:jar:5.3.39
org.springframework:spring-expression:jar:5.3.39
org.springframework:spring-core:jar:5.3.39
org.springframework:spring-jcl:jar:5.3.39
org.springframework:spring-beans:jar:5.3.39
org.springframework:spring-aop:jar:5.3.39
org.springframework:spring-webmvc:jar:5.3.39
org.springframework:spring-web:jar:5.3.39
org.springframework:spring-tx:jar:5.3.39
io.netty:netty-all:jar:4.1.101.Final
io.netty:netty-codec-dns:jar:4.1.101.Final
io.netty:netty-codec-haproxy:jar:4.1.101.Final
io.netty:netty-codec-memcache:jar:4.1.101.Final
io.netty:netty-codec-mqtt:jar:4.1.101.Final
io.netty:netty-codec-redis:jar:4.1.101.Final
io.netty:netty-codec-smtp:jar:4.1.101.Final
io.netty:netty-codec-stomp:jar:4.1.101.Final
io.netty:netty-codec-xml:jar:4.1.101.Final
io.netty:netty-handler-ssl-ocsp:jar:4.1.101.Final
io.netty:netty-resolver-dns:jar:4.1.101.Final
io.netty:netty-transport-rxtx:jar:4.1.101.Final
io.netty:netty-transport-sctp:jar:4.1.101.Final
io.netty:netty-transport-udt:jar:4.1.101.Final
io.netty:netty-transport-classes-kqueue:jar:4.1.101.Final
io.netty:netty-resolver-dns-classes-macos:jar:4.1.101.Final
org.antlr:antlr4:jar:4.8
org.antlr:antlr-runtime:jar:3.5.2
org.antlr:ST4:jar:4.3
org.abego.treelayout:org.abego.treelayout.core:jar:1.0.3
```

```
org.glassfish:javax.json:jar:1.0.4
com.ibm.icu:icu4j:jar:61.1
com.alibaba:fastjson:jar:1.2.83
com.alibaba:druid:jar:1.2.20
com.typesafe:config:jar:1.2.1
commons-pool:commons-pool:jar:1.6
org.apache.dubbo.extensions:dubbo-filter-seata:jar:1.0.2
aopalliance:aopalliance:jar:1.0
```

显然，用户始终应该避免关注过时的 `org.antlr:antlr4-runtime:4.8`。此列表将用于重新创建 ShardingSphere Proxy 的 Docker Image。

此外，用户始终可以通过修改 ShardingSphere 的源代码来将 Seata Client 添加进本地构建的 ShardingSphere Proxy 的 Docker Image 中。

启动 Seata Server，Postgres Server 和 ShardingSphere Proxy

编写 Docker Compose 文件来启动 Seata Server 和 Postgres Server。

```
services:
   postgres:
      image: postgres:17.2-bookworm
      environment:
         POSTGRES_PASSWORD: example
      volumes:
         - ./docker-entrypoint-initdb.d:/docker-entrypoint-initdb.d
   apache-seata-server:
      image: apache/seata-server:2.3.0
      healthcheck:
         test: [ "CMD", "sh", "-c", "curl -s apache-seata-server:7091/health | grep
-q '^ok$'" ]
   shardingsphere-proxy-custom:
      image: example/shardingsphere-proxy-custom:latest
      pull_policy: build
      build:
         context: .
         dockerfile_inline: |
            FROM apache/shardingsphere-proxy:latest
            RUN wget https://repo1.maven.org/maven2/org/apache/shardingsphere/
shardingsphere-transaction-base-seata-at/5.5.2/shardingsphere-transaction-base-
seata-at-5.5.2.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/apache/seata/seata-all/2.3.
0/seata-all-2.3.0.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-
context/5.3.39/spring-context-5.3.39.jar --directory-prefix=/opt/shardingsphere-
proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-
expression/5.3.39/spring-expression-5.3.39.jar --directory-prefix=/opt/
```

```
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-
core/5.3.39/spring-core-5.3.39.jar --directory-prefix=/opt/shardingsphere-proxy/
ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-jcl/
5.3.39/spring-jcl-5.3.39.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-
beans/5.3.39/spring-beans-5.3.39.jar --directory-prefix=/opt/shardingsphere-proxy/
ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-aop/
5.3.39/spring-aop-5.3.39.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-
webmvc/5.3.39/spring-webmvc-5.3.39.jar --directory-prefix=/opt/shardingsphere-
proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-web/
5.3.39/spring-web-5.3.39.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/springframework/spring-tx/
5.3.39/spring-tx-5.3.39.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-all/4.1.101.
Final/netty-all-4.1.101.Final.jar --directory-prefix=/opt/shardingsphere-proxy/ext-
lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-dns/4.1.
101.Final/netty-codec-dns-4.1.101.Final.jar --directory-prefix=/opt/shardingsphere-
proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-haproxy/4.
1.101.Final/netty-codec-haproxy-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-memcache/
4.1.101.Final/netty-codec-memcache-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-mqtt/4.1.
101.Final/netty-codec-mqtt-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-redis/4.1.
101.Final/netty-codec-redis-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-smtp/4.1.
101.Final/netty-codec-smtp-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-stomp/4.1.
101.Final/netty-codec-stomp-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-codec-xml/4.1.
101.Final/netty-codec-xml-4.1.101.Final.jar --directory-prefix=/opt/shardingsphere-
proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-handler-ssl-
ocsp/4.1.101.Final/netty-handler-ssl-ocsp-4.1.101.Final.jar --directory-prefix=/
opt/shardingsphere-proxy/ext-lib
```

```
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-resolver-dns/4.
1.101.Final/netty-resolver-dns-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-transport-rxtx/
4.1.101.Final/netty-transport-rxtx-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-transport-sctp/
4.1.101.Final/netty-transport-sctp-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-transport-udt/4.
1.101.Final/netty-transport-udt-4.1.101.Final.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-transport-
classes-kqueue/4.1.101.Final/netty-transport-classes-kqueue-4.1.101.Final.jar --
directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/io/netty/netty-resolver-dns-
classes-macos/4.1.101.Final/netty-resolver-dns-classes-macos-4.1.101.Final.jar --
directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/antlr/antlr4/4.8/antlr4-4.
8.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/antlr/antlr-runtime/3.5.2/
antlr-runtime-3.5.2.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/antlr/ST4/4.3/ST4-4.3.jar -
-directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/abego/treelayout/org.abego.
treelayout.core/1.0.3/org.abego.treelayout.core-1.0.3.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/glassfish/javax.json/1.0.4/
javax.json-1.0.4.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/com/ibm/icu/icu4j/61.1/icu4j-
61.1.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/com/alibaba/fastjson/1.2.83/
fastjson-1.2.83.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/com/alibaba/druid/1.2.20/druid-
1.2.20.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo.akka.io/maven/com/typesafe/config/1.2.1/config-1.
2.1.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/commons-pool/commons-pool/1.6/
commons-pool-1.6.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/org/apache/dubbo/extensions/
dubbo-filter-seata/1.0.2/dubbo-filter-seata-1.0.2.jar --directory-prefix=/opt/
shardingsphere-proxy/ext-lib
            RUN wget https://repo1.maven.org/maven2/aopalliance/aopalliance/1.0/
aopalliance-1.0.jar --directory-prefix=/opt/shardingsphere-proxy/ext-lib
      volumes:
         - ./conf:/opt/shardingsphere-proxy/conf
      ports:
         - "3308:3308"
```

```
      environment:
         PORT: 3308
      depends_on:
         apache-seata-server:
            condition: service_healthy
```

此外，用户始终可以提前利用 Dockerfile 构建 ShardingSphere Proxy 的 Docker Image，而不是在 Docker Compose 中动态构建 Docker Image。

`./conf` 文件夹包含文件 `global.yaml`，内容如下，

```
authority:
  users:
    - user: root@%
      password: root
      admin: true
  privilege:
    type: ALL_PERMITTED
transaction:
  defaultType: BASE
  providerType: Seata
props:
  proxy-frontend-database-protocol-type: PostgreSQL
```

`./conf` 文件夹包含文件 `file.conf`，内容如下，

```
service {
   vgroupMapping.default_tx_group = "default"
   default.grouplist = "apache-seata-server:8091"
}
```

`./conf` 文件夹包含文件 `registry.conf`，内容如下，

```
registry {
  type = "file"
  file {
    name = "file.conf"
  }
}
config {
  type = "file"
  file {
    name = "file.conf"
  }
}
```

`./conf` 文件夹包含文件 `seata.conf`，内容如下，

```
client {
    application.id = test
    transaction.service.group = default_tx_group
}
```

./docker-entrypoint-initdb.d 文件夹包含文件 init.sh，内容如下，

```
#!/bin/bash
set -e

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" <<-
EOSQL
  CREATE DATABASE demo_ds_0;
  CREATE DATABASE demo_ds_1;
  CREATE DATABASE demo_ds_2;
EOSQL

for i in "demo_ds_0" "demo_ds_1" "demo_ds_2"
do
psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$i" <<-EOSQL
  CREATE TABLE IF NOT EXISTS public.undo_log
  (
      id            SERIAL       NOT NULL,
      branch_id     BIGINT       NOT NULL,
      xid           VARCHAR(128) NOT NULL,
      context       VARCHAR(128) NOT NULL,
      rollback_info BYTEA        NOT NULL,
      log_status    INT          NOT NULL,
      log_created   TIMESTAMP(0) NOT NULL,
      log_modified  TIMESTAMP(0) NOT NULL,
      CONSTRAINT pk_undo_log PRIMARY KEY (id),
      CONSTRAINT ux_undo_log UNIQUE (xid, branch_id)
      );
  CREATE INDEX ix_log_created ON undo_log(log_created);
  COMMENT ON TABLE public.undo_log IS 'AT transaction mode undo table';
  COMMENT ON COLUMN public.undo_log.branch_id IS 'branch transaction id';
  COMMENT ON COLUMN public.undo_log.xid IS 'global transaction id';
  COMMENT ON COLUMN public.undo_log.context IS 'undo_log context,such as
serialization';
  COMMENT ON COLUMN public.undo_log.rollback_info IS 'rollback info';
  COMMENT ON COLUMN public.undo_log.log_status IS '0:normal status,1:defense status
';
  COMMENT ON COLUMN public.undo_log.log_created IS 'create datetime';
  COMMENT ON COLUMN public.undo_log.log_modified IS 'modify datetime';
  CREATE SEQUENCE IF NOT EXISTS undo_log_id_seq INCREMENT BY 1 MINVALUE 1 ;

  CREATE TABLE IF NOT EXISTS t_order (
      order_id BIGSERIAL NOT NULL PRIMARY KEY,
```

```
      order_type INTEGER,
      user_id INTEGER NOT NULL,
      address_id BIGINT NOT NULL,
      status VARCHAR(50)
  );
EOSQL
done
```

创建 ShardingSphere 虚拟库

通过第三方工具在 ShardingSphere Proxy 内创建 ShardingSphere 虚拟库。以 DBeaver Community 为例，若使用 Ubuntu 22.04.4，可通过 Snapcraft 快速安装，

```
sudo apt update && sudo apt upgrade -y
sudo snap install dbeaver-ce
snap run dbeaver-ce
```

在 DBeaver Community 内，使用 `jdbc:postgresql://127.0.0.1:3308/postgres` 的 `jdbcUrl` 连接至 ShardingSphere Proxy，username 和 password 均为 `root`。所需的 JDBC Driver 与 ShardingSphere Proxy 设置的 `proxy-frontend-database-protocol-type` 对应。执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
CREATE DATABASE sharding_db;
```

在 DBeaver Community 内，使用 `jdbc:postgresql://127.0.0.1:3308/sharding_db` 的 `jdbcUrl` 连接至 ShardingSphere Proxy，username 和 password 均为 `root`。执行如下 SQL，

```
-- noinspection SqlNoDataSourceInspectionForFile
REGISTER STORAGE UNIT ds_0 (
  URL="jdbc:postgresql://postgres:5432/demo_ds_0",
  USER="postgres",
  PASSWORD="example"
),ds_1 (
  URL="jdbc:postgresql://postgres:5432/demo_ds_1",
  USER="postgres",
  PASSWORD="example"
),ds_2 (
  URL="jdbc:postgresql://postgres:5432/demo_ds_2",
  USER="postgres",
  PASSWORD="example"
);

CREATE DEFAULT SHARDING DATABASE STRATEGY (
  TYPE="standard",
  SHARDING_COLUMN=user_id,
  SHARDING_ALGORITHM(
    TYPE(
```

```
      NAME=INLINE,
      PROPERTIES(
        "algorithm-expression"="ds_${user_id % 2}"
      )
    )
  )
);

CREATE SHARDING TABLE RULE t_order (
  DATANODES("ds_$->{0..2}.t_order"),
  KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME="SNOWFLAKE"))
);
```

在业务项目引入 Postgres JDBC Driver

在业务项目引入 Postgres JDBC Driver。所需的 JDBC Driver 与 ShardingSphere Proxy 设置的 `proxy-frontend-database-protocol-type` 对应。

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

享受集成

通过 Postgres JDBC Driver 创建 ShardingSphere 的数据源以享受集成。

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
@SuppressWarnings("SqlNoDataSourceInspection")
public class ExampleUtils {
    void test() throws SQLException {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:postgresql://127.0.0.1:3308/sharding_db");
        config.setDriverClassName("org.postgresql.Driver");
        try (HikariDataSource dataSource = new HikariDataSource(config);
             Connection connection = dataSource.getConnection();
             Statement statement = connection.createStatement()) {
            statement.execute("INSERT INTO t_order (user_id, order_type, address_
id, status) VALUES (1, 1, 1, 'INSERT_TEST')");
            statement.executeQuery("SELECT * FROM t_order");
            statement.execute("DELETE FROM t_order WHERE order_id=1");
```

```
        }
    }
}
```

9.2.7 会话管理

ShardingSphere 支持会话管理，可通过原生数据库的 SQL 查看当前会话或终止会话里面执行的 SQL。目前此功能仅限于存储节点为 MySQL 的情况，支持 MySQL `SHOW PROCESSLIST` 命令和 `KILL QUERY` 命令。

相关操作

查看会话

针对不同关联数据库支持不同的查看会话方法，关联 MySQL 数据库可使用 `SHOW PROCESSLIST` 命令查看会话。ShardingSphere 会自动生成唯一的 UUID 标识作为 ID，并将 SQL 执行信息存储在各个实例中。当执行此命令时，ShardingSphere 会通过治理中心收集并同步各个计算节点的 SQL 执行信息，然后汇总返回给用户。

```
mysql> show processlist;
+----------------------------------+------+-----------+-------------+---------+----
--+---------------+------------------+
| Id                               | User | Host      | db          | Command |
Time | State         | Info             |
+----------------------------------+------+-----------+-------------+---------+----
--+---------------+------------------+
| 05ede3bd584fd4a429dcaac382be2973 | root | 127.0.0.1 | sharding_db | Execute | 2
| Executing 0/1 | select sleep(10) |
| f9e5c97431567415fe10badc5fa46378 | root | 127.0.0.1 | sharding_db | Sleep   | 690
|               |                  |
+----------------------------------+------+-----------+-------------+---------+----
--+---------------+------------------+
```

- 输出说明

模拟原生 MySQL 的输出，但 `Id` 字段较为特殊为随机字符串。

终止会话语句

用户根据 `SHOW PROCESSLIST` 返回的结果，判断是否需要执行 `KILL QUERY` 语句，ShardingSphere 会根据 `KILL QUERY` 语句中的 ID 取消该 PROCESS ID 正在执行中的 SQL。

```
-- 取消指定会话里正在执行中的 SQL 语句。
mysql> KILL QUERY 05ede3bd584fd4a429dcaac382be2973;
Query OK, 0 rows affected (0.04 sec)
```

```
mysql> show processlist;
Empty set (0.02 sec)
```

9.2.8 日志配置

背景信息

ShardingSphere 使用 Logback 进行日志管理，内部采用 Java SPI 提供默认日志配置，用户可以使用 XML 文件来配置自定义日志输出，Proxy 将优先读取 `conf` 目录下的 `logback.xml` 提供的日志配置。

下面将介绍如何自定义日志配置。

操作步骤

1. 新建 `conf/logback.xml`

根据需求自定义 logger 级别、pattern 等。> 建议在配置示例的基础上进行修改

2. 查看日志

ShardingSphere-Proxy 启动后，日志将输出到 `logs` 目录下，选择目标日志文件进行查看。

配置示例

```
<?xml version="1.0"?>
<!--
  ~ Licensed to the Apache Software Foundation (ASF) under one or more
  ~ contributor license agreements.  See the NOTICE file distributed with
  ~ this work for additional information regarding copyright ownership.
  ~ The ASF licenses this file to You under the Apache License, Version 2.0
  ~ (the "License"); you may not use this file except in compliance with
  ~ the License.  You may obtain a copy of the License at
  ~
  ~     http://www.apache.org/licenses/LICENSE-2.0
  ~
  ~ Unless required by applicable law or agreed to in writing, software
  ~ distributed under the License is distributed on an "AS IS" BASIS,
  ~ WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  ~ See the License for the specific language governing permissions and
  ~ limitations under the License.
  -->

<configuration>
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %logger{36} -
%msg%n</pattern>
```

```
        </encoder>
    </appender>
    <logger name="org.apache.shardingsphere" level="info" additivity="false">
        <appender-ref ref="console" />
    </logger>

    <logger name="com.zaxxer.hikari" level="error" />

    <logger name="com.atomikos" level="error" />

    <logger name="io.netty" level="error" />

    <root>
        <level value="info" />
        <appender-ref ref="console" />
    </root>
</configuration>
```

9.2.9 CDC

CDC（Change Data Capture）增量数据捕捉。CDC 可以监控 ShardingSphere-Proxy 的存储节点中的数据变化，捕捉到数据操作事件，过滤并提取有用信息，最终将这些变化数据发送到指定的目标上。

CDC 可以用于数据同步，数据备份和恢复等方面，目前支持 openGauss、MySQL 和 PostgreSQL。

运行部署

背景信息

ShardingSphere CDC 分为两个部分，一个是 CDC Server，另一个是 CDC Client。CDC Server 和 ShardingSphere-Proxy 目前是一同部署的。

用户可以在自己的项目中引入 CDC Client，实现数据的消费逻辑。

约束条件

- 纯 JAVA 开发，JDK 建议 1.8 或以上版本。
- CDC Server 要求 ShardingSphere-Proxy 使用集群模式，目前支持 ZooKeeper 作为注册中心。
- CDC 只同步数据，不会同步表结构，目前也不支持 DDL 的语句同步。
- CDC 增量阶段会按照分库事务的维度输出数据，如果要开启 XA 事务的兼容，则 openGauss 和 ShardingSphere-Proxy 都需要 GLT 模块

CDC Server 部署步骤

这里以 openGauss 数据库为例，介绍 CDC Server 的部署步骤。

由于 CDC Server 内置于 ShardingSphere-Proxy，所以需要获取 ShardingSphere-Proxy。详情请参见 proxy 启动手册。

配置 GLT 模块（可选）

官网发布的二进制包默认不包含 GLT 模块，如果使用的是包含 GLT 功能的 openGauss 数据库，则可以额外引入 GLT 模块，保证 XA 事务的完整性。

目前有两种方式引入 GLT 模块，并且需要在 global.yaml 中也进行相应的配置。

1. 源码编译安装

1.1 准备代码环境，提前下载或者使用 Git clone，从 Github 下载 ShardingSphere 源码。

1.2 删除 kernel/global-clock/type/tso/core/pom.xml 中 shardingsphere-global-clock-tso-provider-redis 依赖的 `<scope>provided</scope>` 标签和 kernel/global-clock/type/tso/provider/redis/pom.xml 中 jedis 的 `<scope>provided</scope>` 标签

1.3 编译 ShardingSphere-Proxy，具体编译步骤请参考 ShardingSphere 编译手册。

2. 直接引入 GLT 依赖

可以从 maven 仓库中引入

2.1. shardingsphere-global-clock-tso-provider-redis，下载和 ShardingSphere-Proxy 同名版本

2.2. jedis-4.3.1

CDC Server 使用手册

修改配置文件 `conf/global.yaml`，打开 CDC 功能。目前 `mode` 必须是 `Cluster`，需要提前启动对应的注册中心。如果 GLT provider 使用 Redis，需要提前启动 Redis。

配置示例：

1. 在 `global.yaml` 中开启 CDC 功能。

```
mode:
  type: Cluster
  repository:
    type: ZooKeeper
    props:
      namespace: cdc_demo
      server-lists: localhost:2181
      retryIntervalMilliseconds: 500
```

```
      timeToLiveSeconds: 60
      maxRetries: 3
      operationTimeoutMilliseconds: 500

authority:
  users:
    - user: root@%
      password: root
  privilege:
    type: ALL_PERMITTED

# 使用 GLT 的时候也需要开启分布式事务，目前 GLT 只有 openGauss 数据库支持
#transaction:
#  defaultType: XA
#  providerType: Atomikos
#
#globalClock:
#  enabled: true
#  type: TSO
#  provider: redis
#  props:
#    host: 127.0.0.1
#    port: 6379

props:
  proxy-default-port: 3307 # Proxy default port
  cdc-server-port: 33071 # CDC Server 端口，必须配置
  proxy-frontend-database-protocol-type: openGauss # 和后端数据库的类型一致
```

2. 引入 JDBC 驱动。

proxy 已包含 PostgreSQL JDBC 和 openGauss JDBC 驱动。

如果后端连接以下数据库，请下载相应 JDBC 驱动 jar 包，并将其放入 `${shardingsphere-proxy}/ext-lib` 目录。

数据库	JDBC 驱动
MySQL	mysql-connector-j-8.3.0.jar

4. 启动 ShardingSphere-Proxy：

```
sh bin/start.sh
```

5. 查看 proxy 日志 `logs/stdout.log`，看到日志中出现：

```
[INFO ] [main] o.a.s.p.frontend.ShardingSphereProxy - ShardingSphere-Proxy Cluster mode started successfully
```

确认启动成功。

6. 按需配置 CDC 任务同步配置

6.1. 查询配置。

```
SHOW STREAMING RULE;
```

默认配置如下:

```
+-------------------------------------------------------------+-------------------
-------------------+-------------------------------------------------------+
| read                                                        | write
| stream_channel                                        |
+-------------------------------------------------------------+-------------------
-------------------+-------------------------------------------------------+
| {"workerThread":20,"batchSize":1000,"shardingSize":10000000} | {"workerThread
":20,"batchSize":1000} | {"type":"MEMORY","props":{"block-queue-size":"2000"}} |
+-------------------------------------------------------------+-------------------
-------------------+-------------------------------------------------------+
```

6.2. 修改配置（可选）。

因 streaming rule 具有默认值，无需创建，仅提供 ALTER 语句。

完整配置 DistSQL 示例:

```
ALTER STREAMING RULE (
READ(
  WORKER_THREAD=20,
  BATCH_SIZE=1000,
  SHARDING_SIZE=10000000,
  RATE_LIMITER (TYPE(NAME='QPS',PROPERTIES('qps'='500')))
),
WRITE(
  WORKER_THREAD=20,
  BATCH_SIZE=1000,
  RATE_LIMITER (TYPE(NAME='TPS',PROPERTIES('tps'='2000')))
),
STREAM_CHANNEL (TYPE(NAME='MEMORY',PROPERTIES('block-queue-size'='2000')))
);
```

配置项说明:

```
ALTER STREAMING RULE (
READ( -- 数据读取配置。如果不配置则部分参数默认生效。
  WORKER_THREAD=20, -- 影响全量、增量任务，从源端摄取数据的线程池大小，不配置则使用默认值，需要
确保该值不低于分库的数量。
  BATCH_SIZE=1000, -- 影响全量、增量任务，一次查询操作返回的最大记录数。如果一个事务中的数据量大
于该值，增量情况下可能超过设定的值。
  SHARDING_SIZE=10000000, -- 影响全量任务，存量数据分片大小。如果不配置则使用默认值。
  RATE_LIMITER ( -- 影响全量、增量任务，限流算法。如果不配置则不限流。
  TYPE( -- 算法类型。可选项: QPS
```

```
  NAME='QPS',
  PROPERTIES( -- 算法属性
  'qps'='500'
  )))
),
WRITE( -- 数据写入配置。如果不配置则部分参数默认生效。
  WORKER_THREAD=20, -- 影响全量、增量任务，数据写入到目标端的线程池大小。如果不配置则使用默认值。
  BATCH_SIZE=1000, -- 影响全量、增量任务，存量任务一次批量写入操作的最大记录数。如果不配置则使用默认值。如果一个事务中的数据量大于该值，增量情况下可能超过设定的值。
  RATE_LIMITER ( -- 限流算法。如果不配置则不限流。
  TYPE( -- 算法类型。可选项：TPS
  NAME='TPS',
  PROPERTIES( -- 算法属性
  'tps'='2000'
  )))
),
STREAM_CHANNEL ( -- 数据通道，连接生产者和消费者，用于 read 和 write 环节。如果不配置则默认使用 MEMORY 类型。
TYPE( -- 算法类型。可选项：MEMORY
NAME='MEMORY',
PROPERTIES( -- 算法属性
'block-queue-size'='2000' -- 属性：阻塞队列大小
)))
);
```

CDC Client 手册

CDC Client 不需要额外部署，只需要通过 maven 引入 CDC Client 的依赖就可以在项目中使用。用户可以通过 CDC Client 和服务端进行交互。

如果有需要，用户也可以自行实现一个 CDC Client，进行数据的消费和 ACK。

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-data-pipeline-cdc-client</artifactId>
    <version>${version}</version>
</dependency>
```

CDC Client 介绍

`org.apache.shardingsphere.data.pipeline.cdc.client.CDCClient` 是 CDC Client 的入口类，用户可以通过该类和 CDC Server 进行交互。主要的和新方法如下。

方法名	返回值	说明
connect(Consumer<List> dataConsumer, ExceptionHandler exceptionHandler, ServerErrorResultHandler errorResultHandler	void	和服务端进行连接，连接的时候需要指定 1. 数据的消费处理逻辑 2. 消费时候的异常处理逻辑 3. 服务端错误的异常处理逻辑
login(CDCLoginParameter parameter)	void	CDC 登陆，参数 username：用户名 password：密码
startStreaming(StartStreamingParameter parameter)	streamingId	开启 CDC 订阅 StartStreamingParameter 参数 database：逻辑库名称 schemaTables：订阅的表名 full：是否订阅全量数据
restartStreaming(String streamingId)	void	重启订阅
stopStreaming(String streamingId)	void	停止订阅
dropStreaming(String streamingId)	void	删除订阅
await()	void	阻塞 CDC 线程，等待 channel 关闭
close()	void	关闭 channel，流程结束

使用手册

CDC 功能介绍

CDC 只会同步数据，不会同步表结构，目前也不支持 DDL 的语句同步。

CDC 协议介绍

CDC 协议使用 Protobuf，对应的 Protobuf 类型是根据 Java 中的类型来映射的。

这里以 openGauss 为例，CDC 协议的数据类型和数据库类型的映射关系如下

openGauss 类型	Java 数据类型	CDC 对应的 protobuf 类型	备注
tinyi nt、smallint、integer	Integer	int32	
bigint	Long	int64	
numeric	Bi gDecimal	string	
real、float4	Float	float	
binary_double、double precision	Double	double	
boolean	Boolean	bool	
char 、varchar、text、clob	String	string	
blob、bytea、raw	byte[]	bytes	
d ate、timestamp，times tamptz、smalldatetime	ja va.sql.T imestamp	Timestamp	protobuf 的 Timestamp 类型只包含秒和纳秒，所以和时区无关
time、timetz	java. sql.Time	int64	代表当天的纳秒数，和时区无关
inter val、reltime、abstime	String	string	
point、lseg、box、path、polygon、circle	String	string	
cidr、inet、macaddr	String	string	
tsvector	String	string	
tsquery	String	String	
uuid	String	string	
json、jsonb	String	string	
hll	String	string	
int4range、daterang e、tsrange、tstzrange	String	string	
hash16、hash32	String	string	
bit、bit varying	String	string	bit(1) 的时候返回 Boolean 类型

openGauss 使用手册

环境要求

支持的 openGauss 版本：2.x ~ 3.x。

权限要求

1. 调整源端 WAL 配置。

`postgresql.conf` 示例配置:

```
wal_level = logical
max_wal_senders = 10
max_replication_slots = 10
wal_sender_timeout = 0
max_connections = 600
```

详情请参见 Write Ahead Log 和 Replication。

2. 赋予源端 openGauss 账号 replication 权限。

`pg_hba.conf` 示例配置:

```
host replication repl_acct 0.0.0.0/0 md5
# 0.0.0.0/0 表示允许任意 IP 地址访问，可以根据实际情况调整成 CDC Server 的 IP 地址
```

详情请参见 Configuring Client Access Authentication 和 Example: Logic Replication Code。

3. 赋予 openGauss 账号 DDL DML 权限。

如果使用非超级管理员账号，要求该账号在用到的数据库上，具备 CREATE 和 CONNECT 的权限。

示例:

```
GRANT CREATE, CONNECT ON DATABASE source_ds TO cdc_user;
```

还需要账号对订阅的表和 schema 具备访问权限，以 test schema 下的 t_order 表为例。

```
\c source_ds

GRANT USAGE ON SCHEMA test TO GROUP cdc_user;
GRANT SELECT ON TABLE test.t_order TO cdc_user;
```

openGauss 有 OWNER 的概念，如果是数据库，SCHEMA，表的 OWNER，则可以省略对应的授权步骤。

openGauss 不允许普通账户在 public schema 下操作。所以如果迁移的表在 public schema 下，需要额外授权。

```
GRANT ALL PRIVILEGES TO cdc_user;
```

详情请参见 openGauss GRANT

完整流程示例

前提条件

1. 准备好 CDC 源端的库、表、数据。

```
DROP DATABASE IF EXISTS ds_0;
CREATE DATABASE ds_0;

DROP DATABASE IF EXISTS ds_1;
CREATE DATABASE ds_1;
```

配置 CDC Server

1. 创建逻辑库。

```
CREATE DATABASE sharding_db;

\c sharding_db
```

2. 注册存储单元。

```
REGISTER STORAGE UNIT ds_0 (
    URL="jdbc:opengauss://127.0.0.1:5432/ds_0",
    USER="gaussdb",
    PASSWORD="Root@123",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
), ds_1 (
    URL="jdbc:opengauss://127.0.0.1:5432/ds_1",
    USER="gaussdb",
    PASSWORD="Root@123",
    PROPERTIES("minPoolSize"="1","maxPoolSize"="20","idleTimeout"="60000")
);
```

3. 创建分片规则。

```
CREATE SHARDING TABLE RULE t_order(
STORAGE_UNITS(ds_0,ds_1),
SHARDING_COLUMN=order_id,
TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="2")),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME="snowflake"))
);
```

4. 创建表

在 proxy 执行建表语句。

```
CREATE TABLE t_order (order_id INT NOT NULL, user_id INT NOT NULL, status
VARCHAR(45) NULL, PRIMARY KEY (order_id));
```

启动 CDC Client

目前 CDC Client 只提供了 Java API，用户需要自行实现数据的消费逻辑。

下面是一个简单的启动 CDC Client 的示例。

```
import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;
import org.apache.shardingsphere.data.pipeline.cdc.client.CDCClient;
import org.apache.shardingsphere.data.pipeline.cdc.client.config.
CDCClientConfiguration;
import org.apache.shardingsphere.data.pipeline.cdc.client.handler.
RetryStreamingExceptionHandler;
import org.apache.shardingsphere.data.pipeline.cdc.client.parameter.
CDCLoginParameter;
import org.apache.shardingsphere.data.pipeline.cdc.client.parameter.
StartStreamingParameter;
import org.apache.shardingsphere.data.pipeline.cdc.protocol.request.
StreamDataRequestBody.SchemaTable;

import java.util.Collections;

@Slf4j
public final class Bootstrap {

    @SneakyThrows(InterruptedException.class)
    public static void main(final String[] args) {
        String address = "127.0.0.1";
        // 构造 CDCClient，传入 CDCClientConfiguration，CDCClientConfiguration 中包含了
CDC Server 的地址和端口，以及超时时间
        try (CDCClient cdcClient = new CDCClient(new
CDCClientConfiguration(address, 33071, 10000))) {
            // 先调用 connect 连接到 CDC Server，需要传入 1. 数据的消费处理逻辑 2. 消费时候
的异常处理逻辑 3. 服务端错误的异常处理逻辑
            cdcClient.connect(records -> log.info("records: {}", records), new
RetryStreamingExceptionHandler(cdcClient, 5, 5000),
                    (ctx, result) -> log.error("Server error: {}", result.
getErrorMessage()));
            cdcClient.login(new CDCLoginParameter("root", "root"));
            // 开始 CDC 数据同步，返回的 streamingId 是这次 CDC 任务的唯一标识，CDC Server
生成唯一标识的依据是 订阅的数据库名称 + 订阅的表 + 是否是全量同步
            String streamingId = cdcClient.startStreaming(new
StartStreamingParameter("sharding_db", Collections.singleton(SchemaTable.
newBuilder().setTable("t_order").build()), true));
            log.info("Streaming id={}", streamingId);
```

```
            // 防止 main 主线程退出
            cdcClient.await();
        }
    }
}
```

主要有 4 个步骤 1. 构造 CDCClient，传入 CDCClientConfiguration 2. 调用 CDCClient.connect，这一步是和 CDC Server 建立连接 3. 调用 CDCClient.login，使用 global.yaml 中配置好的用户名和密码登录 4. 调用 CDCClient.startStreaming，开启订阅，需要保证订阅的库和表在 ShardingSphere-Proxy 存在，否则会报错。

> CDCClient.await 是阻塞主线程，非必需的步骤，用其他方式也可以，只要保证 CDC 线程一直在工作就行。

如果需要更复杂数据消费的实现，例如写入到数据库，可以参考 DataSourceRecordConsumer

写入数据

通过 proxy 写入数据，此时 CDC Client 会收到数据变更的通知。

```
INSERT INTO t_order (order_id, user_id, status) VALUES (1,1,'ok1'),(2,2,'ok2'),(3,
3,'ok3');
UPDATE t_order SET status='updated' WHERE order_id = 1;
DELETE FROM t_order WHERE order_id = 2;
```

Bootstrap 会输出类似的日志

```
records: [before {
name: "order_id"
value {
  type_url: "type.googleapis.com/google.protobuf.Empty"
}
......
```

查看 CDC 任务运行情况

CDC 任务的启动和停止目前只能通过 CDC Client 控制，可以通过在 proxy 中执行 DistSQL 查看 CDC 任务状态

1. 查看 CDC 任务列表

SHOW STREAMING LIST;

运行结果

```
sharding_db=> SHOW STREAMING LIST;
                     id                     |  database   | tables  | job_item_
count | active |     create_time     | stop_time
--------------------------------------------+-------------+---------+--------------
```

```
--+--------+---------------------+-----------
 j0302p0000702a83116fcee83f70419ca5e2993791 | sharding_db | t_order | 1
| true   | 2023-10-27 22:01:27 |
(1 row)
```

2. 查看 CDC 任务详情

SHOW STREAMING STATUS j0302p0000702a83116fcee83f70419ca5e2993791;

运行结果

```
sharding_db=> SHOW STREAMING STATUS j0302p0000702a83116fcee83f70419ca5e2993791;
 item | data_source |          status          | active | processed_records_count |
inventory_finished_percentage | incremental_idle_seconds | confirmed_position |
current_position | error_message
------+-------------+--------------------------+--------+-------------------------
+------------------------------+--------------------------+--------------------+--
----------------+---------------
 0    | ds_0        | EXECUTE_INCREMENTAL_TASK | false  | 2                       |
100                           | 115                      | 5/597E43D0         | 5/
597E4810       |
 1    | ds_1        | EXECUTE_INCREMENTAL_TASK | false  | 3                       |
100                           | 115                      | 5/597E4450         | 5/
597E4810       |
(2 rows)
```

3. 删除 CDC 任务

DROP STREAMING j0302p0000702a83116fcee83f70419ca5e2993791;

只有当 CDC 任务没有订阅的时候才可以删除，此时也会删除 openGauss 物理库上的 replication slots

```
sharding_db=> DROP STREAMING j0302p0000702a83116fcee83f70419ca5e2993791;
SUCCESS
```

注意事项

增量数据推送的说明

1. CDC 增量推送目前是按照事务维度的，物理库的事务不会被拆分，所以如果一个事务中有多个表的数据变更，那么这些数据变更会被一起推送。如果要支持 XA 事务（目前只支持 openGauss），则 openGauss 和 Proxy 都需要 GLT 模块。
2. 满足推送的条件是满足了一定大小的数据量或者到了一定的时间间隔（目前是 300ms），在处理 XA 事务时，收到的多个分库增量事件超过了 300ms，可能会导致 XA 事务被拆开推送。

超大事务的处理

目前是将大事务完整解析，这样可能会导致 CDC Server 进程 OOM，后续可能会考虑强制截断。

建议的配置

CDC 的性能目前没有一个固定的值，可以关注配置中读/写的 batchSize，以及内存队列的大小，根据实际情况进行调优。

9.3 通用配置

本章主要介绍通用配置，包括属性配置和内置算法配置。

9.3.1 属性配置

背景信息

Apache ShardingSphere 提供属性配置的方式配置系统级配置。

参数解释

操作步骤

属性配置直接配置在 ShardingSphere-JDBC 所使用的配置文件中，格式如下：

```
props:
    sql-show: true
```

注意事项

`max-connections-size-per-query` 配置项的默认值为 1，这意味着每次查询请求在每个数据库实例中只能使用一个连接。如果您调整此参数以启用内存限制模式（详情参见 内存限制模式），请确保您使用的数据库 JDBC 实现支持流式查询或能够启用流式查询。例如，在 MySQL 中，需要设置 `statement.setFetchSize(Integer.MIN_VALUE)` 才能实现流式查询。

配置示例

ShardingSphere 仓库的示例中包含了多种不同场景的属性配置，请参考：https://github.com/apache/shardingsphere/blob/master/examples

9.3.2 内置算法

简介

Apache ShardingSphere 通过 SPI 方式允许开发者扩展算法；与此同时，Apache ShardingSphere 也提供了大量的内置算法以便于开发者使用。

使用方式

内置算法均通过 type 和 props 进行配置，其中 type 由算法定义在 SPI 中，props 用于传递算法的个性化参数配置。

无论使用哪种配置方式，均是将配置完毕的算法命名，并传递至相应的规则配置中。本章节根据功能区分并罗列 Apache ShardingSphere 全部的内置算法，供开发者参考。

元数据持久化仓库

背景信息

Apache ShardingSphere 为不同的运行模式提供了不同的元数据持久化方式，用户在配置运行模式的同时可以选择合适的方式来存储元数据。

参数解释

数据库持久化

`provider` 的可选值为 H2，MySQL，EmbeddedDerby，DerbyNetworkServer，HSQLDB。由于第三方的 Vulnerability Report 时常误报 H2 Database，避免在 ShardingSphere Standalone Mode 使用 H2 Database 可能是一种选择。讨论 `provider` 不为默认值 `H2` 的情况。

1. 若 `provider` 设置为 `MySQL`，则要求存在已就绪的 MySQL Server。classpath 应包含 `com.mysql:mysql-connector-j:9.0.0` 的 Maven 依赖。
2. 若 `provider` 设置为 `EmbeddedDerby`，则 Derby 数据库引擎将在与应用程序相同的 JVM 内运行。classpath 应包含 `org.apache.derby:derby:10.17.1.0` 和 `org.apache.derby:derbytools:10.17.1.0` 的 Maven 依赖，且要求编译或运行下游项目的 JDK 版本大于或等于 JDK19。可能的配置如下。

```
mode:
  type: Standalone
  repository:
```

```
    type: JDBC
    props:
      provider: EmbeddedDerby
      jdbc_url: jdbc:derby:memory:config;create=true
      username:
```

3. 若 `provider` 设置为 `DerbyNetworkServer`，则要求存在已就绪的 Derby Network Server。Derby Network Server 不存在可用的 Docker Image，用户可能需要手动启动 Derby Network Server。classpath 应包含 `org.apache.derby:derbyclient:10.17.1.0` 和 `org.apache.derby:derbytools:10.17.1.0` 的 Maven 依赖，且要求编译或运行下游项目的 JDK 版本大于或等于 JDK19。
4. 若 `provider` 设置为 `HSQLDB`，则要求存在已就绪的采用 Server Modes 的 HyperSQL，或以 in-process database 的方式创建数据库。classpath 应包含 `classifier` 为 `jdk8` 的 `org.hsqldb:hsqldb:2.7.3` 的 Maven 依赖。采用 Server Modes 的 HyperSQL 不存在可用的 Docker Image，用户可能需要手动启动 Server Modes 的 HyperSQL。若使用 mem: protocol 的 HyperSQL，则可能的配置如下，

```
mode:
  type: Standalone
  repository:
    type: JDBC
    props:
      provider: HSQLDB
      jdbc_url: jdbc:hsqldb:mem:config
      username: SA
```

类型：JDBC

适用模式：Standalone

可配置属性：

名称	数据类型 *	说明	默认值
provider	String	元数据存储类型	H2
jdbc_url	String	JDBC URL	jdbc:h2:mem:config;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=M
username	String	账号	sa
password	String	密码	

ZooKeeper 持久化

类型：ZooKeeper

适用模式：Cluster

可配置属性：

名称	数据类型	说明	默认值
retryIntervalMilliseconds	int	重试间隔毫秒数	500
maxRetries	int	客户端连接最大重试次数	3
timeToLiveSeconds	int	临时数据失效的秒数	60
operationTimeoutMilliseconds	int	客户端操作超时的毫秒数	500
digest	String	登录认证密码	

Etcd 持久化

类型：Etcd

适用模式：Cluster

可配置属性：

名称	数据类型	说明	默认值
timeToLiveSeconds	long	临时数据失效的秒数	30
connectionTimeout	long	连接超时秒数	30

操作步骤

1. 在 global.yaml 中配置 Mode 运行模式
2. 配置元数据持久化仓库类型

配置示例

- 单机模式配置方式

```
mode:
  type: Standalone
  repository:
    type: JDBC
    props:
      provider: H2
      jdbc_url: jdbc:h2:mem:config;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;
MODE=MYSQL
```

```
      username: test
      password: Test@123
```

- 集群模式

```
mode:
  type: Cluster
  repository:
    type: zookeeper
    props:
      namespace: governance_ds
      server-lists: localhost:2181
      retryIntervalMilliseconds: 500
      timeToLiveSeconds: 60
      maxRetries: 3
      operationTimeoutMilliseconds: 500
```

分片算法

背景信息

ShardingSphere 内置提供了多种分片算法，按照类型可以划分为自动分片算法、标准分片算法、复合分片算法和 Hint 分片算法，能够满足用户绝大多数业务场景的需要。此外，考虑到业务场景的复杂性，内置算法也提供了自定义分片算法的方式，用户可以通过编写 Java 代码来完成复杂的分片逻辑。需要注意的是，自动分片算法的分片逻辑由 ShardingSphere 自动管理，需要通过配置 autoTables 分片规则进行使用。

参数解释

自动分片算法

取模分片算法

类型：MOD

可配置属性：

属性名称	数据类型	说明
sharding-count	int	分片数量

哈希取模分片算法

类型：HASH_MOD

可配置属性：

属性名称	数据类型	说明
sharding-count	int	分片数量

基于分片容量的范围分片算法

类型：VOLUME_RANGE

可配置属性：

属性名称	数据类型	说明
range-lower	long	范围下界，超过边界的数据会报错
range-upper	long	范围上界，超过边界的数据会报错
sharding-volume	long	分片容量

基于分片边界的范围分片算法

类型：BOUNDARY_RANGE

可配置属性：

属性名称	数据类型	说明
sharding-ranges	String	分片的范围边界，多个范围边界以逗号分隔

自动时间段分片算法

类型：AUTO_INTERVAL

可配置属性：

属性名称	数据类型	说明
d atetime-lower	St ring	分片的起始时间范围，时间戳格式：yyyy-MM-dd HH:mm:ss
d atetime-upper	St ring	分片的结束时间范围，时间戳格式：yyyy-MM-dd HH:mm:ss
sha rding-seconds	long	单一分片所能承载的最大时间，单位：秒，允许分片键的时间戳格式的秒带有时间精度，但秒后的时间精度会被自动抹去

标准分片算法

Apache ShardingSphere 内置的标准分片算法实现类包括：

行表达式分片算法

使用 `InlineExpressionParser` SPI 的默认实现的 Groovy 的表达式，提供对 SQL 语句中的 = 和 `IN` 的分片操作支持，只支持单分片键。对于简单的分片算法，可以通过简单的配置使用，从而避免繁琐的 Java 代码开发，如: `t_user_$->{u_id % 8}` 表示 `t_user` 表根据 `u_id` 模 8，而分成 8 张表，表名称为 `t_user_0` 到 `t_user_7`。详情请参见行表达式。

类型：INLINE

可配置属性：

属性名称	数据类型	说明	• 默认值 *
algorithm-expression	St ring	分片算法的行表达式	
allow-r ange-query-with-inline-sharding (?)	boo lean	是否允许范围查询。注意：范围查询会无视分片策略，进行全路由	f al se

时间范围分片算法

此算法主动忽视了 `datetime-pattern` 的时区信息。这意味着当 `datetime-lower`, `datetime-upper` 和传入的分片键含有时区信息时，不会因为时区不一致而发生时区转换。

当传入的分片键为 `java.time.Instant` 或 `java.util.Date` 时存在特例处理，其会携带上系统的时区信息后转化为 `datetime-pattern` 的字符串格式，再进行下一步分片。

类型：INTERVAL

可配置属性：

复合分片算法

复合行表达式分片算法

详情请参见行表达式。

类型：COMPLEX_INLINE

属性名称	数据类型	说明	默认值 *
sharding-columns (?)	St ring	分片列名称，多个列用逗号分隔。如不配置无法则不能校验	
algorithm-expression	St ring	分片算法的行表达式	
allow-r ange-query-with-inline-sharding (?)	boo lean	是否允许范围查询。注意：范围查询会无视分片策略，进行全路由	f al se

Hint 分片算法

Hint 行表达式分片算法

详情请参见行表达式。

类型：HINT_INLINE

属性名称	数据类型	说明	默认值
algorithm-expression (?)	String	分片算法的行表达式	${value}

自定义类分片算法

通过配置分片策略类型和算法类名，实现自定义扩展。`CLASS_BASED` 允许向算法类内传入额外的自定义属性，传入的属性可以通过属性名为 `props` 的 `java.util.Properties` 类实例取出。参考 Git 的 `org.apache.shardingsphere.example.extension.sharding.algortihm.classbased.fixture.ClassBasedStandardShardingAlgorithmFixture`。

类型：CLASS_BASED

可配置属性：

属性名称	数据类型	说明
strategy	String	分片策略类型，支持 STANDARD、COMPLEX 或 HINT（不区分大小写）
algorithmClassName	String	分片算法全限定名

操作步骤

1. 使用数据分片时，在 shardingAlgorithms 属性下配置对应的数据分片算法即可；

配置示例

```
rules:
- !SHARDING
  tables:
    t_order:
      actualDataNodes: ds_${0..1}.t_order_${0..1}
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: t_order_inline
      keyGenerateStrategy:
        column: order_id
        keyGeneratorName: snowflake
    t_order_item:
      actualDataNodes: ds_${0..1}.t_order_item_${0..1}
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: t_order_item_inline
      keyGenerateStrategy:
        column: order_item_id
        keyGeneratorName: snowflake
    t_account:
      actualDataNodes: ds_${0..1}.t_account_${0..1}
      tableStrategy:
        standard:
          shardingAlgorithmName: t_account_inline
      keyGenerateStrategy:
        column: account_id
        keyGeneratorName: snowflake
  defaultShardingColumn: account_id
  bindingTables:
```

```
    - t_order,t_order_item
  defaultDatabaseStrategy:
    standard:
      shardingColumn: user_id
      shardingAlgorithmName: database_inline
  defaultTableStrategy:
    none:

  shardingAlgorithms:
    database_inline:
      type: INLINE
      props:
        algorithm-expression: ds_${user_id % 2}
    t_order_inline:
      type: INLINE
      props:
        algorithm-expression: t_order_${order_id % 2}
    t_order_item_inline:
      type: INLINE
      props:
        algorithm-expression: t_order_item_${order_id % 2}
    t_account_inline:
      type: INLINE
      props:
        algorithm-expression: t_account_${account_id % 2}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE

- !BROADCAST
  tables:
    - t_address
```

相关参考

- 核心特性：数据分片
- 开发者指南：数据分片

分布式序列算法

背景信息

传统数据库软件开发中，主键自动生成技术是基本需求。而各个数据库对于该需求也提供了相应的支持，比如 MySQL 的自增键，Oracle 的自增序列等。数据分片后，不同数据节点生成全局唯一主键是非常棘手的问题。同一个逻辑表内的不同实际表之间的自增键由于无法互相感知而产生重复主键。虽然可通过约束自增主键初始值和步长的方式避免碰撞，但需引入额外的运维规则，使解决方案缺乏完整性和可扩展性。

目前有许多第三方解决方案可以完美解决这个问题，如 UUID 等依靠特定算法自生成不重复键，或者通过引入主键生成服务等。为了方便用户使用、满足不同用户不同使用场景的需求，Apache ShardingSphere 不仅提供了内置的分布式主键生成器，例如 UUID、SNOWFLAKE，还抽离出分布式主键生成器的接口，方便用户自行实现自定义的自增主键生成器。

参数解释

雪花算法

类型：SNOWFLAKE

可配置属性：

注意：worker-id 为选配项 1. 在单机模式下支持用户自定义配置，如果用户不配置使用默认值为 0。2. 在集群模式下会由系统自动生成，相同的命名空间下不会生成重复的值。

UUID

类型：UUID

可配置属性：无

操作步骤

1. 配置数据分片规则时为列配置分布式主键生成策略

配置示例

- 雪花算法

```
keyGenerators:
  snowflake:
    type: SNOWFLAKE
```

- UUID

```
keyGenerators:
  uuid:
    type: UUID
```

负载均衡算法

背景信息

ShardingSphere 内置提供了多种负载均衡算法，具体包括了轮询算法、随机访问算法和权重访问算法，能够满足用户绝大多数业务场景的需要。此外，考虑到业务场景的复杂性，内置算法也提供了扩展方式，用户可以基于 SPI 接口实现符合自己业务需要的负载均衡算法。

参数解释

轮询负载均衡算法

类型：ROUND_ROBIN

随机负载均衡算法

类型：RANDOM

权重负载均衡算法

类型：WEIGHT

可配置属性：

属性名称	数据类型	说明
${replica-name}	double	属性名使用读库名称，参数填写读库对应的权重值。权重参数范围最小值 > 0，合计 <= Double.MAX_VALUE。

操作步骤

1. 使用读写分离时，在 loadBalancers 属性下配置对应的负载均衡算法即可；

配置示例

```
rules:
- !READWRITE_SPLITTING
  dataSourceGroups:
    readwrite_ds:
      writeDataSourceName: write_ds
      readDataSourceNames:
        - read_ds_0
        - read_ds_1
      loadBalancerName: random
      transactionalReadQueryStrategy: PRIMARY
  loadBalancers:
    random:
      type: RANDOM
      props:
```

相关参考

- 核心特性：读写分离
- 开发者指南：读写分离

加密算法

背景信息

加密算法是 Apache ShardingSphere 的加密功能使用的算法，ShardingSphere 内置了多种算法，可以让用户方便使用。

参数解释

标准加密算法

AES 加密算法

类型：AES

可配置属性：

名称	数据类型	说明
aes-key-value	String	AES 使用的 KEY
digest-algorithm-name	String	AES KEY 的摘要算法

辅助查询加密算法

MD5 辅助查询加密算法

类型：MD5

可配置属性：

名称	数据类型	说明
salt	String	盐值（可选）

操作步骤

1. 在加密规则中配置加密器
2. 为加密器指定加密算法类型

配置示例

```
rules:
- !ENCRYPT
  tables:
    t_user:
      columns:
        username:
          cipher:
            name: username
            encryptorName: name_encryptor
          assistedQuery:
            name: assisted_username
            encryptorName: assisted_encryptor
  encryptors:
    name_encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
        digest-algorithm-name: SHA-1
    assisted_encryptor:
      type: MD5
      props:
        salt: 123456
```

相关参考

- 核心特性：数据加密
- 开发者指南：数据加密

影子算法

背景信息

影子库功能对执行的 SQL 语句进行影子判定。影子判定支持两种类型算法，用户可根据实际业务需求选择一种或者组合使用。

参数解释

列影子算法

列值匹配算法

类型：VALUE_MATCH

属性名称	数据类型	说明
column	String	影子列
operation	String	SQL 操作类型（INSERT, UPDATE, DELETE, SELECT)
value	String	影子列匹配的值

列正则表达式匹配算法

类型：REGEX_MATCH

属性名称	数据类型	说明
column	String	匹配列
operation	String	SQL 操作类型 (INSERT, UPDATE, DELETE, SELECT)
regex	String	影子列匹配正则表达式

Hint 影子算法

SQL HINT 影子算法

类型：SQL_HINT

```
/* SHARDINGSPHERE_HINT: SHADOW=true */
```

配置示例

- Java API

```
public final class ShadowConfiguration {
    // ...

    private AlgorithmConfiguration createShadowAlgorithmConfiguration() {
        Properties userIdInsertProps = new Properties();
        userIdInsertProps.setProperty("operation", "insert");
        userIdInsertProps.setProperty("column", "user_id");
        userIdInsertProps.setProperty("value", "1");
        return new AlgorithmConfiguration("VALUE_MATCH", userIdInsertProps);
    }

    // ...
}
```

- YAML:

```
shadowAlgorithms:
  user-id-insert-algorithm:
    type: VALUE_MATCH
    props:
      column: user_id
      operation: insert
      value: 1
```

SQL 翻译

原生 SQL 翻译器

类型：NATIVE

可配置属性：

无

默认使用的 SQL 翻译器，但目前暂未实现

分片审计算法

背景信息

分片审计功能是针对数据库分片场景下对执行的 SQL 语句进行审计操作。分片审计既可以进行拦截操作，拦截系统配置的非法 SQL 语句，也可以是对 SQL 语句进行统计操作。

参数解释

DML_SHARDING_CONDITIONS 算法

类型：DML_SHARDING_CONDITIONS

操作步骤

1. 配置数据分片规则时设置分配审计生成策略

配置示例

- DML_SHARDING_CONDITIONS

```
auditors:
  sharding_key_required_auditor:
    type: DML_SHARDING_CONDITIONS
```

脱敏算法

背景信息

脱敏算法是 Apache ShardingSphere 的脱敏功能使用的算法，ShardingSphere 内置了多种算法，可以让用户方便使用。

参数解释

哈希脱敏算法

MD5 脱敏算法

类型：MD5

可配置属性：

名称	数据类型	说明
salt	String	盐值（可选）

遮盖脱敏算法

保留前 N 后 M 脱敏算法

类型：KEEP_FIRST_N_LAST_M

可配置属性：

名称	数据类型	说明
first-n	int	前 n 位
last-m	int	后 n 位
replace-char	String	替换字符

保留自 X 至 Y 脱敏算法

类型：KEEP_FROM_X_TO_Y

可配置属性：

名称	数据类型	说明
from-x	int	起始位置（从 0 开始）
to-y	int	结束位置（从 0 开始）
replace-char	String	替换字符

遮盖前 N 后 M 脱敏算法

类型：MASK_FIRST_N_LAST_M

可配置属性：

名称	数据类型	说明
first-n	int	前 n 位
last-m	int	后 n 位
replace-char	String	替换字符

遮盖自 X 至 Y 脱敏算法

类型：MASK_FROM_X_TO_Y

可配置属性：

名称	数据类型	说明
from-x	int	起始位置（从 0 开始）
to-y	int	结束位置（从 0 开始）
replace-char	String	替换字符

特殊字符前遮盖脱敏算法

类型：MASK_BEFORE_SPECIAL_CHARS

可配置属性：

名称	数据类型	说明
special-chars	String	特殊字符（首次出现）
replace-char	String	替换字符

特殊字符后遮盖脱敏算法

类型：MASK_AFTER_SPECIAL_CHARS

可配置属性：

名称	数据类型	说明
special-chars	String	特殊字符（首次出现）
replace-char	String	替换字符

替换脱敏算法

通用表格随机替换

类型：GENERIC_TABLE_RANDOM_REPLACE

可配置属性：

名称	数据类型 *	说明
uppercase -letter-codes	St ri ng	大写字母码表（以英文逗号分隔，默认值：A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z）
lowercase -letter-codes	St ri ng	小写字母码表（以英文逗号分隔，默认值：a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z）
digital-codes	St ri ng	数字码表（以英文逗号分隔，默认值：0,1,2,3,4,5,6,7,8,9）
special-codes	St ri ng	特殊字符码表（以英文逗号分隔，默认值：~,!,@,#,$,%,^,&,*,:,<,>,|）

操作步骤

1. 在脱敏规则中配置脱敏算法；
2. 为脱敏算法指定脱敏算法类型。

配置示例

```
rules:
- !MASK
  tables:
    t_user:
      columns:
        password:
          maskAlgorithm: md5_mask
        email:
          maskAlgorithm: mask_before_special_chars_mask
        telephone:
          maskAlgorithm: keep_first_n_last_m_mask

  maskAlgorithms:
    md5_mask:
      type: MD5
    mask_before_special_chars_mask:
      type: MASK_BEFORE_SPECIAL_CHARS
      props:
        special-chars: '@'
        replace-char: '*'
    keep_first_n_last_m_mask:
      type: KEEP_FIRST_N_LAST_M
```

```
    props:
      first-n: 3
      last-m: 4
      replace-char: '*'
```

相关参考

- 核心特性：数据脱敏
- 开发者指南：数据脱敏

行表达式

使用 Groovy 语法的行表达式

在配置中使用 `${ expression }` 或 `$->{ expression }` 标识 Groovy 表达式即可。Groovy 表达式使用的是 Groovy 的语法，Groovy 能够支持的所有操作，行表达式均能够支持。`${begin..end}` 表示范围区间；`${[unit1, unit2, unit_x]}` 表示枚举值。行表达式中如果出现连续多个 `${ expression }` 或 `$->{ expression }` 表达式，整个表达式最终的结果将会根据每个子表达式的结果进行笛卡尔组合。

类型：GROOVY

用例：

- `<GROOVY>t_order_${1..3}` 将被转化为 `t_order_1, t_order_2, t_order_3`
- `<GROOVY>${['online', 'offline']}_table${1..3}` 将被转化为 `online_table1, online_table2, online_table3, offline_table1, offline_table2, offline_table3`

使用标准列表的行表达式

`LITERAL` 实现将不对表达式部分做任何符号的转化，从标准列表的输入直接获得标准列表的输出。此有助于解决 GraalVM Native Image 下不便于使用 Groovy 表达式的问题。

类型：LITERAL

用例：

- `<LITERAL>t_order_1, t_order_2, t_order_3` 将被转化为 `t_order_1, t_order_2, t_order_3`
- `<LITERAL>t_order_${1..3}` 将被转化为 `t_order_${1..3}`

基于固定时间范围的 Key-Value 语法的行表达式

`INTERVAL` 实现引入了 Key-Value 风格的属性语法，来通过单行字符串定义一组时间范围内的字符串。这通常用于简化对 数据分片功能的 `actualDataNodes` 的定义。`INTERVAL` 生成的字符串仅适用于单个真实库中的多张表。若需要处理多个真实库中的多张表，考虑自行创建 `org.apache.shardingsphere.infra.expr.spi.InlineExpressionParser` 的 SPI 实现类。

`INTERVAL` 实现定义多个属性的方式为 `Key1=Value1;Key2=Value2`，通过 `;` 号分割键值对，通过 `=` 号分割 `Key` 值和 `Value` 值。

此实现主动忽视了 `SP` 的时区信息，这意味着当 `DL` 和 `DU` 含有时区信息时，不会因为时区不一致而发生时区转换。

此实现键值对的顺序不敏感，行表达式尾部不携带 `;` 号。

`INTERVAL` 实现引入了如下 Key 的键值：

1. `P` 代表 prefix 的缩写，意为结果列表单元的前缀，通常代表真实表的前缀格式。
2. `SP` 代表 suffix pattern 的缩写，意为结果列表单元的后缀的时间戳格式，通常代表真实表的后缀格式，必须遵循 Java DateTimeFormatter 的格式。例如：yyyyMMdd，yyyyMM 或 yyyy 等。
3. `DIA` 代表 datetime interval amount 的缩写，意为结果列表单元的时间间隔。
4. `DIU` 代表 datetime interval unit 的缩写，意为分片键时间间隔单位，必须遵循 Java `java.time.temporal.ChronoUnit#toString()` 的枚举值。例如：`Months`。
5. `DL` 代表 datetime lower 的缩写，意为时间下界值，格式与 `SP` 定义的时间戳格式一致。
6. `DU` 代表 datetime upper 的缩写，意为时间上界值，格式与 `SP` 定义的时间戳格式一致。
7. `C` 代表 chronology 的缩写，意为日历系统，必须遵循 Java `java.time.chrono.Chronology#getId()` 的格式。例如：`Japanese`，`Minguo`，`ThaiBuddhist`。存在默认值为 `ISO`。

对于 `C` 的 Key 对应的 Value 是否可用，这取决于 JVM 所处的系统环境。这意味着如果用户需要设置 `C=Japanese`，则可能需要在应用的启动类调用 `java.util.Locale.setDefault(java.util.Locale.JAPAN);` 以修改系统环境。讨论两种 JVM 环境。

1. Hotspot JVM 在 RunTime 决定 `java.util.Locale.getDefault()` 的返回值。
2. GraalVM Native Image 在 BuildTime 决定 `java.util.Locale.Locale.getDefault()` 的返回值，与 Hotspot JVM 的表现并不一致，参考 https://github.com/oracle/graal/issues/8022 。

类型：INTERVAL

用例：

- `<INTERVAL>P=t_order_;SP=yyyy_MMdd;DIA=1;DIU=Days;DL=2023_1202;DU=2023_1204` 将被转化为 `t_order_2023_1202, t_order_2023_1203, t_order_2023_1204`
- `<INTERVAL>P=t_order_;SP=yyyy_MM;DIA=1;DIU=Months;DL=2023_10;DU=2023_12` 将被转化为 `t_order_2023_10, t_order_2023_11, t_order_2023_12`

- `<INTERVAL>P=t_order_;SP=yyyy;DIA=1;DIU=Years;DL=2021;DU=2023` 将被转化为 `t_order_2021, t_order_2022, t_order_2023`
- `<INTERVAL>P=t_order_;SP=HH_mm_ss_SSS;DIA=1;DIU=Millis;DL=22_48_52_131;DU=22_48_52_133` 将被转化为 `t_order_22_48_52_131, t_order_22_48_52_132, t_order_22_48_52_133`
- `<INTERVAL>P=t_order_;SP=yyyy_MM_dd_HH_mm_ss_SSS;DIA=1;DIU=Days;DL=2023_12_04_22_48_52_131;DU=2023_12_06_22_48_52_131` 将被转化为 `t_order_2023_12_04_22_48_52_131, t_order_2023_12_05_22_48_52_131, t_order_2023_12_06_22_48_52_131`
- `<INTERVAL>P=t_order_;SP=MM;DIA=1;DIU=Months;DL=10;DU=12` 将被转化为 `t_order_10, t_order_11, t_order_12`
- `<INTERVAL>P=t_order_;SP=GGGGyyyy_MM_dd;DIA=1;DIU=Days;DL=平成0001_12_05;DU=平成0001_12_06;C=Japanese` 将被转化为 `t_order_平成0001_12_05, t_order_平成0001_12_06`
- `<INTERVAL>P=t_order_;SP=GGGGyyy_MM_dd;DIA=1;DIU=Days;DL=平成001_12_05;DU=平成001_12_06;C=Japanese` 将被转化为 `t_order_平成001_12_05, t_order_平成001_12_06`
- `<INTERVAL>P=t_order_;SP=GGGGy_MM_dd;DIA=1;DIU=Days;DL=平成1_12_05;DU=平成1_12_06;C=Japanese` 将被转化为 `t_order_平成1_12_05, t_order_平成1_12_06`

基于 GraalVM Truffle 的 Espresso 实现的使用 Groovy 语法的行表达式

此为可选实现，你需要在自有项目的 `pom.xml` 主动声明如下依赖。并且请确保自有项目通过 OpenJDK 21+ 或其下游发行版编译。

由于 https://www.graalvm.org/jdk21/reference-manual/java-on-truffle/faq/#does-java-running-on-truffle-run-on-hotspot-too 的限制，当此模块在非 GraalVM Native Image 的环境中被使用时，仅在 System Property `os.arch` 为 `amd64` 的 Linux 上就绪。

Truffle 与 JDK 的向后兼容性矩阵位于 https://medium.com/graalvm/40027a59c401 。

```
<dependencies>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>shardingsphere-infra-expr-espresso</artifactId>
        <version>${shardingsphere.version}</version>
    </dependency>
    <dependency>
        <groupId>org.graalvm.polyglot</groupId>
        <artifactId>polyglot</artifactId>
        <version>24.1.0</version>
    </dependency>
    <dependency>
        <groupId>org.graalvm.polyglot</groupId>
```

```
        <artifactId>java</artifactId>
        <version>24.1.0</version>
        <type>pom</type>
    </dependency>
</dependencies>
```

ESPRESSO 仍为实验性模块，其允许在 GraalVM Native Image 下通过 GraalVM Truffle 的 Espresso 实现来使用带 Groovy 语法的行表达式。

语法部分与 GROOVY 实现规则相同。

类型：ESPRESSO

用例：

- `<ESPRESSO>t_order_${1..3}` 将被转化为 `t_order_1, t_order_2, t_order_3`
- `<ESPRESSO>${['online', 'offline']}_table${1..3}` 将被转化为 `online_table1, online_table2, online_table3, offline_table1, offline_table2, offline_table3`

自定义实现

用户总是可以自行创建 `org.apache.shardingsphere.infra.expr.spi.InlineExpressionParser` 的实现类，以覆盖更复杂的场景，包括连接至远程 ElasticSearch 集群执行 ES|QL 以获得 `java.util.List<String>`。

考虑一个简单的 SPI 实现类，

```
package org.example;
import org.apache.shardingsphere.infra.expr.spi.InlineExpressionParser;
import java.util.Arrays;
import java.util.List;
import java.util.Properties;
public final class CustomInlineExpressionParserFixture implements
InlineExpressionParser {
    private String inlineExpression;
    @Override
    public void init(final Properties props) {
        inlineExpression = props.getProperty(INLINE_EXPRESSION_KEY);
    }
    @Override
    public List<String> splitAndEvaluate() {
        if ("spring".equals(inlineExpression)) {
            return Arrays.asList("t_order_2024_01", "t_order_2024_02");
        }
        return Arrays.asList("t_order_2024_03", "t_order_2024_04");
    }
    @Override
    public Object getType() {
```

```
        return "CUSTOM.FIXTURE";
    }
}
```

并在项目的 classpath 添加 `META-INF/services/org.apache.shardingsphere.infra.expr.spi.InlineExpressionParser` 文件，

```
org.example.CustomInlineExpressionParserFixture
```

此时对于 ShardingSphere 配置文件中的 `actualDataNodes`，1. 若配置为 `<CUSTOM.FIXTURE>spring`，将被转化为 `t_order_2024_01`，`t_order_2024_02`。2. 若配置为 `<CUSTOM.FIXTURE>summer`，将被转化为 `t_order_2024_03`，`t_order_2024_04`。

操作步骤

使用需要使用 行表达式的属性时，如在 数据分片功能中，在 `actualDataNodes` 属性下指明特定的 SPI 实现的 Type Name 即可。

若 行表达式不指明 SPI 的 Type Name，默认将使用 `GROOVY` 的 SPI 实现。

配置示例

```
rules:
- !SHARDING
  tables:
    t_order:
      actualDataNodes: <LITERAL>ds_0.t_order_0, ds_0.t_order_1, ds_1.t_order_0, ds_
1.t_order_1
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: t_order_inline
      keyGenerateStrategy:
        column: order_id
        keyGeneratorName: snowflake
  defaultDatabaseStrategy:
    standard:
      shardingColumn: user_id
      shardingAlgorithmName: database_inline
  shardingAlgorithms:
    database_inline:
      type: INLINE
      props:
        algorithm-expression: <GROOVY>ds_${user_id % 2}
    t_order_inline:
      type: INLINE
      props:
```

```
      algorithm-expression: t_order_${order_id % 2}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
```

相关参考

- 核心概念
- 数据分片

9.3.3 SQL Hint

背景信息

目前，主流的关系型数据库基本都提供了 SQL Hint 作为 SQL 语法的补充，SQL Hint 允许用户通过数据库内置的 Hint 语法来干预 SQL 的执行过程，从而完成一些特殊的功能，或实现对 SQL 执行的优化。ShardingSphere 同样提供了丰富的 SQL Hint 语法，允许用户进行数据分片、读写分离的强制路由以及数据源透传等灵活控制。

使用规范

ShardingSphere 的 SQL Hint 语法需要以注释的形式编写在 SQL 中，SQL Hint 语法格式暂时只支持 `/* */`，Hint 内容需要以 `SHARDINGSPHERE_HINT:` 为起始，然后定义不同功能所对应的属性键值对，当存在多个属性时使用逗号分隔。ShardingSphere 的 SQL Hint 语法格式如下：

```
/* SHARDINGSPHERE_HINT: {key} = {value}, {key} = {value} */ SELECT * FROM t_order;
```

如果使用 MySQL 客户端连接，需要添加 `-c` 选项保留注释，客户端默认是 `--skip-comments` 过滤注释。

参数解释

ShardingSphere SQL Hint 中可以定义如下的属性，为了兼容低版本 SQL Hint 语法，也可以使用别名中定义的属性：

SQL Hint

数据分片

数据分片 SQL Hint 功能的可选属性包括：

- `{table}.SHARDING_DATABASE_VALUE`：用于添加 `{table}` 表对应的数据源分片键值，多个属性使用逗号分隔；

- `{table}.SHARDING_TABLE_VALUE`：用于添加 `{table}` 表对应的表分片键值，多个属性使用逗号分隔。

  分库不分表情况下，强制路由至某一个分库时，可使用 `SHARDING_DATABASE_VALUE` 方式设置分片，无需指定 `{table}`。

数据分片 SQL Hint 功能的使用示例：

```
/* SHARDINGSPHERE_HINT: t_order.SHARDING_DATABASE_VALUE=1, t_order.SHARDING_TABLE_
VALUE=1 */ SELECT * FROM t_order;
```

读写分离

读写分离 SQL Hint 功能的可选属性为 `WRITE_ROUTE_ONLY`，`true` 表示将当前 SQL 强制路由到主库执行。

读写分离 SQL Hint 功能的使用示例：

```
/* SHARDINGSPHERE_HINT: WRITE_ROUTE_ONLY=true */ SELECT * FROM t_order;
```

数据源透传

数据源透传 SQL Hint 功能可选属性为 `DATA_SOURCE_NAME`，需要指定注册在 ShardingSphere 逻辑库中的数据源名称。

数据源透传 SQL Hint 功能的使用示例：

```
/* SHARDINGSPHERE_HINT: DATA_SOURCE_NAME=ds_0 */ SELECT * FROM t_order;
```

跳过 SQL 改写

跳过 SQL 改写 SQL Hint 功能可选属性为 `SKIP_SQL_REWRITE`，`true` 表示跳过当前 SQL 的改写阶段。

跳过 SQL 改写 SQL Hint 功能的使用示例：

```
/* SHARDINGSPHERE_HINT: SKIP_SQL_REWRITE=true */ SELECT * FROM t_order;
```

跳过 SQL 执行元数据校验

跳过 SQL 执行元数据校验 SQL Hint 功能可选属性为 `SKIP_METADATA_VALIDATE`，`true` 表示跳过当前 SQL 执行的元数据校验。

跳过 SQL 执行元数据校验 SQL Hint 功能的使用示例：

```
/* SHARDINGSPHERE_HINT: SKIP_METADATA_VALIDATE=true */ SELECT * FROM t_order;
```

禁用 SQL 审计

禁用 SQL 审计 SQL Hint 功能可选属性为 `DISABLE_AUDIT_NAMES`，需要指定需要禁用的 SQL 审计算法名称，多个 SQL 审计算法需要使用逗号分隔。

禁用 SQL 审计 SQL Hint 功能的使用示例：

```
/* SHARDINGSPHERE_HINT: DISABLE_AUDIT_NAMES=sharding_key_required_auditor */ SELECT
* FROM t_order;
```

影子库压测

影子库压测 SQL Hint 功能可选属性为 `SHADOW`，`true` 表示将当前 SQL 路由至影子库数据源执行。

影子库压测 SQL Hint 功能的使用示例：

```
/* SHARDINGSPHERE_HINT: SHADOW=true */ SELECT * FROM t_order;
```

9.4 错误码

本章列举 Apache ShardingSphere 错误码。包含 SQL 错误码和服务器错误码。

本章节所有内容均为草稿，错误码仍可能调整。

9.4.1 SQL 错误码

SQL 错误码以标准的 SQL State，Vendor Code 和详细错误信息提供，在 SQL 执行错误时返回给客户端。

目前内容为草稿，错误码仍可能调整。

内核异常

元数据

Vendor Code	SQL State	错误信息
10000	42S02	Database is required.
10001	42S02	Schema ‘%s’ does not exist.
10002	42S02	Table or view ‘%s’ does not exist.
10003	42S02	Unknown column ‘%s’ in ‘%s’.
10004	42S02	Index ‘%s’ does not exist.
10005	42S01	Index ‘%s’ already exists.
10010	HY000	Rule and storage meta data mismatched, reason is: %s.
10012	HY000	Load table meta data failed for database ‘%s’ and tables ‘%s’.

续下页

表 1 – 接上页

V endor Code	SQL S tate	错误信息
10100	H Y000	Can not %s storage units ‘%s’.
10101	4 2S02	There is no storage unit in database ‘%s’.
10102	4 4000	Storage units ‘%s’ do not exist in database ‘%s’.
10103	4 4000	Storage unit ‘%s’ still used by ‘%s’.
10104	4 2S01	Duplicate storage unit names ‘%s’.
10110	0 8000	Storage units can not connect, error messages are: %s.
10111	0 A000	Can not alter connection info in storage units: ‘%s’.
10120	4 4000	Invalid storage unit status, error message is: %s.
10200	4 4000	Invalid ‘%s’ rule ‘%s’, error message is: %s
10201	4 2S02	There is no rule in database ‘%s’.
10202	4 2S02	%s rules ‘%s’ do not exist in database ‘%s’.
10203	4 4000	%s rules ‘%s’ in database ‘%s’ are still in used.
10204	4 2S01	Duplicate %s rule names ‘%s’ in database ‘%s’.
10210	4 2S02	%s strategies ‘%s’ do not exist.
10300	H Y000	Invalid format for actual data node ‘%s’.
10301	0 A000	Can not support 3-tier structure for actual data node ‘%s’ with JDBC ‘%s’.
10400	4 4000	Algorithm ‘%s’ initialization failed, reason is: %s.
10401	4 2S02	‘%s’ algorithm on %s is required.
10402	4 2S02	‘%s’ algorithm ‘%s’ on %s is unregistered.
10403	4 4000	%s algorithms ‘%s’ in database ‘%s’ are still in used.
10404	4 4000	Invalid %s algorithm configuration ‘%s’.
10410	0 A000	Unsupported %s.%s with database type ‘%s’.
10440	H Y000	Algorithm ‘%s’ execute failed, reason is: %s.
10500	4 4000	Invalid single rule configuration, reason is: %s.
10501	4 2S02	Single table ‘%s’ does not exist.
10502	H Y000	Can not load table with database name ‘%s’ and data source name ‘%s’, reason is: %
10503	0 A000	Can not drop schema ‘%s’ because of contains tables.

数据

Vendor Code	SQL State	错误信息
11000	HY004	Unsupported conversion data type ‘%s’ for value ‘%s’.
11001	HY004	Unsupported conversion stream charset ‘%s’.

语法

Vendor Code	SQL State	错误信息
12000	42000	SQL String can not be NULL or empty.
12010	44000	Can not support variable ‘%s’.
12011	HY004	Invalid variable value ‘%s’.
12020	HV008	Column index ‘%d’ is out of range.
12021	42S02	Can not find column label ‘%s’.
12022	HY000	Column ‘%s’ in %s is ambiguous.
12100	42000	You have an error in your SQL syntax: %s
12101	42000	Can not accept SQL type ‘%s’.
12200	42000	Hint data source ‘%s’ does not exist.
12300	0A000	DROP TABLE …CASCADE is not supported.
12500	42000	Not unique table/alias: ‘%s’.
12600	HY000	In definition of view, derived table or common table expression, SELECT list and column names list have different column counts.

连接

VendorCode	SQLState	错误信息
13000	08000	Can not get %d connections one time, partition succeed connection(%d) have released. Please consider increasing the ‘maxPoolSize’ of the data sources or decreasing the ‘max-connections-size-per-query’ in properties.
13001	08000	SQL execution has been interrupted.
13010	01000	Circuit break open, the request has been ignored.
13100	0A000	Unsupported storage type of URL ‘%s’.
13101	08000	The URL ‘%s’ is not recognized, please refer to the pattern ‘%s’.
13200	08000	Can not register driver.
13201	08000	Connection has been closed.
13202	08000	Result set has been closed.
13400	HY000	Load datetime from database failed, reason: %s

事务

Vendor Code	SQL State	错误信息
14000	25 000	Switch transaction type failed, please terminate the current transaction.
14001	42 S02	Can not find transaction manager of ‘%s’.
14002	44 000	Max length of unique resource name ‘%s’ exceeded, should be less than 45.
14003	25 000	Transaction timeout should more than 0.
14004	25 000	Close transaction manager failed.
14200	25 000	Failed to create ‘%s’ XA data source.
14201	25 000	Can not start new XA transaction in a active transaction.
14202	25 000	Check XA transaction privileges failed on data source, please grant ‘%s’ to current user.
14400	44 000	No application id within ‘seata.conf’ file.
14401	25 000	Seata-AT transaction has been disabled.

锁

Vendor Code	SQL State	错误信息
15030	HY000	Cluster is already locked.
15031	HY000	Cluster is not locked.

集群

Vendor Code	SQL State	错误信息
17000	44000	Mode must be ‘cluster’.
17001	HY000	Worker ID assigned failed, which should be in [0, %s).
17010	HY000	Cluster persist repository error, reason is: %s
17011	HY000	Failed to reload meta data context.
17020	HY000	The cluster status is %s, can not support SQL statement ‘%s’.
17100	42S02	Cluster persist repository configuration is required.

数据管道

Vendor Code	SQL State	错误信息
18000	22023	There is invalid parameter value ‘%s’.
18100	42S02	Target database ‘%s’ does not exist.
18101	42S02	Can not find pipeline job ‘%s’.
18102	44000	Sharding count of job ‘%s’ is 0.
18103	42S02	Can not get meta data for table ‘%s’ when split by range.
18104	HY000	Can not split by unique key ‘%s’ for table ‘%s’.
18105	HY000	Target table ‘%s’ is not empty.
18106	01007	Source data source lacks ‘%s’ privilege(s).
18107	HY000	Source data source required ‘%s = %s’, now is ‘%s’.
18108	42S02	User ‘%s’ does exist.
18109	08000	Check privileges failed on source data source.
18110	HY000	Importer job write data failed.
18111	08000	Get binlog position failed by job ‘%s’.
18112	HY000	Can not find consistency check job of ‘%s’.
18113	HY000	Uncompleted consistency check job ‘%s’ exists, progress ‘%s’.
18114	HY000	Failed to get DDL for table ‘%s’.
18200	HY000	Before data record is ‘%s’, after data record is ‘%s’.
18201	08000	Data check table ‘%s’ failed.
18202	0A000	Unsupported pipeline database type ‘%s’.
18400	42S02	Can not find stream data source table.
18401	42S02	Database ‘%s’ does not exist.
18410	42S02	CDC Login request body is empty.
18411	08004	Illegal username or password.

功能异常

数据分片

Ve nd or Co de	S QL S ta te	错误信息
2 00 00	4 2S 02	%s configuration does not exist in database ‘%s’.
2 00 01	4 2S 02	Can not find table rule with logic tables ‘%s’.
2 00 02	4 2S 02	Can not find data source in sharding rule, invalid actual data node ‘%s’.
2 00 03	4 2S 02	Data nodes is required for sharding table ‘%s’.
2 00 04	4 2S 02	Actual table ‘%s.%s’ is not in table rule configuration.
2 00 05	4 2S 02	Can not find binding actual table, data source is ‘%s’, logic table is ‘%s’, other actu
2 00 06	4 40 00	Actual tables ‘%s’ are in use.
2 00 09	4 2S 01	View name has to bind to %s tables.

表 2 – 接上页

Ve nd or Co de	S QL S ta te	错误信息
2 00 10	4 40 00	Invalid binding table configuration.
2 00 11	4 40 00	Only allowed 0 or 1 sharding strategy configuration.
2 00 12	4 2S 01	Same actual data node cannot be configured in multiple logic tables in same database, l
2 00 20	4 40 00	Sharding value can not be null in SQL statement.
2 00 21	H Y0 04	Found different types for sharding value ‘%s’.
2 00 22	H Y0 04	Invalid %s, datetime pattern should be ‘%s’, value is ‘%s’.
2 00 23	4 40 00	Sharding value %s subtract stop offset %d can not be less than start offset %d.
2 00 24	4 40 00	%s value ‘%s’ must implements Comparable.
2 00 30	0 A0 00	Can not support operation ‘%s’ with sharding table ‘%s’.
2 00 31	4 40 00	Can not update sharding value for table ‘%s’.
2 00 32	0 A0 00	The CREATE VIEW statement contains unsupported query statement.
2 00 33	4 40 00	PREPARE statement can not support sharding tables route to same data sources.
2 00 34	4 40 00	The table inserted and the table selected must be the same or bind tables.
2 00 35	0 A0 00	Can not support DML operation with multiple tables ‘%s’.
2 00 36	4 20 00	%s …LIMIT can not support route to multiple data nodes.
2 00 37	4 40 00	Can not find actual data source intersection for logic tables ‘%s’.
2 00 38	4 20 00	INSERT INTO …SELECT can not support applying key generator with absent generate k
2 00 39	0 A0 00	Alter view rename .. to .. statement should have same config for ‘%s’ and ‘%s’.
2 00 40	H Y0 00	‘%s %s’ can not route correctly for %s ‘%s’.
2 00 41	4 2S 02	Can not get route result, please check your sharding rule configuration.
2 00 42	3 40 00	Can not get cursor name from fetch statement.
2 00 50	H Y0 00	Sharding algorithm class ‘%s’ should be implement ‘%s’.
2 00 51	H Y0 00	Routed target ‘%s’ does not exist, available targets are ‘%s’.
2 00 52	4 40 00	Inline sharding algorithms expression ‘%s’ and sharding column ‘%s’ do not matc
2 00 53	4 40 00	Complex inline algorithm need %d sharding columns, but only found %d.
2 00 54	4 40 00	No sharding database route info.
2 00 55	4 40 00	Some routed data sources do not belong to configured data sources. routed data sources
2 00 56	4 40 00	Please check your sharding conditions ‘%s’ to avoid same record in table ‘%s’ rout
2 00 57	4 40 00	Can not find routing table factor, data source ‘%s’, actual table ‘%s’.
2 00 60	H Y0 00	Invalid %s strategy ‘%s’, strategy does not match data nodes.
2 00 90	4 20 00	Not allow DML operation without sharding conditions.

联邦查询

Vendor Code	SQL State	错误信息
20100	42000	Unsupported SQL node conversion for SQL statement ‘%s’.
20101	42000	SQL federation does not support SQL ‘%s’.
20102	42S02	SQL federation schema ‘%s’ not found in SQL ‘%s’.

读写分离

Vendor Code	SQL State	错误信息
20200	42S02	Readwrite-splitting data source rule name is required in database ‘%s’.
20201	42S02	Can not find readwrite-splitting data source rule ‘%s’ in database ‘%s’.
20202	42S02	Readwrite-splitting [READ/WRITE] data source is required in %s.
20203	42S02	Can not find readwrite-splitting [READ/WRITE] data source ‘%s’ in %s.
20204	42S01	Readwrite-splitting [READ/WRITE] data source ‘%s’ is duplicated in %s.
20205	44000	Readwrite-splitting [READ/WRITE] data source inline expression error in %s.

SQL 方言转换

Vendor Code	SQL State	错误信息
20400	0A000	Can not support database ‘%s’ in SQL translation.

流量治理

Vendor Code	SQL State	错误信息
20500	42S02	Can not get traffic execution unit.

数据加密

Vendor Code	SQL State	错误信息
21000	42S02	%s column is required in %s.
21001	42S02	Can not find encrypt table ‘%s’.
21002	42S02	Can not find logic encrypt column by ‘%s’.
21003	42S02	Can not find encrypt column ‘%s’ from table ‘%s’.
21004	HY000	‘%s’ column’s encrypt algorithm ‘%s’ should support %s in database ‘%s’.
21005	HY000	Column ‘%s’ of table ‘%s’ is not configured with %s query algorithm.
21010	44000	Altered column ‘%s’ must use same encrypt algorithm with previous column ‘%s’ in table ‘%s’.
21020	0A000	The SQL clause ‘%s’ is unsupported in encrypt feature.
21030	22000	Failed to decrypt the ciphertext ‘%s’ in the column ‘%s’ of table ‘%s’.

影子库

Vendor Code	SQL State	错误信息
22000	42S02	Production data source configuration does not exist in database ‘%s’.
22001	42S02	Shadow data source configuration does not exist in database ‘%s’.
22002	42S02	No available shadow data sources mappings in shadow table ‘%s’.
22003	44000	Default shadow algorithm class should be implement HintShadowAlgorithm.
22010	HY004	Shadow column ‘%s’ of table ‘%s’ does not support ‘%s’ type.
22020	42000	Insert value of index ‘%d’ can not support for shadow.

其他异常

Vendor Code	SQL State	错误信息
30000	HY000	Unknown exception: %s
30001	0A000	Unsupported SQL operation: %s
30002	HY000	Database protocol exception: %s
30003	0A000	Unsupported command: %s
30004	HY000	Server exception: %s
30005	HY000	Underlying SQL state: %s, underlying error code: %s.
30010	HY000	Can not find plugin class ‘%s’.
30020	HY000	File access failed, file is: %s
30030	HY000	Unexpected tableless route engine.

9.4.2 服务器错误码

服务器发生错误时所提供的唯一错误码，打印在 Proxy 后端或 JDBC 启动日志中。

错误码	错误信息
SPI-00001	No implementation class load from SPI ‘%s’ with type ‘%s’.
DATA-SOURCE-00001	Data source ‘%s’ is unavailable.
PROPS-00001	Properties convert failed, details are: %s.
PROXY-00001	Load database server info failed.

10 开发者手册

Apache ShardingSphere 可插拔架构提供了数十个基于 SPI 的扩展点。对于开发者来说，可以十分方便的对功能进行定制化扩展。

本章节将 Apache ShardingSphere 的 SPI 扩展点悉数列出。如无特殊需求，用户可以使用 Apache ShardingSphere 提供的内置实现；高级用户则可以参考各个功能模块的接口进行自定义实现。

Apache ShardingSphere 社区非常欢迎开发者将自己的实现类反馈至开源社区，让更多用户从中收益。

10.1 运行模式

10.1.1 StandalonePersistRepository

全限定类名

`org.apache.shardingsphere.mode.repository.standalone.StandalonePersistRepository <https://github.com/apache/shardingsphere/blob/master/mode/type/standalone/repository/api/src/main/java/org/apache/shardingsphere/mode/repository/standalone/StandalonePersistRepository.java>‘__

定义

单机模式配置信息持久化定义

已知实现

10.1.2 ClusterPersistRepository

全限定类名

`org.apache.shardingsphere.mode.repository.cluster.ClusterPersistRepository <https://github.com/apache/shardingsphere/blob/master/mode/type/cluster/repository/api/src/main/java/org/apache/shardingsphere/mode/repository/cluster/ClusterPersistRepository.java>‘__

定义

集群模式配置信息持久化定义

已知实现

10.2 SQL 解析

10.2.1 DatabaseTypedSQLParserFacade

全限定类名

`org.apache.shardingsphere.sql.parser.spi.DialectSQLParserFacade <https://github.com/apache/shardingsphere/blob/master/parser/sql/spi/src/main/java/org/apache/shardingsphere/sql/parser/spi/DialectSQLParserFacade.java>‘__

定义

配置用于 SQL 解析的词法分析器和语法分析器入口

已知实现

• 详 | * 全限定类名 配 | 细 | 置 | 说 | 标 | 明 * | 识 | | * | |		
M y S Q L	基于 My SQL 的 SQL 解析器入口	`org .apache. shardingsphere.sql. parser.mysql.parser. MySQLParserF acade <https://github.com/apache/shardingsphere/blob/master/parser/sql/dialect/mysql/src/main/java/org/apa ng-sphere/sql/parser/mysql/parser/MySQLParserFac
P o s t g r e S Q L	基于 P ost gre SQL 的 SQL 解析器入口	`org.apache. shardingsphere. sq l.parser. postgresql.parser. PostgreSQLParserFacade <https://github.com/apache/shardingsphere/blob/maste ialect/postgresql/src/main/java/org/apache/shard ql/parser/postgresql/parser/PostgreSQLParserFac
o p e n G a u s s	基于 ope nGa uss 的 SQL 解析器入口	`org.apache. shardingsphe re.sql.parser. opengauss.parser. OpenGaussParserFacade <htt ps://github.com/apache/shardingsphere/blob/ma l/dialect/opengauss/src/main/java/org/apache/sha /sql/parser/opengauss/parser/OpenGaussParserF
O r a c l e	基于 Ora cle 的 SQL 解析器入口	`org.apac he. shardingsphere.sql. parser.oracle.parser. OracleParserFaca de <https://github.com/apache/shardingsphere/blob/master/parser/sql/dialect/oracle/src/main/java/org/apach sphere/sql/parser/oracle/parser/OracleParserFac
S Q L S e r v e r	基于 SQL Ser ver 的 SQL 解析器入口	`org.apache. shardingsphe re.sql.parser. sqlserver.parser. SQLServerParserFacade <htt ps://github.com/apache/shardingsphere/blob/ma l/dialect/sqlserver/src/main/java/org/apache/shar /sql/parser/sqlserver/parser/SQLServerParserFac
C l i c k H o u s e	基于 C lick H ous e 的 SQL 解析	`org.apache.

10.2.2 SQLStatementVisitorFacade

全限定类名

`org.apache.shardingsphere.sql.parser.spi.SQLStatementVisitorFacade <https://github.com/apache/shardingsphere/blob/master/parser/sql/spi/src/main/java/org/apache/shardingsphere/sql/parser/spi/SQLStatementVisitorFacade.java>‘__

定义

SQL 语法树访问器入口

已知实现

• 详 | * 全限定类名 配 | 细 | 置 | 说 | 标 | 明 * | 识 | | * | |		
M y S Q L	基于 My SQL 的 SQL 语法树访问器入口	`org.apache. shardingsphere. sql.parser.mysq l. visitor.statement. MySQLStatementVisitorFacade <h t t p s : / / g ithub.com/apache/shardingsphere/blob/master/p ct/mysql/src/main/java/org/apache/shardingsphe mysql/visitor/statement/MySQLStatementVisitorF
P o s t g r e S Q L	基于 P ost gre SQL 的 SQL 语法树访问器入口	`org.ap ache. shardingsphere.sql. parser.postgresql. visitor.statement. PostgreSQLStatementVisitorFacade <https://github.com/apach e/shardingsphere/blob/master/parser/sql/dialect/ rc/main/java/org/apache/shardingsphere/sql/pars /visi- tor/statement/PostgreSQLStatementVisitorFacade
S Q L S e r v e r	基于 SQL Ser ver 的 SQL 语法树访问器入口	`o rg.apache. shardingsphere.sql. parser.sqlserver. visitor.statem ent. SQLServerStatementVisitorFacade <h t t p s : / / g i t h u b . c o m / a p ache/shardingsphere/blob/master/parser/sql/dial /src/main/java/org/apache/shardingsphere/sql/pa r/visitor/statement/SQLServerStatementVisitorFa
O r a c l e	基于 Ora cle 的 SQL 语法树访问器入口	`org.apache. shardingsphere.sql. parser.oracle.vi sitor.statement. OracleStatementVisitorFacade <h t t p s : / / g i th ub.com/apache/shardingsphere/blob/master/pars ora- cle/src/main/java/org/apache/shardingsphere/sql acle/visitor/statement/OracleStatementVisitorFac
S Q L 9 2	基于 SQ L92 的 SQL 语法树访问器入口	`org.apache. shardingsphere. sql.parser.sql9 2. visitor.statement. SQL92StatementVisitorFacade <h t t p s : / / g

10.3 数据分片

10.3.1 ShardingAlgorithm

全限定类名

`org.apache.shardingsphere.sharding.spi.ShardingAlgorithm <https://github.com/apache/shardingsphere/blob/master/features/sharding/api/src/main/java/org/apache/shardingsphere/sharding/spi/ShardingAlgorithm.java>‘__

定义

分片算法

已知实现

10.3.2 ShardingAuditAlgorithm

全限定类名

`org.apache.shardingsphere.sharding.spi.ShardingAuditAlgorithm <https://github.com/apache/shardingsphere/blob/master/features/sharding/api/src/main/java/org/apache/shardingsphere/sharding/spi/ShardingAuditAlgorithm.java>‘__

定义

分片审计算法

已知实现

10.3.3 DatetimeService

全限定类名

`org.apache.shardingsphere.timeservice.spi.TimestampService <https://github.com/apache/shardingsphere/blob/master/kernel/time-service/api/src/main/java/org/apache/shardingsphere/timeservice/spi/TimestampService.java>‘__

定义

获取当前时间进行路由

已知实现

10.3.4 InlineExpressionParser

全限定类名

`org.apache.shardingsphere.infra.expr.core.InlineExpressionParser`

定义

解析行表达式

已知实现

配置标识 *	详细说明	全限定类名
GR OO VY	使用 Groovy 语法的行表达式	“org.apache.shardingsphere.infra.expr.groovy.GroovyInlineExpressionParser“
L IT ER AL	使用标准列表的行表达式	`or g.apache.shardingsphere.infra.expr.li teral.LiteralInlineExpressionParser`
IN TE RV AL	基于固定时间范围的 Key-Value 语法的行表达式	`org. apache.shardingsphere.infra.expr.inte rval.IntervalInlineExpressionParser`
ES PR ES SO	基于 GraalVM Truffle 的 Espresso 实现的使用 Groovy 语法的行表达式	`org. apache.shardingsphere.infra.expr.espr esso.EspressoInlineExpressionParser`

10.4 基础算法

10.4.1 LoadBalanceAlgorithm

全限定类名

`org.apache.shardingsphere.infra.algorithm.loadbalancer.core.LoadBalanceAlgorithm <https://github.com/apache/shardingsphere/blob/master/infra/algorithm/type/load-balancer/core/src/main/java/org/apache/shardingsphere/infra/algorithm/loadbalancer/core/LoadBalanceAlgorithm.java>‘__

定义

负载均衡算法，可以使用在读写分离、JDBC 双路由等功能中。

已知实现

10.4.2 KeyGenerateAlgorithm

全限定类名

`org.apache.shardingsphere.keygen.core.algorithm.KeyGenerateAlgorithm <https://github.com/apache/shardingsphere/blob/master/infra/algorithm/type/key-generator/core/src/main/java/org/apache/shardingsphere/infra/algorithm/keygen/core/KeyGenerateAlgorithm.java>‘__

定义

分布式主键生成算法，可以使用在数据分片功能中。

已知实现

• 详 | * 全限定类名 配 | 细 | 置 | 说 | 标 | 明 * | 识 | | * | |		
SNOWFLAKE	基于雪花算法的分布式主键生成算法	`org.apache . shardingsphere.keygen. snowflake.algorithm. SnowflakeKeyGen erateAlgorithm <https://github.com/a pache/shardingsphere /blob/master/infra/algorithm/type/key-generator/type/snowf lake/src/main/java/org/apache/shardingsphere/in hm/keygen/snowflake/SnowflakeKeyGenerateAlg
UUID	基于 U UID 的分布式主键生成算法	`org.apache. shardingsphere.keygen. uuid.algor ithm. UUIDKeyGenerateAlgorithm <https://github.com/apache /shardingsphere/blob/master/infra/algorithm/typ gener ator/type/uuid/src/main/java/org/apache/shardin ra/algorithm/keygen/uuid/UUIDKeyGenerateAlgo

10.4.3 MessageDigestAlgorithm

全限定类名

`org.apache.shardingsphere.infra.algorithm.messagedigest.core. MessageDigestAlgorithm <https://github.com/apache/shardingsphere/blob/master/infra /algorithm/type/message-digest/core/src/main/java/org/apache/shardingsphere/infra/algorithm/mes sagedigest/core/MessageDigestAlgorithm.java>‘__

定义

消息摘要算法，可以使用在数据脱敏、数据加密功能中。

已知实现

10.5 SQL 审计

10.5.1 SQLAuditor

全限定类名

`org.apache.shardingsphere.infra.executor.audit.SQLAuditor <https://github.com/apache/shardingsphere/blob/master/infra/executor/src/main/java/org/apache/shardingsphere/infra/executor/audit/SQLAuditor.java>‘__

定义

SQL 审计定义接口

已知实现

10.6 数据加密

10.6.1 EncryptAlgorithm

全限定类名

`org.apache.shardingsphere.encrypt.spi.EncryptAlgorithm <https://github.com/apache/shardingsphere/blob/master/features/encrypt/api/src/main/java/org/apache/shardingsphere/encrypt/spi/EncryptAlgorithm.java>‘__

定义

数据加密算法

已知实现

10.7 数据脱敏

10.7.1 MaskAlgorithm

全限定类名

`org.apache.shardingsphere.mask.spi.MaskAlgorithm <https://github.com/apache/shardingsphere/blob/master/features/mask/api/src/main/java/org/apache/shardingsphere/mask/spi/MaskAlgorithm.java>‘__

定义

数据脱敏算法

已知实现

10.8 影子库

10.8.1 ShadowAlgorithm

全限定类名

`org.apache.shardingsphere.shadow.spi.ShadowAlgorithm <https://github.com/apache/shardingsphere/blob/master/features/shadow/api/src/main/java/org/apache/shardingsphere/shadow/spi/ShadowAlgorithm.java>‘__

定义

影子库提供的影子算法

已知实现

10.9 可观察性

10.9.1 PluginLifecycleService

全限定类名

`org.apache.shardingsphere.agent.spi.PluginLifecycleService <https://github.com/apache/shardingsphere/blob/master/agent/api/src/main/java/org/apache/shardingsphere/agent/spi/PluginLifecycleService.java>‘__

定义

插件生命周期管理接口

已知实现

11 测试手册

Apache ShardingSphere 提供了完善的整合测试、模块测试和性能测试。

11.1 整合测试

通过真实的 Apache ShardingSphere 和数据库的连接，提供端到端的测试。

整合测试引擎以 XML 方式定义 SQL，分别为各个数据库独立运行测试用例。为了方便上手，测试引擎无需修改任何 **Java** 代码，只需修改相应的配置文件即可运行断言。测试引擎不依赖于任何第三方环境，用于测试的 ShardingSphere-Proxy 计算节点和数据库均由 Docker 镜像提供。

11.2 模块测试

将复杂的模块单独提炼成为测试引擎。

模块测试引擎同样以 XML 方式定义 SQL，分别为各个数据库独立运行测试用例，包括 SQL 解析和 SQL 改写模块。

11.3 性能测试

提供多样性的性能测试方法，包括 Sysbench、JMH、TPCC 等。

11.4 集成测试

11.4.1 设计

集成测试包括 3 个模块：测试用例、测试环境以及测试引擎。

测试用例

用于定义待测试的 SQL 以及测试结果的断言数据。每个用例定义一条 SQL，SQL 可定义多种数据库执行类型。

测试环境

用于搭建运行测试用例的数据库和 ShardingSphere-Proxy 环境。环境又具体分为环境准备方式，数据库类型和场景。

环境准备方式分为 Native 和 Docker，未来还将增加 Embed 类型的支持。

- Native 环境用于测试用例直接运行在开发者提供的测试环境中，适用于调试场景。
- Docker 环境由 Testcontainer 创建，适用于云编译环境和测试 ShardingSphere-Proxy 的场景，如：GitHub Action。
- Embed 环境由测试框架自动搭建嵌入式 MySQL，适用于 ShardingSphere-JDBC 的本地环境测试。

当前默认采用 Docker 环境，使用 Testcontainer 创建运行时环境并执行测试用例。

数据库类型目前支持 MySQL、PostgreSQL、SQLServer 和 Oracle，并且可以支持使用 ShardingSphere-JDBC 或是使用 ShardingSphere-Proxy 执行测试用例。

场景用于对 ShardingSphere 支持规则进行测试，目前支持数据分片、数据加密、数据脱敏和读写分离的相关场景，未来会不断完善场景的组合。

测试引擎

用于批量读取测试用例，并逐条执行和断言测试结果。

测试引擎通过将用例和环境进行排列组合，以达到用最少的用例测试尽可能多场景的目的。

每条 SQL 会以 `数据库类型 * 接入端类型 * SQL 执行模式 * JDBC 执行模式 * 场景`的组合方式生成测试报告，目前各个维度的支持情况如下：

- 数据库类型：H2、MySQL、PostgreSQL、SQLServer 和 Oracle；
- 接入端类型：ShardingSphere-JDBC 和 ShardingSphere-Proxy；
- SQL 执行模式：Statement 和 PreparedStatement；
- JDBC 执行模式：execute 和 executeQuery（查询）/ executeUpdate（更新）；
- 场景：分库、分表、读写分离和分库分表 + 读写分离。

因此，1 条 SQL 会驱动：`数据库类型（5）* 接入端类型（2）* SQL 执行模式（2）* JDBC 执行模式（2）* 场景（4）= 160` 个测试用例运行，以达到项目对于高质量的追求。

11.4.2 使用指南

模块路径：test/e2e/sql

测试用例配置

SQL 用例在 resources/cases/${SQL-TYPE}/e2e-${SQL-TYPE}-${cases-description}.xml。

用例文件格式如下：

```
<e2e-test-cases>
    <test-case sql="${SQL}">
        <!-- 查询场景 -->
        <assertion parameters="${value_1}:${type_1}, ${value_2}:${type_2}"
expected-data-source-name="{datasource-name}" />
        <!-- 非查询场景 -->
        <assertion parameters="${value_1}:${type_1}, ${value_2}:${type_2}"
expected-data-file="${dataset_file_1}.xml" />
        <!-- ... more assertions -->
    </test-case>

    <test-case sql="${SQL}">
        <assertion parameters="${value_1}:${type_1}, ${value_2}:${type_2}"
expected-data-file="${dataset_file_1}.xml" />
        <!-- ... more assertions -->
        <assertion parameters="${value_3}:${type_3}, ${value_4}:${type_4}"
expected-data-file="${dataset_file_2}.xml" />
    </test-case>

    <!-- ... more test cases -->
</e2e-test-cases>
```

expected-data-file 的查找规则是：1. 查找同级目录中 dataset\${SCENARIO_NAME}\${DATABASE_TYPE}\${dataset_file}.xml 文件；2. 查找同级目录中 dataset\${SCENARIO_NAME}\${dataset_file}.xml 文件；3. 查找同级目录中 dataset\${dataset_file}.xml 文件；4. 都找不到则报错。

断言文件格式如下：

```
<dataset>
    <metadata>
        <column name="column_1" />
        <!-- ... more columns -->
        <column name="column_n" />
    </metadata>
    <row values="value_01, value_02" />
    <!-- ... more rows -->
```

```
    <row values="value_n1, value_n2" />
</dataset>
```

环境配置

${SCENARIO-TYPE} 表示场景名称，在测试引擎运行中用于标识唯一场景。${DATABASE-TYPE} 表示数据库类型。

Native 环境配置

修改 e2e-sql 模块 src/test/resources/env/it-env.properties 文件中 it.cluster.env.type 为 NATIVE 模式，然后修改如下的属性为本地数据库地址和账号。

```
it.native.storage.host=127.0.0.1
it.native.storage.port=3306
it.native.storage.username=root
it.native.storage.password=123456
```

修改完成后，可以再调整 it-env.properties 中其他属性，测试 ShardingSphere 的 Proxy、JDBC 接入端，或者测试单机、集群模式。

Docker 环境配置

修改 e2e-sql 模块 src/test/resources/env/it-env.properties 文件中 it.cluster.env.type 为 DOCKER 模式，如果执行 Proxy 接入端测试，需要执行如下的命令打包 Proxy 镜像。

```
./mvnw -B clean install -am -pl test/e2e/sql -Pit.env.docker -DskipTests -
Dspotless.apply.skip=true -Drat.skip=true
```

如果是 Mac 平台 M 系列芯片，在打包 Proxy 镜像前，需要先执行如下的命令，然后再进行 Proxy 镜像打包。

```
# 安装 socat
brew install socat
socat TCP-LISTEN:2375,reuseaddr,fork UNIX-CLIENT:/var/run/docker.sock

# 在创建镜像的窗口中执行
export DOCKER_HOST=tcp://127.0.0.1:2375
```

修改完成后，可以再调整 it-env.properties 中其他属性，测试 ShardingSphere 的 Proxy、JDBC 接入端，或者测试单机、集群模式。

Docker 环境配置为 ShardingSphere-Proxy 提供了远程调试端口，可以在 “test/e2e/fixture/src/test/assembly/bin/start.sh“ 文件的 “JAVA_OPTS“ 中找到第 2 个暴露的端口用于远程调试。

运行测试引擎

配置测试引擎运行环境

通过配置 `src/test/resources/env/it-env.properties` 控制测试引擎。

所有的属性值都可以通过 Maven 命令行 `-D` 的方式动态注入。

```
# 运行模式，多个值可用逗号分隔。可选值：Standalone，Cluster
it.run.modes=Cluster

# 场景类型，多个值可用逗号分隔。可选值：db，tbl，dbtbl_with_replica_query，replica_query
it.scenarios=db,tbl,dbtbl_with_replica_query,replica_query

# 是否运行附加测试用例
it.run.additional.cases=false
# 是否运行冒烟测试
it.run.smoke=false

# 配置环境类型，只支持单值。可选值：DOCKER，NATIVE
it.cluster.env.type=${it.env}
# 待测试的接入端类型，多个值可用逗号分隔。可选值：jdbc，proxy，默认值：jdbc
it.cluster.adapters=jdbc

# 场景类型，多个值可用逗号分隔。可选值：H2，MySQL，PostgreSQL，openGauss
it.cluster.databases=H2,MySQL,PostgreSQL,openGauss

# 测试数据库的镜像版本
it.cluster.database.mysql.image=mysql:8.2.0

# NATIVE 模式下数据库连接信息及账号
it.native.storage.host=127.0.0.1
it.native.storage.port=3306
it.native.storage.username=root
it.native.storage.password=123456
```

运行调试模式

- 标准测试引擎运行 `org.apache.shardingsphere.test.e2e.it.sql.${SQL-TYPE}.General${SQL-TYPE}E2EIT` 以启动不同 SQL 类型的测试引擎。
- 批量测试引擎运行 `org.apache.shardingsphere.test.e2e.it.sql.dml.BatchDMLE2EIT`，以启动为 DML 语句提供的测试 `addBatch()` 的批量测试引擎。
- 附加测试引擎运行 `org.apache.shardingsphere.test.e2e.it.sql.${SQL-TYPE}.Additional${SQL-TYPE}E2EIT` 以启动使用更多 JDBC 方法调用的测试引擎。附加测试引擎需要通过设置 `it.run.additional.cases=true` 开启。

运行 Docker 模式

```
./mvnw -B clean install -f test/e2e/pom.xml -Pit.env.docker -Dit.cluster.
adapters=proxy,jdbc -Dit.scenarios=${scenario_name_1,scenario_name_2,scenario_name_
n} -Dit.cluster.databases=MySQL
```

运行以上命令会构建出一个用于集成测试的 Docker 镜像 `apache/shardingsphere-proxy-test:latest`。如果仅修改了测试代码，可以复用已有的测试镜像，无须重新构建。使用以下命令可以跳过镜像构建，直接运行集成测试：

```
./mvnw -B clean install -f test/e2e/sql/pom.xml -Pit.env.docker -Dit.cluster.
adapters=proxy,jdbc -Dit.scenarios=${scenario_name_1,scenario_name_2,scenario_name_
n} -Dit.cluster.databases=MySQL
```

远程 debug Docker 容器中的 Proxy 代码

首先修改要测试模块的配置文件 it-env.properties，将 function.it.env.type 设置为 `docker`；设置对应的数据库镜像版本，例如 `transaction.it.docker.mysql.version=mysql:5.7`。其次通过命令生成测试镜像，例如：

```
# for operation, replace ${operation} with transaction、pipeline or showprocesslist
./mvnw -B clean install -am -pl test/e2e/operation/${operation} -Pit.env.docker -
DskipTests

# for e2e sql
./mvnw -B clean install -am -pl test/e2e/sql -Pit.env.docker -DskipTests -
Dspotless.apply.skip=true
```

远程调试通过镜像启动的 Proxy

E2E 测试的 Proxy 镜像默认开启了 3308 端口用于远程调试容器中的实例。使用 IDEA 等 IDE 工具可以通过如下方式连接并 debug 容器中的 Proxy 代码：

IDEA -> Run -> Edit Configurations -> Add New Configuration -> Remote JVM Debug

编辑对应的信息：- Name：一个描述性的名字，例如 e2e-debug。- Host：可以访问 docker 的 IP，例如 127.0.0.1。- Port：调试端口（需要在下一步中设置）。- use module classpath：项目根目录 shardingsphere。

编辑好上面的信息后，在 IDEA 中 Run -> Run -> e2e-debug 即可启动 IDEA 的远程 debug。

远程调试通过 Testcontainer 启动的 Proxy

> 注意：如果通过 Testcontainer 启动 Proxy 容器，由于 Testcontainer 启动前 3308 端口还没有暴露出来，无法通过 远程调试通过镜像启动的 `Proxy` 方式进行 debug。可以通过如下方式 debug Testcontainer 启动的 Proxy 容器：- 在 Testcontainer 的相关启动类后打一个断点，例如 sql 测试中 E2EContainerComposer -> `containerComposer.start();` 后面的一行打断点，此时相关容器一定已经启动。- 通过快捷键 Alt + F8，进入断点调试模式，通过命令 `docker ps` 查看 containerComposer 下的 Proxy 对象 3308 映射的端口（Testcontainer 对外映射端口是随机的）。- 参考 远程调试通过镜像启动的 `Proxy` 中的方式，将 Remote JVM Debug 配置中的 Port 设置为上一步中获取到的端口，例如 51837。

编辑好上面的信息后，在 IDEA 中 Run -> Run -> e2e-debug -> debug 即可启动 IDEA 的远程 debug。

注意事项

1. 如需测试 Oracle，请在 pom.xml 中增加 Oracle 驱动依赖；
2. 为了保证测试数据的完整性和易读性，整合测试中的分库分表采用了 10 库 10 表的方式，完全运行测试用例所需时间较长。

11.5 性能测试

提供各个压测工具的性能测试结果。

11.5.1 Sysbench ShardingSphere Proxy 空 Rules 性能测试

测试目的

对 ShardingSphere-Proxy 及 MySQL 进行性能对比 1. sysbench 直接压测 MySQL 性能 2. sysbench 压测 ShardingSphere-Proxy(底层透传 MySQL)

基于以上两组实验，得到使用 ShardingSphere-Proxy 对于 MySQL 的损耗。

测试环境搭建

服务器信息

1. DB 相关配置：推荐内存大于压测的数据量，使得数据均在内存热块中，其余可自行调整；
2. ShardingSphere-Proxy 相关配置：推荐使用高性能多核 CPU，其余可自行调整；
3. 压测涉及服务器均关闭 swap 分区。

数据库

```
[mysqld]
innodb_buffer_pool_size=${MORE_THAN_DATA_SIZE}
innodb-log-file-size=3000000000
innodb-log-files-in-group=5
innodb-flush-log-at-trx-commit=0
innodb-change-buffer-max-size=40
back_log=900
innodb_max_dirty_pages_pct=75
innodb_open_files=20480
innodb_buffer_pool_instances=8
innodb_page_cleaners=8
innodb_purge_threads=2
innodb_read_io_threads=8
innodb_write_io_threads=8
table_open_cache=102400
log_timestamps=system
thread_cache_size=16384
transaction_isolation=READ-COMMITTED

# 可参考进行适当调优，旨在放大底层 DB 性能，不让实验受制于 DB 性能瓶颈。
```

压测工具

可通过 sysbench 官网自行获取

ShardingSphere-Proxy

bin/start.sh

```
-Xmx16g -Xms16g -Xmn8g  # 调整 JVM 相关参数
```

config.yaml

```
databaseName: sharding_db

dataSources:
  ds_0:
    url: jdbc:mysql://***.***.***.***:****/test?serverTimezone=UTC&useSSL=false # 参数可适当调整
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
```

```
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200 # 最大链接池设为 ${压测并发数} 与压测并发数保持一致，屏蔽压测过程中额外的链接带来的影响
    minPoolSize: 200 # 最小链接池设为 ${压测并发数} 与压测并发数保持一致，屏蔽压测过程中初始化链接带来的影响

rules: []
```

测试阶段

环境准备

```
sysbench oltp_read_write --mysql-host=${DB_IP} --mysql-port=${DB_PORT} --mysql-user=${USER} --mysql-password=${PASSWD} --mysql-db=test --tables=10 --table-size=1000000 --report-interval=10 --time=100 --threads=200  cleanup
sysbench oltp_read_write --mysql-host=${DB_IP} --mysql-port=${DB_PORT} --mysql-user=${USER} --mysql-password=${PASSWD} --mysql-db=test --tables=10 --table-size=1000000 --report-interval=10 --time=100 --threads=200  prepare
```

压测命令

```
sysbench oltp_read_write --mysql-host=${DB/PROXY_IP} --mysql-port=${DB/PROXY_PORT} --mysql-user=${USER} --mysql-password=${PASSWD} --mysql-db=test --tables=10 --table-size=1000000 --report-interval=10 --time=100 --threads=200  run
```

压测报告分析

```
sysbench 1.0.20 (using bundled LuaJIT 2.1.0-beta2)
Running the test with following options:
Number of threads: 200
Report intermediate results every 10 second(s)
Initializing random number generator from current time
Initializing worker threads...
Threads started!
# 每 10 秒钟报告一次测试结果，tps、每秒读、每秒写、95% 以上的响应时长统计
[ 10s ] thds: 200 tps: 11161.70 qps: 223453.06 (r/w/o: 156451.76/44658.51/22342.80) lat (ms,95%): 27.17 err/s: 0.00 reconn/s: 0.00
...
[ 120s ] thds: 200 tps: 11731.00 qps: 234638.36 (r/w/o: 164251.67/46924.69/23462.00) lat (ms,95%): 24.38 err/s: 0.00 reconn/s: 0.00
SQL statistics:
    queries performed:
```

```
        read:                            19560590                       # 读总数
        write:                           5588740                        # 写总数
        other:                           27943700                       # 其他操作总
数 (COMMIT 等)
        total:                           27943700                       # 全部总数
    transactions:                        1397185 (11638.59 per sec.)    # 总事务数 (
每秒事务数 )
    queries:                             27943700 (232771.76 per sec.)  # 执行语句总
数 ( 每秒执行语句次数 )
    ignored errors:                      0      (0.00 per sec.)         # 忽略错误数
( 每秒忽略错误数 )
    reconnects:                          0      (0.00 per sec.)         # 重连次数 (
每秒重连次数 )

General statistics:
    total time:                          120.0463s                      # 总共耗时
    total number of events:              1397185                        # 总共发生多
少事务数

Latency (ms):
         min:                                    5.37                   # 最小延时
         avg:                                   17.13                   # 平均延时
         max:                                  109.75                   # 最大延时
         95th percentile:                       24.83                   # 超过 95%
平均耗时
         sum:                             23999546.19

Threads fairness:
    events (avg/stddev):           6985.9250/34.74                    # 平均每线程
完成 6985.9250 次 event，标准差为 34.74
    execution time (avg/stddev):   119.9977/0.01                      # 每个线程平
均耗时 119.9977 秒，标准差为 0.01
```

压测过程中值得关注的点

1. ShardingSphere-Proxy 所在服务器 CPU 利用率，充分利用 CPU 为佳；
2. DB 所在服务器磁盘 IO，物理读越低越好；
3. 压测中涉及服务器的网络 IO。

11.5.2 BenchmarkSQL ShardingSphere Proxy 分片性能测试

测试目的

使用 BenchmarkSQL 工具测试 ShardingSphere Proxy 的分片性能。

测试方法

ShardingSphere Proxy 支持通过 BenchmarkSQL 5.0 进行 TPC-C 测试。除本文说明的内容外，BenchmarkSQL 操作步骤按照原文档 `HOW-TO-RUN.txt` 即可。

测试工具微调

与单机数据库压测不同，分布式数据库解决方案难免在功能支持上有所取舍。使用 BenchmarkSQL 压测 ShardingSphere Proxy 建议进行如下调整。

移除外键与 extraHistID

修改 BenchmarkSQL 目录下 `run/runDatabaseBuild.sh`，文件第 17 行。

修改前：

```
AFTER_LOAD="indexCreates foreignKeys extraHistID buildFinish"
```

修改后：

```
AFTER_LOAD="indexCreates buildFinish"
```

压测环境或参数建议

注意：本节中提到的任何参数都不是绝对值，都需要根据实际测试结果进行调整或取舍。

建议使用 Java 17 运行 ShardingSphere

编译 ShardingSphere 可以使用 Java 8。

使用 Java 17 可以在默认情况下尽量提升 ShardingSphere 的性能。

ShardingSphere 数据分片建议

对 BenchmarkSQL 的数据分片，可以考虑以各个表中的 warehouse id 作为分片键。

其中一个表 `bmsql_item` 没有 warehouse id，数据量固定 10 万行：- 可以取 `i_id` 作为分片键。但可能会导致同一个 Proxy 连接同时持有多个不同数据源的连接。- 或考虑不做分片，存在单个数据源内。可能会导致某一数据源压力较大。- 或对 `i_id` 进行范围分片，例如 1-50000 分布在数据源 0、50001-100000 分布在数据源 1。

BenchmarkSQL 中有如下 SQL 涉及多表：

```
SELECT c_discount, c_last, c_credit, w_tax
FROM bmsql_customer
    JOIN bmsql_warehouse ON (w_id = c_w_id)
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

```
SELECT o_id, o_entry_d, o_carrier_id
FROM bmsql_oorder
WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
  AND o_id = (
      SELECT max(o_id)
      FROM bmsql_oorder
      WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
      )
```

如果以 warehouse id 作为分片键，以上 SQL 涉及的表可以配置为 bindingTable：

```
rules:
  - !SHARDING
    bindingTables:
      - bmsql_warehouse, bmsql_customer
      - bmsql_stock, bmsql_district, bmsql_order_line
```

以 warehouse id 为分片键的数据分片配置可以参考本文附录。

PostgreSQL JDBC URL 参数建议

对 BenchmarkSQL 所使用的配置文件中的 JDBC URL 进行调整，即参数名 `conn` 的值：- 增加参数 `defaultRowFetchSize=50` 可能减少多行结果集的 fetch 次数，需要根据实际测试结果适当增大或减小。- 增加参数 `reWriteBatchedInserts=true` 可能减少批量插入的耗时，例如准备数据或 New Order 业务的批量插入，需要根据实际测试结果决定是否启用。

props.pg 文件节选，建议修改的位置为第 3 行 `conn` 的参数值：

```
db=postgres
driver=org.postgresql.Driver
conn=jdbc:postgresql://localhost:5432/postgres?defaultRowFetchSize=50&
reWriteBatchedInserts=true
```

```
user=benchmarksql
password=PWbmsql
```

ShardingSphere Proxy global.yaml 参数建议

`proxy-backend-query-fetch-size` 参数值默认值为 -1，修改为 50 左右可以尽量减少多行结果集的 fetch 次数。`proxy-frontend-executor-size` 参数默认值为 CPU * 2，可以根据实际测试结果减少至 CPU * 0.5 左右；如果涉及 NUMA，可以根据实际测试结果设置为单个 CPU 的物理核数。

`global.yaml` 文件节选：

```
props:
  proxy-backend-query-fetch-size: 50
  # proxy-frontend-executor-size: 32 # 4 路 32C aarch64
  # proxy-frontend-executor-size: 12 # 2 路 12C24T x86
```

附录

BenchmarkSQL 数据分片参考配置

Pool size 请根据实际压测情况适当调整。

```
databaseName: bmsql_sharding
dataSources:
  ds_0:
    url: jdbc:postgresql://db0.ip:5432/bmsql
    username: postgres
    password: postgres
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000
  ds_1:
    url: jdbc:postgresql://db1.ip:5432/bmsql
    username: postgres
    password: postgres
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000
  ds_2:
    url: jdbc:postgresql://db2.ip:5432/bmsql
    username: postgres
    password: postgres
```

```
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000
  ds_3:
    url: jdbc:postgresql://db3.ip:5432/bmsql
    username: postgres
    password: postgres
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000

rules:
  - !SHARDING
    bindingTables:
      - bmsql_warehouse, bmsql_customer
      - bmsql_stock, bmsql_district, bmsql_order_line
    defaultDatabaseStrategy:
      none:
    defaultTableStrategy:
      none:
    keyGenerators:
      snowflake:
        type: SNOWFLAKE
    tables:
      bmsql_config:
        actualDataNodes: ds_0.bmsql_config

      bmsql_warehouse:
        actualDataNodes: ds_${0..3}.bmsql_warehouse
        databaseStrategy:
          standard:
            shardingColumn: w_id
            shardingAlgorithmName: mod_4

      bmsql_district:
        actualDataNodes: ds_${0..3}.bmsql_district
        databaseStrategy:
          standard:
            shardingColumn: d_w_id
            shardingAlgorithmName: mod_4

      bmsql_customer:
        actualDataNodes: ds_${0..3}.bmsql_customer
        databaseStrategy:
```

```
        standard:
          shardingColumn: c_w_id
          shardingAlgorithmName: mod_4

    bmsql_item:
      actualDataNodes: ds_${0..3}.bmsql_item
      databaseStrategy:
        standard:
          shardingColumn: i_id
          shardingAlgorithmName: mod_4

    bmsql_history:
      actualDataNodes: ds_${0..3}.bmsql_history
      databaseStrategy:
        standard:
          shardingColumn: h_w_id
          shardingAlgorithmName: mod_4

    bmsql_oorder:
      actualDataNodes: ds_${0..3}.bmsql_oorder
      databaseStrategy:
        standard:
          shardingColumn: o_w_id
          shardingAlgorithmName: mod_4

    bmsql_stock:
      actualDataNodes: ds_${0..3}.bmsql_stock
      databaseStrategy:
        standard:
          shardingColumn: s_w_id
          shardingAlgorithmName: mod_4

    bmsql_new_order:
      actualDataNodes: ds_${0..3}.bmsql_new_order
      databaseStrategy:
        standard:
          shardingColumn: no_w_id
          shardingAlgorithmName: mod_4

    bmsql_order_line:
      actualDataNodes: ds_${0..3}.bmsql_order_line
      databaseStrategy:
        standard:
          shardingColumn: ol_w_id
          shardingAlgorithmName: mod_4

  shardingAlgorithms:
    mod_4:
```

```
        type: MOD
        props:
          sharding-count: 4
```

BenchmarkSQL 5.0 PostgreSQL 语句列表

Create tables

```
create table bmsql_config (
  cfg_name    varchar(30) primary key,
  cfg_value   varchar(50)
);

create table bmsql_warehouse (
  w_id        integer   not null,
  w_ytd       decimal(12,2),
  w_tax       decimal(4,4),
  w_name      varchar(10),
  w_street_1  varchar(20),
  w_street_2  varchar(20),
  w_city      varchar(20),
  w_state     char(2),
  w_zip       char(9)
);

create table bmsql_district (
  d_w_id       integer       not null,
  d_id         integer       not null,
  d_ytd        decimal(12,2),
  d_tax        decimal(4,4),
  d_next_o_id  integer,
  d_name       varchar(10),
  d_street_1   varchar(20),
  d_street_2   varchar(20),
  d_city       varchar(20),
  d_state      char(2),
  d_zip        char(9)
);

create table bmsql_customer (
  c_w_id         integer        not null,
  c_d_id         integer        not null,
  c_id           integer        not null,
  c_discount     decimal(4,4),
  c_credit       char(2),
  c_last         varchar(16),
  c_first        varchar(16),
```

```
  c_credit_lim   decimal(12,2),
  c_balance      decimal(12,2),
  c_ytd_payment  decimal(12,2),
  c_payment_cnt  integer,
  c_delivery_cnt integer,
  c_street_1     varchar(20),
  c_street_2     varchar(20),
  c_city         varchar(20),
  c_state        char(2),
  c_zip          char(9),
  c_phone        char(16),
  c_since        timestamp,
  c_middle       char(2),
  c_data         varchar(500)
);

create sequence bmsql_hist_id_seq;

create table bmsql_history (
  hist_id  integer,
  h_c_id   integer,
  h_c_d_id integer,
  h_c_w_id integer,
  h_d_id   integer,
  h_w_id   integer,
  h_date   timestamp,
  h_amount decimal(6,2),
  h_data   varchar(24)
);

create table bmsql_new_order (
  no_w_id  integer   not null,
  no_d_id  integer   not null,
  no_o_id  integer   not null
);

create table bmsql_oorder (
  o_w_id       integer      not null,
  o_d_id       integer      not null,
  o_id         integer      not null,
  o_c_id       integer,
  o_carrier_id integer,
  o_ol_cnt     integer,
  o_all_local  integer,
  o_entry_d    timestamp
);

create table bmsql_order_line (
```

```
  ol_w_id         integer   not null,
  ol_d_id         integer   not null,
  ol_o_id         integer   not null,
  ol_number       integer   not null,
  ol_i_id         integer   not null,
  ol_delivery_d   timestamp,
  ol_amount       decimal(6,2),
  ol_supply_w_id  integer,
  ol_quantity     integer,
  ol_dist_info    char(24)
);

create table bmsql_item (
  i_id     integer      not null,
  i_name   varchar(24),
  i_price  decimal(5,2),
  i_data   varchar(50),
  i_im_id  integer
);

create table bmsql_stock (
  s_w_id       integer       not null,
  s_i_id       integer       not null,
  s_quantity   integer,
  s_ytd        integer,
  s_order_cnt  integer,
  s_remote_cnt integer,
  s_data       varchar(50),
  s_dist_01    char(24),
  s_dist_02    char(24),
  s_dist_03    char(24),
  s_dist_04    char(24),
  s_dist_05    char(24),
  s_dist_06    char(24),
  s_dist_07    char(24),
  s_dist_08    char(24),
  s_dist_09    char(24),
  s_dist_10    char(24)
);
```

Create indexes

```
alter table bmsql_warehouse add constraint bmsql_warehouse_pkey
  primary key (w_id);

alter table bmsql_district add constraint bmsql_district_pkey
  primary key (d_w_id, d_id);

alter table bmsql_customer add constraint bmsql_customer_pkey
  primary key (c_w_id, c_d_id, c_id);

create index bmsql_customer_idx1
  on  bmsql_customer (c_w_id, c_d_id, c_last, c_first);

alter table bmsql_oorder add constraint bmsql_oorder_pkey
  primary key (o_w_id, o_d_id, o_id);

create unique index bmsql_oorder_idx1
  on  bmsql_oorder (o_w_id, o_d_id, o_carrier_id, o_id);

alter table bmsql_new_order add constraint bmsql_new_order_pkey
  primary key (no_w_id, no_d_id, no_o_id);

alter table bmsql_order_line add constraint bmsql_order_line_pkey
  primary key (ol_w_id, ol_d_id, ol_o_id, ol_number);

alter table bmsql_stock add constraint bmsql_stock_pkey
  primary key (s_w_id, s_i_id);

alter table bmsql_item add constraint bmsql_item_pkey
  primary key (i_id);
```

New Order 业务

stmtNewOrderSelectWhseCust

```
UPDATE bmsql_district
    SET d_next_o_id = d_next_o_id + 1
    WHERE d_w_id = ? AND d_id = ?
```

stmtNewOrderSelectDist

```
SELECT d_tax, d_next_o_id
    FROM bmsql_district
    WHERE d_w_id = ? AND d_id = ?
    FOR UPDATE
```

stmtNewOrderUpdateDist

```
UPDATE bmsql_district
    SET d_next_o_id = d_next_o_id + 1
    WHERE d_w_id = ? AND d_id = ?
```

stmtNewOrderInsertOrder

```
INSERT INTO bmsql_oorder (
    o_id, o_d_id, o_w_id, o_c_id, o_entry_d,
    o_ol_cnt, o_all_local)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

stmtNewOrderInsertNewOrder

```
INSERT INTO bmsql_new_order (
    no_o_id, no_d_id, no_w_id)
VALUES (?, ?, ?)
```

stmtNewOrderSelectStock

```
SELECT s_quantity, s_data,
       s_dist_01, s_dist_02, s_dist_03, s_dist_04,
       s_dist_05, s_dist_06, s_dist_07, s_dist_08,
       s_dist_09, s_dist_10
    FROM bmsql_stock
    WHERE s_w_id = ? AND s_i_id = ?
    FOR UPDATE
```

stmtNewOrderSelectItem

```
SELECT i_price, i_name, i_data
    FROM bmsql_item
    WHERE i_id = ?
```

stmtNewOrderUpdateStock

```
UPDATE bmsql_stock
    SET s_quantity = ?, s_ytd = s_ytd + ?,
        s_order_cnt = s_order_cnt + 1,
        s_remote_cnt = s_remote_cnt + ?
    WHERE s_w_id = ? AND s_i_id = ?
```

stmtNewOrderInsertOrderLine

```
INSERT INTO bmsql_order_line (
    ol_o_id, ol_d_id, ol_w_id, ol_number,
    ol_i_id, ol_supply_w_id, ol_quantity,
    ol_amount, ol_dist_info)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
```

Payment 业务

stmtPaymentSelectWarehouse

```
SELECT w_name, w_street_1, w_street_2, w_city,
       w_state, w_zip
    FROM bmsql_warehouse
    WHERE w_id = ?
```

stmtPaymentSelectDistrict

```
SELECT d_name, d_street_1, d_street_2, d_city,
       d_state, d_zip
    FROM bmsql_district
    WHERE d_w_id = ? AND d_id = ?
```

stmtPaymentSelectCustomerListByLast

```
SELECT c_id
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?
    ORDER BY c_first
```

stmtPaymentSelectCustomer

```
SELECT c_first, c_middle, c_last, c_street_1, c_street_2,
       c_city, c_state, c_zip, c_phone, c_since, c_credit,
       c_credit_lim, c_discount, c_balance
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
    FOR UPDATE
```

stmtPaymentSelectCustomerData

```
SELECT c_data
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtPaymentUpdateWarehouse

```
UPDATE bmsql_warehouse
    SET w_ytd = w_ytd + ?
    WHERE w_id = ?
```

stmtPaymentUpdateDistrict

```
UPDATE bmsql_district
    SET d_ytd = d_ytd + ?
    WHERE d_w_id = ? AND d_id = ?
```

stmtPaymentUpdateCustomer

```
UPDATE bmsql_customer
    SET c_balance = c_balance - ?,
        c_ytd_payment = c_ytd_payment + ?,
        c_payment_cnt = c_payment_cnt + 1
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtPaymentUpdateCustomerWithData

```
UPDATE bmsql_customer
    SET c_balance = c_balance - ?,
        c_ytd_payment = c_ytd_payment + ?,
        c_payment_cnt = c_payment_cnt + 1,
        c_data = ?
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtPaymentInsertHistory

```
INSERT INTO bmsql_history (
    h_c_id, h_c_d_id, h_c_w_id, h_d_id, h_w_id,
    h_date, h_amount, h_data)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)
```

Order Status 业务

stmtOrderStatusSelectCustomerListByLast

```
SELECT c_id
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?
    ORDER BY c_first
```

stmtOrderStatusSelectCustomer

```
SELECT c_first, c_middle, c_last, c_balance
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtOrderStatusSelectLastOrder

```
SELECT o_id, o_entry_d, o_carrier_id
    FROM bmsql_oorder
    WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
      AND o_id = (
          SELECT max(o_id)
              FROM bmsql_oorder
              WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
          )
```

stmtOrderStatusSelectOrderLine

```
SELECT ol_i_id, ol_supply_w_id, ol_quantity,
       ol_amount, ol_delivery_d
    FROM bmsql_order_line
    WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
    ORDER BY ol_w_id, ol_d_id, ol_o_id, ol_number
```

Stock level 业务

stmtStockLevelSelectLow

```
SELECT count(*) AS low_stock FROM (
    SELECT s_w_id, s_i_id, s_quantity
        FROM bmsql_stock
        WHERE s_w_id = ? AND s_quantity < ? AND s_i_id IN (
            SELECT ol_i_id
                FROM bmsql_district
                JOIN bmsql_order_line ON ol_w_id = d_w_id
                 AND ol_d_id = d_id
                 AND ol_o_id >= d_next_o_id - 20
                 AND ol_o_id < d_next_o_id
                WHERE d_w_id = ? AND d_id = ?
        )
    ) AS L
```

Delivery BG 业务

stmtDeliveryBGSelectOldestNewOrder

```
SELECT no_o_id
    FROM bmsql_new_order
    WHERE no_w_id = ? AND no_d_id = ?
    ORDER BY no_o_id ASC
```

stmtDeliveryBGDeleteOldestNewOrder

```
DELETE FROM bmsql_new_order
    WHERE no_w_id = ? AND no_d_id = ? AND no_o_id = ?
```

stmtDeliveryBGSelectOrder

```
SELECT o_c_id
    FROM bmsql_oorder
    WHERE o_w_id = ? AND o_d_id = ? AND o_id = ?
```

stmtDeliveryBGUpdateOrder

```
UPDATE bmsql_oorder
    SET o_carrier_id = ?
    WHERE o_w_id = ? AND o_d_id = ? AND o_id = ?
```

stmtDeliveryBGSelectSumOLAmount

```
SELECT sum(ol_amount) AS sum_ol_amount
    FROM bmsql_order_line
    WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
```

stmtDeliveryBGUpdateOrderLine

```
UPDATE bmsql_order_line
SET ol_delivery_d = ?
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
```

stmtDeliveryBGUpdateCustomer

```
UPDATE bmsql_customer
    SET c_balance = c_balance + ?,
        c_delivery_cnt = c_delivery_cnt + 1
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

11.6 模块测试

提供复杂模块的测试引擎。

11.6.1 SQL 解析测试

数据准备

SQL 解析无需真实的测试环境，开发者只需定义好待测试的 SQL，以及解析后的断言数据即可：

SQL 数据

在集成测试的部分提到过 `sql-case-id`，其对应的 SQL，可以在不同模块共享。开发者只需要在 `shardingsphere-test-it-parser` 模块中添加待测试的 SQL 即可，位置位于 `test/it/parser/src/main/resources/sql/supported/${SQL-TYPE}/*.xml`。

断言数据

断言的解析数据保存在 `test/it/parser/src/main/resources/case/${SQL-TYPE}/*.xml` 在 `xml` 文件中，可以针对表名，token，SQL 条件等进行断言，例如如下的配置：

```
<parser-result-sets>
    <parser-result sql-case-id="insert_with_multiple_values">
        <tables>
            <table name="t_order" />
        </tables>
        <tokens>
            <table-token start-index="12" table-name="t_order" length="7" />
        </tokens>
        <sharding-conditions>
            <and-condition>
                <condition column-name="order_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="1" type="int" />
                </condition>
                <condition column-name="user_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="1" type="int" />
                </condition>
            </and-condition>
            <and-condition>
                <condition column-name="order_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="2" type="int" />
                </condition>
                <condition column-name="user_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="2" type="int" />
                </condition>
            </and-condition>
        </sharding-conditions>
    </parser-result>
</parser-result-sets>
```

设置好上面两类数据，开发者就可以通过 `test/it/parser` 下对应的测试引擎启动 SQL 解析的测试了。

11.6.2 SQL 改写测试

目标

面向逻辑库与逻辑表书写的 SQL，并不能够直接在真实的数据库中执行，SQL 改写用于将逻辑 SQL 改写为在真实数据库中可以正确执行的 SQL。它包括正确性改写和优化改写两部分，所以 SQL 改写的测试都是基于这些改写方向进行校验的。

测试

SQL 改写测试用例位于 `test/it/rewriter` 下的 test 中。SQL 改写的测试主要依赖如下几个部分：

- 测试引擎
- 环境配置
- 验证数据

测试引擎是 SQL 改写测试的入口，跟其他引擎一样，通过 Junit 的 Parameterized 逐条读取 `test\resources` 目录中测试类型下对应的 xml 文件，然后按读取顺序一一进行验证。

环境配置存放在 `test\resources\yaml` 路径中测试类型下对应的 yaml 中。配置了 dataSources,shardingRule，encryptRule 等信息，例子如下：

```
dataSources:
  db: !!com.zaxxer.hikari.HikariDataSource
    driverClassName: org.h2.Driver
    jdbcUrl: jdbc:h2:mem:db;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MYSQL
    username: sa
    password:

## sharding 规则
rules:
- !SHARDING
  tables:
    t_account:
      actualDataNodes: db.t_account_${0..1}
      tableStrategy:
        standard:
          shardingColumn: account_id
          shardingAlgorithmName: account_table_inline
      keyGenerateStrategy:
        column: account_id
        keyGeneratorName: snowflake
    t_account_detail:
      actualDataNodes: db.t_account_detail_${0..1}
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: account_detail_table_inline
```

```
  bindingTables:
    - t_account, t_account_detail
  shardingAlgorithms:
    account_table_inline:
      type: INLINE
      props:
        algorithm-expression: t_account_${account_id % 2}
    account_detail_table_inline:
      type: INLINE
      props:
        algorithm-expression: t_account_detail_${account_id % 2}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
```

验证数据存放在 `test\resources` 路径中测试类型下对应的 xml 文件中。验证数据中，`yaml-rule` 指定了环境以及 rule 的配置文件，`input` 指定了待测试的 SQL 以及参数，`output` 指定了期待的 SQL 以及参数。其中 `db-type` 决定了 SQL 解析的类型，默认为 `SQL92`，例如：

```
<rewrite-assertions yaml-rule="yaml/sharding/sharding-rule.yaml">
    <!-- 替换数据库类型需要在这里更改 db-type -->
    <rewrite-assertion id="create_index_for_mysql" db-type="MySQL">
        <input sql="CREATE INDEX index_name ON t_account ('status')" />
        <output sql="CREATE INDEX index_name ON t_account_0 ('status')" />
        <output sql="CREATE INDEX index_name ON t_account_1 ('status')" />
    </rewrite-assertion>
</rewrite-assertions>
```

只需在 xml 文件中编写测试数据，配置好相应的 yaml 配置文件，就可以在不更改任何 Java 代码的情况下校验对应的 SQL 了。

11.7 Pipeline E2E 测试

11.7.1 测试目的

验证 pipeline 各个场景的功能正确性。

11.7.2 测试环境类型

目前支持 NATIVE 和 DOCKER。

1. NATIVE：运行在开发者本机环境。需要在本机启动 ShardingSphere-Proxy 实例和数据库实例。一般用于本机调试。
2. DOCKER：运行在 Maven 插件拉起的 docker 环境。一般用于 GitHub Action，也可以在本机运行。

支持的数据库：MySQL、PostgreSQL、openGauss。

11.7.3 使用指南

模块路径 `test/e2e/operation/pipeline`。

环境配置

`${DOCKER-IMAGE}` 表示 docker 镜像名称，如 `mysql:5.7` 。`${DATABASE-TYPE}` 表示数据库类型。

目录：`src/test/resources/env/` - `it-env.properties`：环境配置文件。- `${DATABASE-TYPE}/global.yaml`：ShardingSphere-Proxy 配置文件。- `${DATABASE-TYPE}/initdb.sql`：数据库初始化 SQL 文件。- `${DATABASE-TYPE}/*.cnf,*.conf`：数据库配置文件。- `common/*.xml`：测试用到的 DistSQL 文件。- `scenario/`：各个测试场景的 SQL 文件。

测试用例

用例示例：MySQLMigrationGeneralE2EIT。覆盖的功能点如下：- 库级别迁移（所有表）- 表级别迁移（任意多个表）- 迁移数据一致性校验 - 数据迁移过程中支持重启 - 数据迁移支持整型主键 - 数据迁移支持字符串主键 - 使用非管理员账号进行数据迁移

运行测试用例

`it-env.properties` 所有属性都可以通过 Maven 命令行 `-D` 的方式传入，优先级高于配置文件。

NATIVE 环境启动

1. 在本地启动 ShardingSphere-Proxy（使用 3307 端口）：参考 proxy 启动手册，或者修改 `proxy/bootstrap/src/main/resources/conf/global.yaml` 之后在 IDE 运行 `org.apache.shardingsphere.proxy.Bootstrap`。

Proxy 配置可以参考：- test/e2e/operation/pipeline/src/test/resources/env/mysql/server-8.yaml - test/e2e/operation/pipeline/src/test/resources/env/postgresql/global.yaml - test/e2e/operation/pipeline/src/test/resources/env/opengauss/global.yaml

2. 启动注册中心（如 ZooKeeper）和数据库。
3. 以 MySQL 为例，`it-env.properties` 可以配置如下：

```
pipeline.it.env.type=NATIVE
pipeline.it.native.database=mysql
pipeline.it.native.mysql.username=root
pipeline.it.native.mysql.password=root
pipeline.it.native.mysql.port=3306
```

4. 找到对应的测试类，在 IDE 启动运行。

DOCKER 环境启动

参考 `.github/workflows/e2e-pipeline.yml`。

1. 打包镜像

```
./mvnw -B clean install -am -pl test/e2e/operation/pipeline -Pit.env.docker -
DskipTests
```

运行以上命令会构建出一个用于 E2E 测试的 docker 镜像 `apache/shardingsphere-proxy-test:latest`。

该镜像设置了远程调试的端口，默认是 `3308`。

如果仅修改了测试代码，可以复用已有的测试镜像。

2. 修改 `it-env.properties` 配置

```
pipeline.it.env.type=DOCKER
pipeline.it.docker.mysql.version=mysql:5.7
```

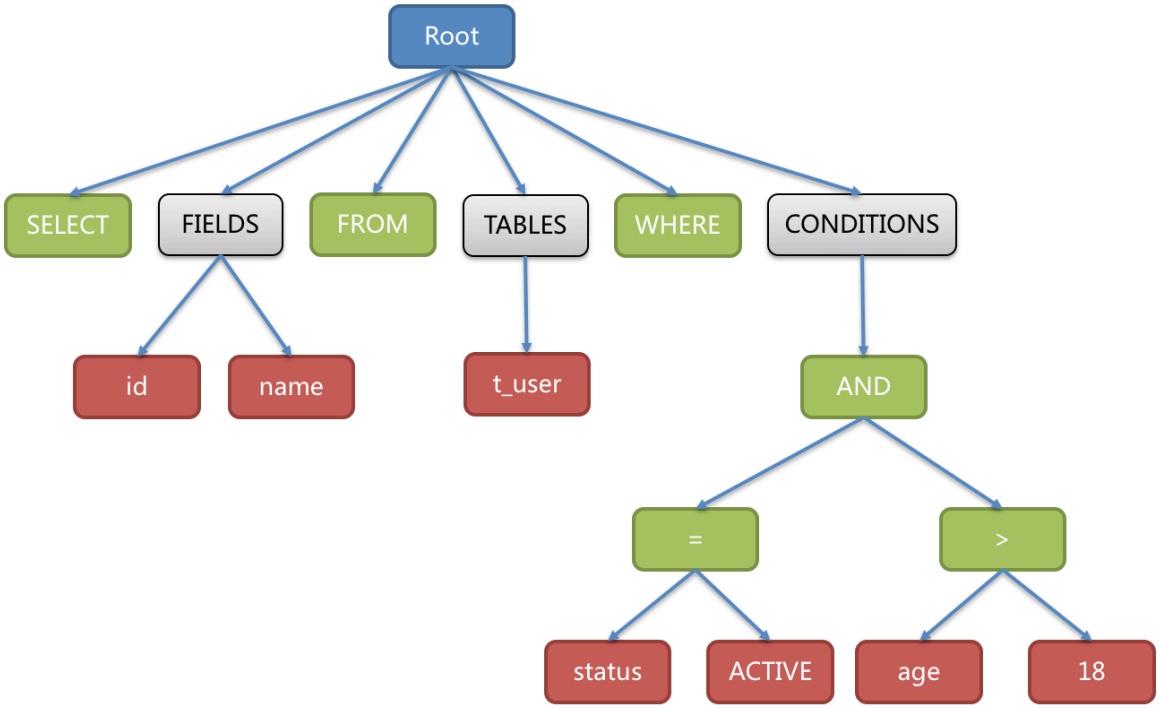
3. 通过 Maven 运行测试用例。以 MySQL 为例：

```
./mvnw -nsu -B install -f test/e2e/operation/pipeline/pom.xml -Dpipeline.it.env.
type=docker -Dpipeline.it.docker.mysql.version=mysql:5.7
```

12 技术参考

本章包含了 Apache ShardingSphere 的技术实现细节，供开发者和用户参考。

12.1 数据兼容性



- SQL 兼容

SQL 是使用者与数据库交流的标准语言。SQL 解析引擎负责将 SQL 字符串解析为抽象语法树，供 Apache ShardingSphere 理解并实现其增量功能。

ShardingSphere 目前支持 MySQL, PostgreSQL, SQLServer, Oracle, openGauss, ClickHouse, Doris, Hive, Presto 以及符合 SQL92 规范的 SQL 方言。由于 SQL 语法的复杂性，目前仍然存在少量不支持的 SQL。

- 数据库协议兼容

Apache ShardingSphere 目前根据不同的数据协议，实现了 MySQL 和 PostgreSQL 协议。

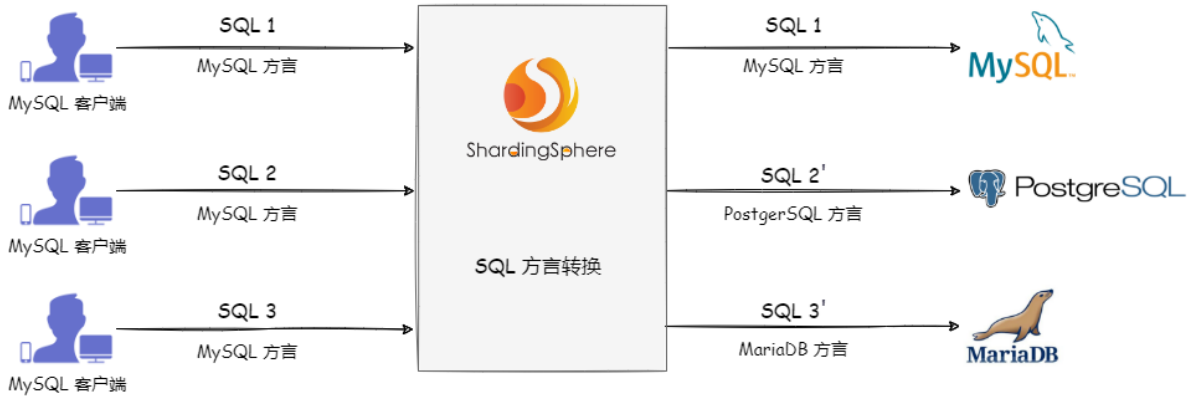
- 特性支持

Apache ShardingSphere 为数据库提供了分布式协作的能力，同时将一部分数据库特性抽象到了上层，进行统一管理，以降低用户的使用难度。

因此，对于统一提供的特性，原生的 SQL 将不再下发到数据库，并提示该操作不被支持，用户可使用 ShardingSphere 提供的的方式进行代替。

12.2 数据库网关

Apache ShardingSphere 提供了 SQL 方言翻译的能力，能实现数据库方言之间的自动转换。例如，用户可以使用 MySQL 客户端连接 ShardingSphere 并发送基于 MySQL 方言的 SQL，ShardingSphere 能自动识别用户协议与存储节点类型，自动完成 SQL 方言转换，访问 PostgreSQL 等异构存储节点。



12.3 管控

12.3.1 注册中心数据结构

在定义的命名空间下，`rules`、`props` 和 `metadata` 节点以 YAML 格式存储配置，可通过修改节点来实现对于配置的动态管理。`nodes` 存储数据库访问对象运行节点，用于区分不同数据库访问实例。`statistics` 存储系统表中的数据记录。

```
namespace
    ├──rules                                                # 全局规则配置
    ├     ├──transaction
    ├     ├     ├──active_version
    ├     ├     ├──versions
    ├     ├     ├     ├──0
    ├──props                                                 # 属性配置
    ├     ├──active_verison
    ├     ├──versions
    ├     ├     ├──0
    ├──metadata                                              # Metadata 配置
```

```
│     ├──${databaseName}
│     │     ├──data_sources
│     │     │     ├──units                              # 存储单元结构配置
│     │     │     │    ├──${dataSourceName}
│     │     │     │    │     ├──active_verison              # 激活版本
│     │     │     │    │     ├──versions                    # 版本号
│     │     │     │    │     │     ├──0
│     │     │     │    ├──...
│     │     │     ├──nodes                              # 存储节点结构配置
│     │     │     │    ├──${dataSourceName}
│     │     │     │    │     ├──active_verison              # 激活版本
│     │     │     │    │     ├──versions                    # 版本号
│     │     │     │    │     │     ├──0
│     │     │     │    ├──...
│     │     ├──schemas                                  # Schema 列表
│     │     │     ├──${schemaName}
│     │     │     │     ├──tables                       # 表结构配置
│     │     │     │     │     ├──${tableName}
│     │     │     │     │     │     ├──active_verison # 激活版本
│     │     │     │     │     │     ├──versions       # 版本号
│     │     │     │     │     │     │     ├──0
│     │     │     │     │     ├──...
│     │     │     │     ├──views                        # 视图结构配置
│     │     │     │     │     ├──${viewName}
│     │     │     │     │     │     ├──active_verison # 激活版本
│     │     │     │     │     │     ├──versions       # 版本号
│     │     │     │     │     │     │     ├──0
│     │     │     │     │     ├──...
│     │     ├──rules
│     │     │     ├──sharding
│     │     │     │     ├──algorithms
│     │     │     │     │     ├──${algorithmName}     # algorithm 名称
│     │     │     │     │     │     ├──active_verison # 激活版本
│     │     │     │     │     │     ├──versions       # 版本号
│     │     │     │     │     │     │     ├──0
│     │     │     │     │     ├──...
│     │     │     │     ├──key_generators
│     │     │     │     │     ├──${keyGeneratorName}  # keyGenerator 名称
│     │     │     │     │     │     ├──active_verison # 激活版本
│     │     │     │     │     │     ├──versions       # 版本号
│     │     │     │     │     │     │     ├──0
│     │     │     │     │     ├──...
│     │     │     │     ├──tables
│     │     │     │     │     ├──${tableName}         # 逻辑表名称
│     │     │     │     │     │     ├──active_verison # 激活版本
│     │     │     │     │     │     ├──versions       # 版本号
│     │     │     │     │     │     │     ├──0
│     │     │     │     │     ├──...
```

```
├──nodes
├    ├──compute_nodes
├    ├     ├──online
├    ├     ├     ├──proxy
├    ├     ├     ├     ├──UUID              # Proxy 实例唯一标识
├    ├     ├     ├     ├──....
├    ├     ├     ├──jdbc
├    ├     ├     ├     ├──UUID              # JDBC 实例唯一标识
├    ├     ├     ├     ├──....
├    ├     ├──status
├    ├     ├     ├──UUID
├    ├     ├     ├──....
├    ├     ├──worker_id
├    ├     ├     ├──UUID
├    ├     ├     ├──....
├    ├     ├──show_process_list_trigger
├    ├     ├     ├──process_id:UUID
├    ├     ├     ├──....
├    ├     ├──labels
├    ├     ├     ├──UUID
├    ├     ├     ├──....
├    ├──qualified_data_sources
├    ├     ├──${databaseName.groupName.dataSourceName}
├    ├     ├──${databaseName.groupName.dataSourceName}
├──statistics
├    ├──databases
├    ├     ├──shardingsphere
├    ├     ├     ├──schemas
├    ├     ├     ├     ├──shardingsphere
├    ├     ├     ├     ├     ├──tables # 系统表
├    ├     ├     ├     ├     ├   ├──cluster_information    # 集群信息表
```

/rules

全局规则配置，事务配置。

```
transaction:
  defaultType: XA
  providerType: Atomikos
```

/props

属性配置，详情请参见配置手册。

```
kernel-executor-size: 20
sql-show: true
```

/metadata/${databaseName}/data_sources/units/ds_0/versions/0

数据库连接池的，不同数据库连接池属性自适配（例如：DBCP，C3P0，Druid，HikariCP）。

```
ds_0:
  initializationFailTimeout: 1
  validationTimeout: 5000
  maxLifetime: 1800000
  leakDetectionThreshold: 0
  minimumIdle: 1
  password: root
  idleTimeout: 60000
  jdbcUrl: jdbc:mysql://127.0.0.1:3306/ds_0?serverTimezone=UTC&useSSL=false
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  maximumPoolSize: 50
  connectionTimeout: 30000
  username: root
  poolName: HikariPool-1
```

/metadata/${databaseName}/data_sources/nodes/ds_0/versions/0

数据库连接池的，不同数据库连接池属性自适配（例如：HikariCP）。

```
ds_0:
  initializationFailTimeout: 1
  validationTimeout: 5000
  maxLifetime: 1800000
  leakDetectionThreshold: 0
  minimumIdle: 1
  password: root
  idleTimeout: 60000
  jdbcUrl: jdbc:mysql://127.0.0.1:3306/ds_0?serverTimezone=UTC&useSSL=false
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  maximumPoolSize: 50
  connectionTimeout: 30000
  username: root
  poolName: HikariPool-1
```

/metadata/${databaseName}/rules/sharding/tables/t_order/versions/0

分片规则配置。

```
actualDataNodes: ds_${0..1}.t_order_${0..1}
auditStrategy:
  allowHintDisable: true
  auditorNames:
    - t_order_dml_sharding_conditions_0
databaseStrategy:
  standard:
    shardingAlgorithmName: t_order_database_inline
    shardingColumn: user_id
keyGenerateStrategy:
  column: another_id
  keyGeneratorName: t_order_snowflake
logicTable: t_order
tableStrategy:
  standard:
    shardingAlgorithmName: t_order_table_inline
    shardingColumn: order_id
```

/metadata/$databaseName/schemas/$ {schemaName}/tables/t_order/versions/0

表结构配置，每个表使用单独节点存储。

```
name: t_order                              # 表名
columns:                                   # 列
  id:                                      # 列名
    caseSensitive: false
    dataType: 0
    generated: false
    name: id
    primaryKey: trues
  order_id:
    caseSensitive: false
    dataType: 0
    generated: false
    name: order_id
    primaryKey: false
indexs:                                    # 索引
  t_user_order_id_index:                   # 索引名
    name: t_user_order_id_index
```

/nodes/compute_nodes

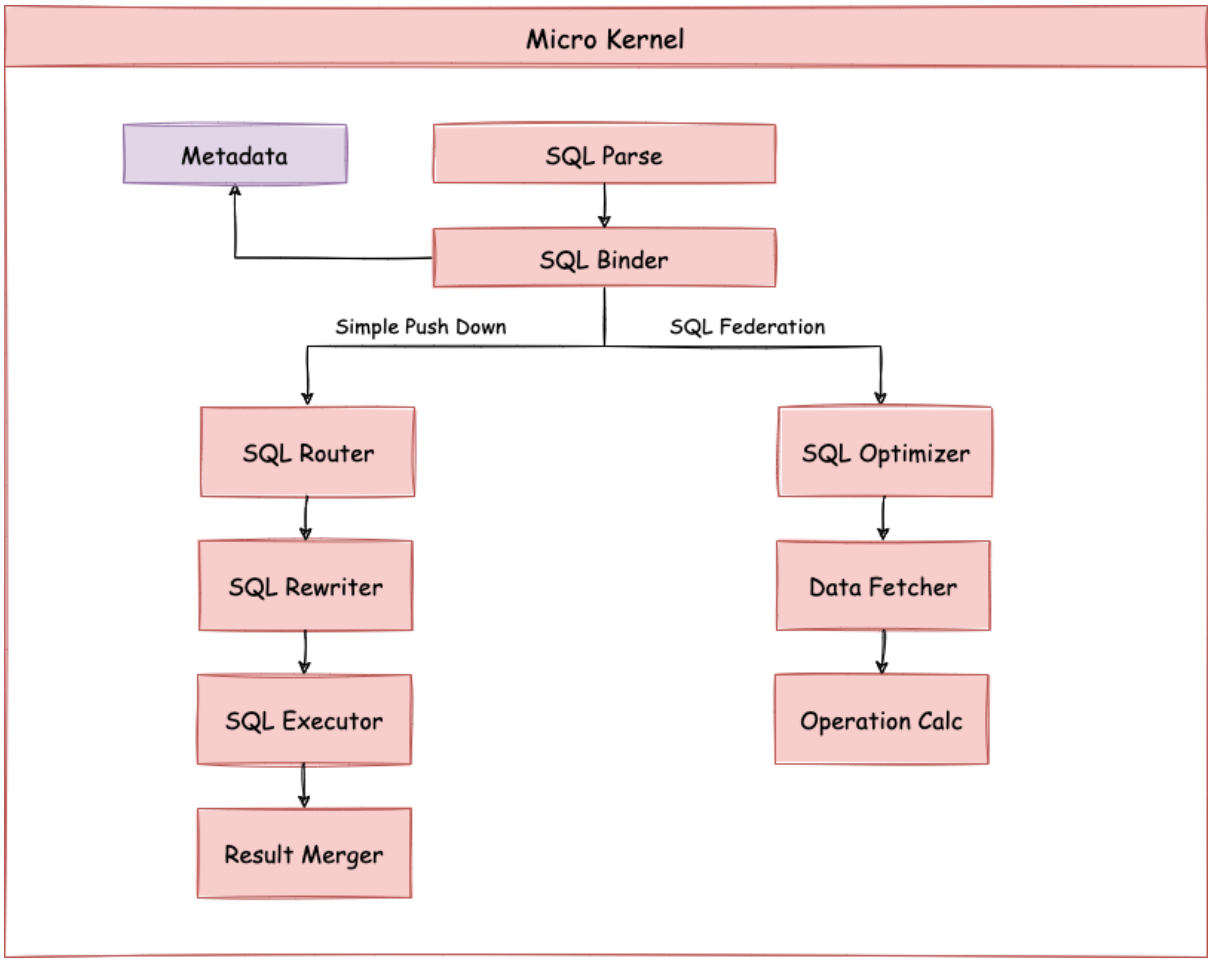
数据库访问对象运行实例信息，子节点是当前运行实例的标识。运行实例标识使用 UUID 生成，每次启动重新生成。运行实例标识均为临时节点，当实例上线时注册，下线时自动清理。注册中心监控这些节点的变化来治理运行中实例对数据库的访问等。

/nodes/qualified_data_sources

可以治理读写分离从库，可动态禁用。

12.4 数据分片

ShardingSphere 数据分片的原理如下图所示，按照是否需要进行查询优化，可以分为 Simple Push Down 下推流程和 SQL Federation 执行引擎流程。Simple Push Down 下推流程由 SQL 解析 => SQL 绑定 => SQL 路由 => SQL 改写 => SQL 执行 => 结果归并组成，主要用于处理标准分片场景下的 SQL 执行。SQL Federation 执行引擎流程由 SQL 解析 => SQL 绑定 => 逻辑优化 => 物理优化 => 数据拉取 => 算子执行组成，SQL Federation 执行引擎内部进行逻辑优化和物理优化，在优化执行阶段依赖 Standard 内核流程，对优化后的逻辑 SQL 进行路由、改写、执行和归并。



12.4.1 SQL 解析

分为词法解析和语法解析。先通过词法解析器将 SQL 拆分为一个个不可再分的单词。再使用语法解析器对 SQL 进行理解，并最终提炼出解析上下文。解析上下文包括表、选择项、排序项、分组项、聚合函数、分页信息、查询条件以及可能需要修改的占位符的标记。

12.4.2 SQL 路由

根据解析上下文匹配用户配置的分片策略，并生成路由路径。目前支持分片路由和广播路由。

12.4.3 SQL 改写

将 SQL 改写为在真实数据库中可以正确执行的语句。SQL 改写分为正确性改写和优化改写。

12.4.4 SQL 执行

通过多线程执行器异步执行。

12.4.5 结果归并

将多个执行结果集归并以便于通过统一的 JDBC 接口输出。结果归并包括流式归并、内存归并和使用装饰者模式的追加归并这几种方式。

12.4.6 查询优化

由 Federation 执行引擎（开发中）提供支持，对关联查询、子查询等复杂查询进行优化，同时支持跨多个数据库实例的分布式查询，内部使用关系代数优化查询计划，通过最优计划查询出结果。

12.4.7 解析引擎

相对于其他编程语言，SQL 是比较简单的。不过，它依然是一门完善的编程语言，因此对 SQL 的语法进行解析，与解析其他编程语言（如：Java 语言、C 语言、Go 语言等）并无本质区别。

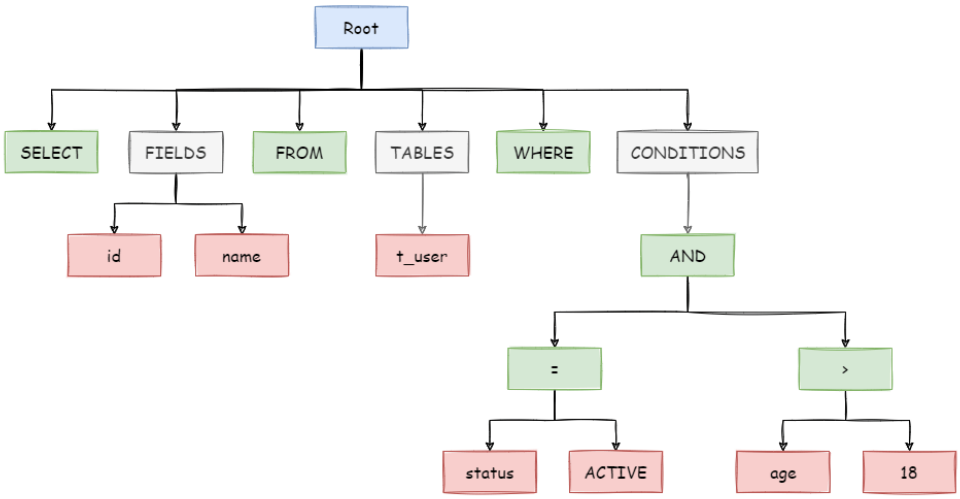
抽象语法树

解析过程分为词法解析和语法解析。词法解析器用于将 SQL 拆解为不可再分的原子符号，称为 Token。并根据不同数据库方言所提供的字典，将其归类为关键字，表达式，字面量和操作符。再使用语法解析器将词法解析器的输出转换为抽象语法树。

例如，以下 SQL：

```
SELECT id, name FROM t_user WHERE status = 'ACTIVE' AND age > 18
```

解析之后的为抽象语法树见下图。



为了便于理解，抽象语法树中的关键字的 Token 用绿色表示，变量的 Token 用红色表示，灰色表示需要进一步拆分。

最后，通过 `visitor` 对抽象语法树遍历构造域模型，通过域模型（`SQLStatement`）去提炼分片所需的上下文，并标记有可能需要改写的位置。供分片使用的解析上下文包含查询选择项（Select Items）、表信息（Table）、分片条件（Sharding Condition）、自增主键信息（Auto increment Primary Key）、排序信息（Order By）、分组信息（Group By）以及分页信息（Limit、Rownum、Top）。SQL 的一次解析过程是不可逆的，一个个 Token 按 SQL 原本的顺序依次进行解析，性能很高。考虑到各种数据库 SQL 方言的异同，在解析模块提供了各类数据库的 SQL 方言字典。

SQL 解析引擎

历史

SQL 解析作为分库分表类产品的核心，其性能和兼容性是最重要的衡量指标。ShardingSphere 的 SQL 解析器经历了 3 代产品的更新迭代。

第一代 SQL 解析器为了追求性能与快速实现，在 1.4.x 之前的版本使用 Druid 作为 SQL 解析器。经实际测试，它的性能远超其它解析器。

第二代 SQL 解析器从 1.5.x 版本开始，ShardingSphere 采用完全自研的 SQL 解析引擎。由于目的不同，ShardingSphere 并不需要将 SQL 转为一颗完全的抽象语法树，也无需通过访问器模式进行二次遍历。它采用对 SQL 半理解的方式，仅提炼数据分片需要关注的上下文，因此 SQL 解析的性能和兼容性得到了进一步的提高。

第三代 SQL 解析器从 3.0.x 版本开始，尝试使用 ANTLR 作为 SQL 解析引擎的生成器，并采用 Visit 的方式从 AST 中获取 SQL Statement。从 5.0.x 版本开始，解析引擎的架构已完成重构调整，同时通过将第一次解析得到的 AST 放入缓存，方便下次直接获取相同 SQL 的解析结果，来提高解析效率。因此我们建议用户采用 `PreparedStatement` 这种 SQL 预编译的方式来提升性能。

功能点

- 提供独立的 SQL 解析功能
- 可以非常方便的对语法规则进行扩充和修改（使用了 `ANTLR`）
- 支持多种方言的 SQL 解析

数据库	支持状态
MySQL	支持，完善
PostgreSQL	支持，完善
SQLServer	支持
Oracle	支持
SQL92	支持
openGauss	支持
ClickHouse	支持
Doris	支持
Hive	支持
Presto	支持

API 使用

- 引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-parser-sql-engine</artifactId>
    <version>${project.version}</version>
</dependency>
<!-- 根据需要引入指定方言的解析模块（以 MySQL 为例），可以添加所有支持的方言，也可以只添加使用到的
-->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-parser-sql-mysql</artifactId>
    <version>${project.version}</version>
</dependency>
```

- 获取语法树

```
CacheOption cacheOption = new CacheOption(128, 1024L);
SQLParserEngine parserEngine = new SQLParserEngine("MySQL", cacheOption);
ParseASTNode parseASTNode = parserEngine.parse(sql, useCache);
```

- 获取 SQLStatement

```
CacheOption cacheOption = new CacheOption(128, 1024L);
SQLParserEngine parserEngine = new SQLParserEngine("MySQL", cacheOption);
ParseASTNode parseASTNode = parserEngine.parse(sql, useCache);
SQLVisitorEngine sqlVisitorEngine = new SQLVisitorEngine(sql, "STATEMENT",
useCache, new Properties());
SQLStatement sqlStatement = sqlVisitorEngine.visit(parseASTNode);
```

12.4.8 路由引擎

根据解析上下文匹配数据库和表的分片策略，并生成路由路径。对于携带分片键的 SQL，根据分片键的不同可以划分为单片路由（分片键的操作符是等号）、多片路由（分片键的操作符是 IN）和范围路由（分片键的操作符是 BETWEEN）。不携带分片键的 SQL 则采用广播路由。

分片策略通常可以采用由数据库内置或由用户方配置。数据库内置的方案较为简单，内置的分片策略大致可分为尾数取模、哈希、范围、标签、时间等。由用户方配置的分片策略则更加灵活，可以根据使用方需求定制复合分片策略。如果配合数据自动迁移来使用，可以做到无需用户关注分片策略，自动由数据库中间层分片和平衡数据即可，进而做到使分布式数据库具有的弹性伸缩的能力。在 ShardingSphere 的线路规划中，弹性伸缩将于 4.x 开启。

分片路由

用于根据分片键进行路由的场景，又细分为直接路由、标准路由和笛卡尔积路由这 3 种类型。

直接路由

满足直接路由的条件相对苛刻，它需要通过 Hint（使用 HintAPI 直接指定路由至库表）方式分片，并且是只分库不分表的前提下，则可以避免 SQL 解析和之后的结果归并。因此它的兼容性最好，可以执行包括子查询、自定义函数等复杂情况的任意 SQL。直接路由还可以用于分片键不在 SQL 中的场景。例如，设置用于数据库分片的键为 3，

```
hintManager.setDatabaseShardingValue(3);
```

假如路由算法为 `value % 2`，当一个逻辑库 `t_order` 对应 2 个真实库 `t_order_0` 和 `t_order_1` 时，路由后 SQL 将在 `t_order_1` 上执行。下方是使用 API 的代码样例：

```
String sql = "SELECT * FROM t_order";
try (
        HintManager hintManager = HintManager.getInstance();
        Connection conn = dataSource.getConnection();
        PreparedStatement pstmt = conn.prepareStatement(sql)) {
```

```
    hintManager.setDatabaseShardingValue(3);
    try (ResultSet rs = pstmt.executeQuery()) {
        while (rs.next()) {
            //...
        }
    }
}
```

标准路由

标准路由是 ShardingSphere 最为推荐使用的分片方式，它的适用范围是不包含关联查询或仅包含绑定表之间关联查询的 SQL。当分片运算符是等于号时，路由结果将落入单库（表），当分片运算符是 BETWEEN 或 IN 时，则路由结果不一定落入唯一的库（表），因此一条逻辑 SQL 最终可能被拆分为多条用于执行的真实 SQL。举例说明，如果按照 `order_id` 的奇数和偶数进行数据分片，一个单表查询的 SQL 如下：

```
SELECT * FROM t_order WHERE order_id IN (1, 2);
```

那么路由的结果应为：

```
SELECT * FROM t_order_0 WHERE order_id IN (1, 2);
SELECT * FROM t_order_1 WHERE order_id IN (1, 2);
```

绑定表的关联查询与单表查询复杂度和性能相当。举例说明，如果一个包含绑定表的关联查询的 SQL 如下：

```
SELECT * FROM t_order o JOIN t_order_item i ON o.order_id=i.order_id  WHERE order_
id IN (1, 2);
```

那么路由的结果应为：

```
SELECT * FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
SELECT * FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
```

可以看到，SQL 拆分的数目与单表是一致的。

笛卡尔路由

笛卡尔路由是最复杂的情况，它无法根据绑定表的关系定位分片规则，因此非绑定表之间的关联查询需要拆解为笛卡尔积组合执行。如果上个示例中的 SQL 并未配置绑定表关系，那么路由的结果应为：

```
SELECT * FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
SELECT * FROM t_order_0 o JOIN t_order_item_1 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
SELECT * FROM t_order_1 o JOIN t_order_item_0 i ON o.order_id=i.order_id  WHERE
```

```
order_id IN (1, 2);
SELECT * FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
```

笛卡尔路由查询性能较低，需谨慎使用。

广播路由

对于不携带分片键的 SQL，则采取广播路由的方式。根据 SQL 类型又可以划分为全库表路由、全库路由、全实例路由、单播路由和阻断路由这 5 种类型。

全库表路由

全库表路由用于处理对数据库中与其逻辑表相关的所有真实表的操作，主要包括不带分片键的 DQL 和 DML，以及 DDL 等。例如：

```
SELECT * FROM t_order WHERE good_prority IN (1, 10);
```

则会遍历所有数据库中的所有表，逐一匹配逻辑表和真实表名，能够匹配得上则执行。路由后成为

```
SELECT * FROM t_order_0 WHERE good_prority IN (1, 10);
SELECT * FROM t_order_1 WHERE good_prority IN (1, 10);
SELECT * FROM t_order_2 WHERE good_prority IN (1, 10);
SELECT * FROM t_order_3 WHERE good_prority IN (1, 10);
```

全库路由

全库路由用于处理对数据库的操作，包括用于库设置的 SET 类型的数据库管理命令，以及 TCL 这样的事务控制语句。在这种情况下，会根据逻辑库的名字遍历所有符合名字匹配的真实库，并在真实库中执行该命令，例如：

```
SET autocommit=0;
```

在 `t_order` 中执行，`t_order` 有 2 个真实库。则实际会在 `t_order_0` 和 `t_order_1` 上都执行这个命令。

全实例路由

全实例路由用于 DCL 操作，授权语句针对的是数据库的实例。无论一个实例中包含多少个 Schema，每个数据库的实例只执行一次。例如：

```
CREATE USER customer@127.0.0.1 identified BY '123';
```

这个命令将在所有的真实数据库实例中执行，以确保 customer 用户可以访问每一个实例。

单播路由

单播路由用于获取某一真实表信息的场景，它仅需要从任意库中的任意真实表中获取数据即可。例如：

```
DESCRIBE t_order;
```

t_order 的两个真实表 t_order_0，t_order_1 的描述结构相同，所以这个命令在任意真实表上选择执行一次。

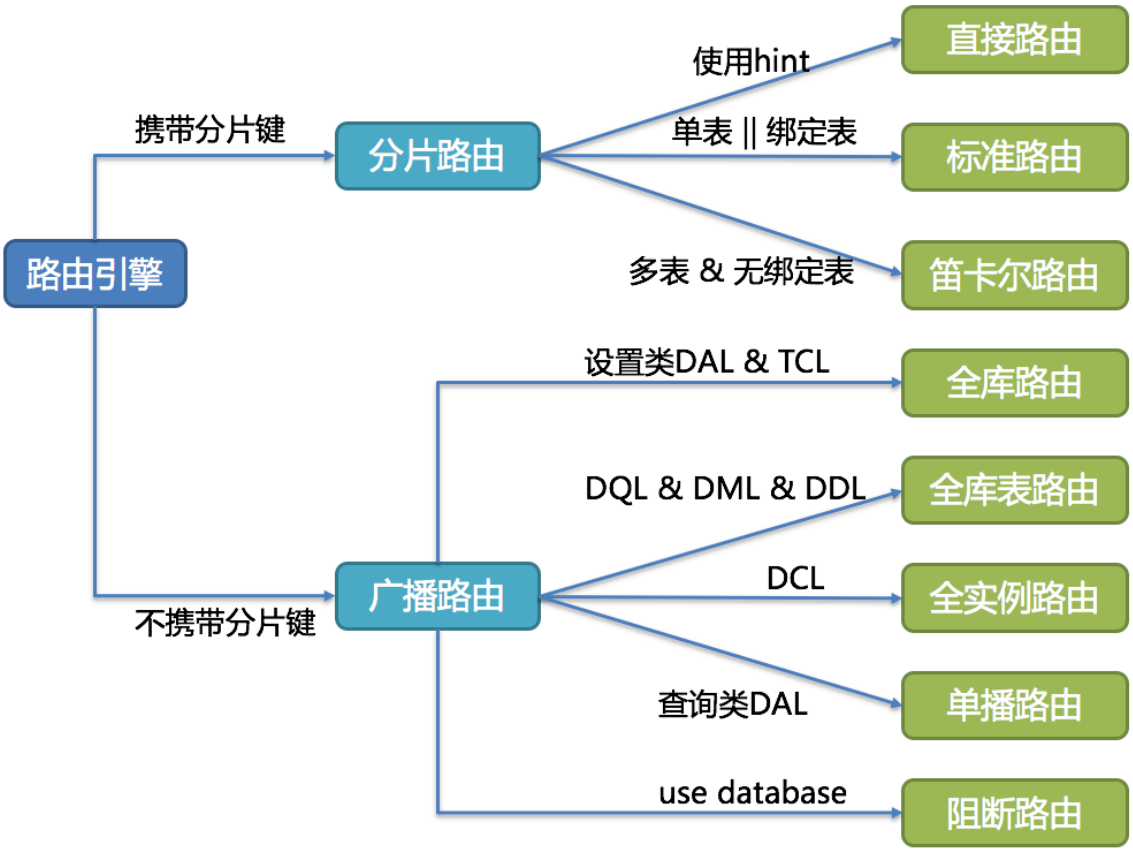
阻断路由

阻断路由用于屏蔽 SQL 对数据库的操作，例如：

```
USE order_db;
```

这个命令不会在真实数据库中执行，因为 ShardingSphere 采用的是逻辑 Schema 的方式，无需将切换数据库 Schema 的命令发送至数据库中。

路由引擎的整体结构划分如下图。



12.4.9 改写引擎

工程师面向逻辑库与逻辑表书写的 SQL，并不能够直接在真实的数据库中执行，SQL 改写用于将逻辑 SQL 改写为在真实数据库中可以正确执行的 SQL。它包括正确性改写和优化改写两部分。

正确性改写

在包含分表的场景中，需要将分表配置中的逻辑表名称改写为路由之后所获取的真实表名称。仅分库则不需要表名称的改写。除此之外，还包括补列和分页信息修正等内容。

标识符改写

需要改写的标识符包括表名称、索引名称以及 Schema 名称。

表名称改写是指将找到逻辑表在原始 SQL 中的位置，并将其改写为真实表的过程。表名称改写是一个典型的需要对 SQL 进行解析的场景。从一个最简单的例子开始，若逻辑 SQL 为：

```
SELECT order_id FROM t_order WHERE order_id=1;
```

假设该 SQL 配置分片键 order_id，并且 order_id=1 的情况，将路由至分片表 1。那么改写之后的 SQL 应该为：

```
SELECT order_id FROM t_order_1 WHERE order_id=1;
```

在这种最简单的 SQL 场景中，是否将 SQL 解析为抽象语法树似乎无关紧要，只要通过字符串查找和替换就可以达到 SQL 改写的效果。但是下面的场景，就无法仅仅通过字符串的查找替换来正确的改写 SQL 了：

```
SELECT order_id FROM t_order WHERE order_id=1 AND remarks=' t_order xxx';
```

正确改写的 SQL 应该是：

```
SELECT order_id FROM t_order_1 WHERE order_id=1 AND remarks=' t_order xxx';
```

而非：

```
SELECT order_id FROM t_order_1 WHERE order_id=1 AND remarks=' t_order_1 xxx';
```

由于表名之外可能含有表名称的类似字符，因此不能通过简单的字符串替换的方式去改写 SQL。

下面再来看一个更加复杂的 SQL 改写场景：

```
SELECT t_order.order_id FROM t_order WHERE t_order.order_id=1 AND remarks=' t_order
xxx';
```

上面的 SQL 将表名作为字段的标识符，因此在 SQL 改写时需要一并修改：

```
SELECT t_order_1.order_id FROM t_order_1 WHERE t_order_1.order_id=1 AND remarks='
t_order xxx';
```

而如果 SQL 中定义了表的别名，则无需连同别名一起修改，即使别名与表名相同亦是如此。例如：

```
SELECT t_order.order_id FROM t_order AS t_order WHERE t_order.order_id=1 AND
remarks=' t_order xxx';
```

SQL 改写则仅需要改写表名称就可以了：

```
SELECT t_order.order_id FROM t_order_1 AS t_order WHERE t_order.order_id=1 AND
remarks=' t_order xxx';
```

索引名称是另一个有可能改写的标识符。在某些数据库中（如 MySQL、SQLServer），索引是以表为维度创建的，在不同的表中的索引是可以重名的；而在另外的一些数据库中（如 PostgreSQL、Oracle），索引是以数据库为维度创建的，即使是作用在不同表上的索引，它们也要求其名称的唯一性。

在 ShardingSphere 中，管理 Schema 的方式与管理表如出一辙，它采用逻辑 Schema 去管理一组数据源。因此，ShardingSphere 需要将用户在 SQL 中书写的逻辑 Schema 替换为真实的数据库 Schema。

ShardingSphere 目前还不支持在 DQL 和 DML 语句中使用 Schema。它目前仅支持在数据库管理语句中使用 Schema，例如：

```
SHOW COLUMNS FROM t_order FROM order_ds;
```

Schema 的改写指的是将逻辑 Schema 采用单播路由的方式，改写为随机查找到的一个正确的真实 Schema。

补列

需要在查询语句中补列通常由两种情况导致。第一种情况是 ShardingSphere 需要在结果归并时获取相应数据，但该数据并未能通过查询的 SQL 返回。这种情况主要是针对 GROUP BY 和 ORDER BY。结果归并时，需要根据 `GROUP BY` 和 `ORDER BY` 的字段项进行分组和排序，但如果原始 SQL 的选择项中若并未包含分组项或排序项，则需要对原始 SQL 进行改写。先看一下原始 SQL 中带有结果归并所需信息的场景：

```
SELECT order_id, user_id FROM t_order ORDER BY user_id;
```

由于使用 user_id 进行排序，在结果归并中需要能够获取到 user_id 的数据，而上面的 SQL 是能够获取到 user_id 数据的，因此无需补列。

如果选择项中不包含结果归并时所需的列，则需要进行补列，如以下 SQL：

```
SELECT order_id FROM t_order ORDER BY user_id;
```

由于原始 SQL 中并不包含需要在结果归并中需要获取的 user_id，因此需要对 SQL 进行补列改写。补列之后的 SQL 是：

```
SELECT order_id, user_id AS ORDER_BY_DERIVED_0 FROM t_order ORDER BY user_id;
```

值得一提的是，补列只会补充缺失的列，不会全部补充，而且，在 SELECT 语句中包含 * 的 SQL，也会根据表的元数据信息选择性补列。下面是一个较为复杂的 SQL 补列场景：

```
SELECT o.* FROM t_order o, t_order_item i WHERE o.order_id=i.order_id ORDER BY
user_id, order_item_id;
```

我们假设只有 t_order_item 表中包含 order_item_id 列，那么根据表的元数据信息可知，在结果归并时，排序项中的 user_id 是存在于 t_order 表中的，无需补列；order_item_id 并不在 t_order 中，因此需要补列。补列之后的 SQL 是：

```
SELECT o.*, order_item_id AS ORDER_BY_DERIVED_0 FROM t_order o, t_order_item i
WHERE o.order_id=i.order_id ORDER BY user_id, order_item_id;
```

补列的另一种情况是使用 AVG 聚合函数。在分布式的场景中，使用 (avg1 + avg2 + avg3) / 3 计算平均值并不正确，需要改写为 (sum1 + sum2 + sum3) / (count1 + count2 + count3)。这就需要将包含 AVG 的 SQL 改写为 SUM 和 COUNT，并在结果归并时重新计算平均值。例如以下 SQL：

```
SELECT AVG(price) FROM t_order WHERE user_id=1;
```

需要改写为：

```
SELECT COUNT(price) AS AVG_DERIVED_COUNT_0, SUM(price) AS AVG_DERIVED_SUM_0 FROM t_
order WHERE user_id=1;
```

然后才能够通过结果归并正确的计算平均值。

最后一种补列是在执行 INSERT 的 SQL 语句时，如果使用数据库自增主键，是无需写入主键字段的。但数据库的自增主键是无法满足分布式场景下的主键唯一的，因此 ShardingSphere 提供了分布式自增主键的生成策略，并且可以通过补列，让使用方无需改动现有代码，即可将分布式自增主键透明的替换数据库现有的自增主键。分布式自增主键的生成策略将在下文中详述，这里只阐述与 SQL 改写相关的内容。举例说明，假设表 t_order 的主键是 order_id，原始的 SQL 为：

```
INSERT INTO t_order (`field1`, `field2`) VALUES (10, 1);
```

可以看到，上述 SQL 中并未包含自增主键，是需要数据库自行填充的。ShardingSphere 配置自增主键后，SQL 将改写为：

```
INSERT INTO t_order (`field1`, `field2`, order_id) VALUES (10, 1, xxxxx);
```

改写后的 SQL 将在 INSERT FIELD 和 INSERT VALUE 的最后部分增加主键列名称以及自动生成的自增主键值。上述 SQL 中的 `xxxxx` 表示自动生成的自增主键值。

如果 INSERT 的 SQL 中并未包含表的列名称，ShardingSphere 也可以根据判断参数个数以及表元信息中的列数量对比，并自动生成自增主键。例如，原始的 SQL 为：

```
INSERT INTO t_order VALUES (10, 1);
```

改写的 SQL 将只在主键所在的列顺序处增加自增主键即可：

```
INSERT INTO t_order VALUES (xxxxx, 10, 1);
```

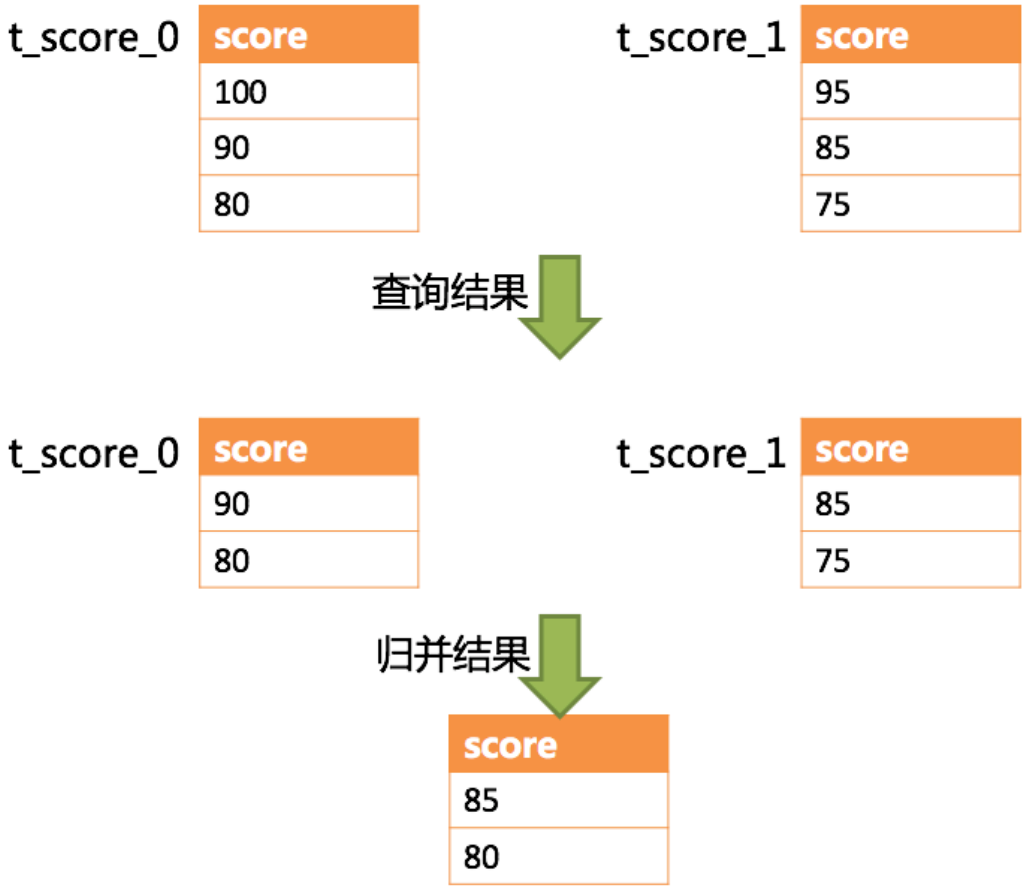
自增主键补列时，如果使用占位符的方式书写 SQL，则只需要改写参数列表即可，无需改写 SQL 本身。

分页修正

从多个数据库获取分页数据与单数据库的场景是不同的。假设每 10 条数据为一页，取第 2 页数据。在分片环境下获取 LIMIT 10, 10，归并之后再根据排序条件取出前 10 条数据是不正确的。举例说明，若 SQL 为：

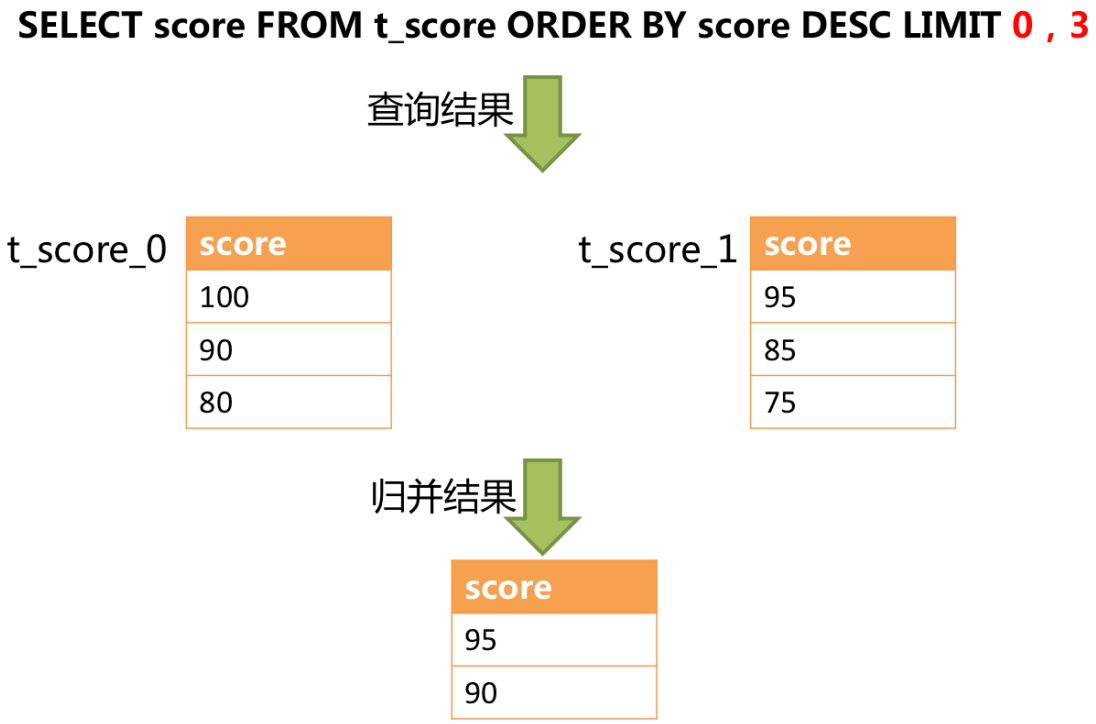
```
SELECT score FROM t_score ORDER BY score DESC LIMIT 1, 2;
```

下图展示了不进行 SQL 的改写的分页执行结果。



通过图中所示，想要取得两个表中共同的按照分数排序的第 2 条和第 3 条数据，应该是 95 和 90。由于执行的 SQL 只能从每个表中获取第 2 条和第 3 条数据，即从 t_score_0 表中获取的是 90 和 80；从 t_score_1 表中获取的是 85 和 75。因此进行结果归并时，只能从获取的 90，80，85 和 75 之中进行归并，那么结果归并无论怎么实现，都不可能获得正确的结果。

正确的做法是将分页条件改写为 LIMIT 0，3，取出所有前两页数据，再结合排序条件计算出正确的数据。下图展示了进行 SQL 改写之后的分页执行结果。



越获取偏移量位置靠后数据，使用 LIMIT 分页方式的效率就越低。有很多方法可以避免使用 LIMIT 进行分页。比如构建行记录数量与行偏移量的二级索引，或使用上次分页数据结尾 ID 作为下次查询条件的分页方式等。

分页信息修正时，如果使用占位符的方式书写 SQL，则只需要改写参数列表即可，无需改写 SQL 本身。

批量拆分

在使用批量插入的 SQL 时，如果插入的数据是跨分片的，那么需要对 SQL 进行改写来防止将多余的数据写入到数据库中。插入操作与查询操作的不同之处在于，查询语句中即使用了不存在于当前分片的分片键，也不会对数据产生影响；而插入操作则必须将多余的分片键删除。举例说明，如下 SQL：

```
INSERT INTO t_order (order_id, xxx) VALUES (1, 'xxx'), (2, 'xxx'), (3, 'xxx');
```

假设数据库仍然是按照 order_id 的奇偶值分为两片的，仅将这条 SQL 中的表名进行修改，然后发送至数据库完成 SQL 的执行，则两个分片都会写入相同的记录。虽然只有符合分片查询条件的数据才能够被查询语句取出，但存在冗余数据的实现方案并不合理。因此需要将 SQL 改写为：

```
INSERT INTO t_order_0 (order_id, xxx) VALUES (2, 'xxx');
INSERT INTO t_order_1 (order_id, xxx) VALUES (1, 'xxx'), (3, 'xxx');
```

使用 IN 的查询与批量插入的情况相似，不过 IN 操作并不会导致数据查询结果错误。通过对 IN 查询的改写，可以进一步的提升查询性能。如以下 SQL：

```
SELECT * FROM t_order WHERE order_id IN (1, 2, 3);
```

改写为：

```
SELECT * FROM t_order_0 WHERE order_id IN (2);
SELECT * FROM t_order_1 WHERE order_id IN (1, 3);
```

可以进一步的提升查询性能。ShardingSphere 暂时还未实现此改写策略，目前的改写结果是：

```
SELECT * FROM t_order_0 WHERE order_id IN (1, 2, 3);
SELECT * FROM t_order_1 WHERE order_id IN (1, 2, 3);
```

虽然 SQL 的执行结果是正确的，但并未达到最优的查询效率。

优化改写

优化改写的目的是在不影响查询正确性的情况下，对性能进行提升的有效手段。它分为单节点优化和流式归并优化。

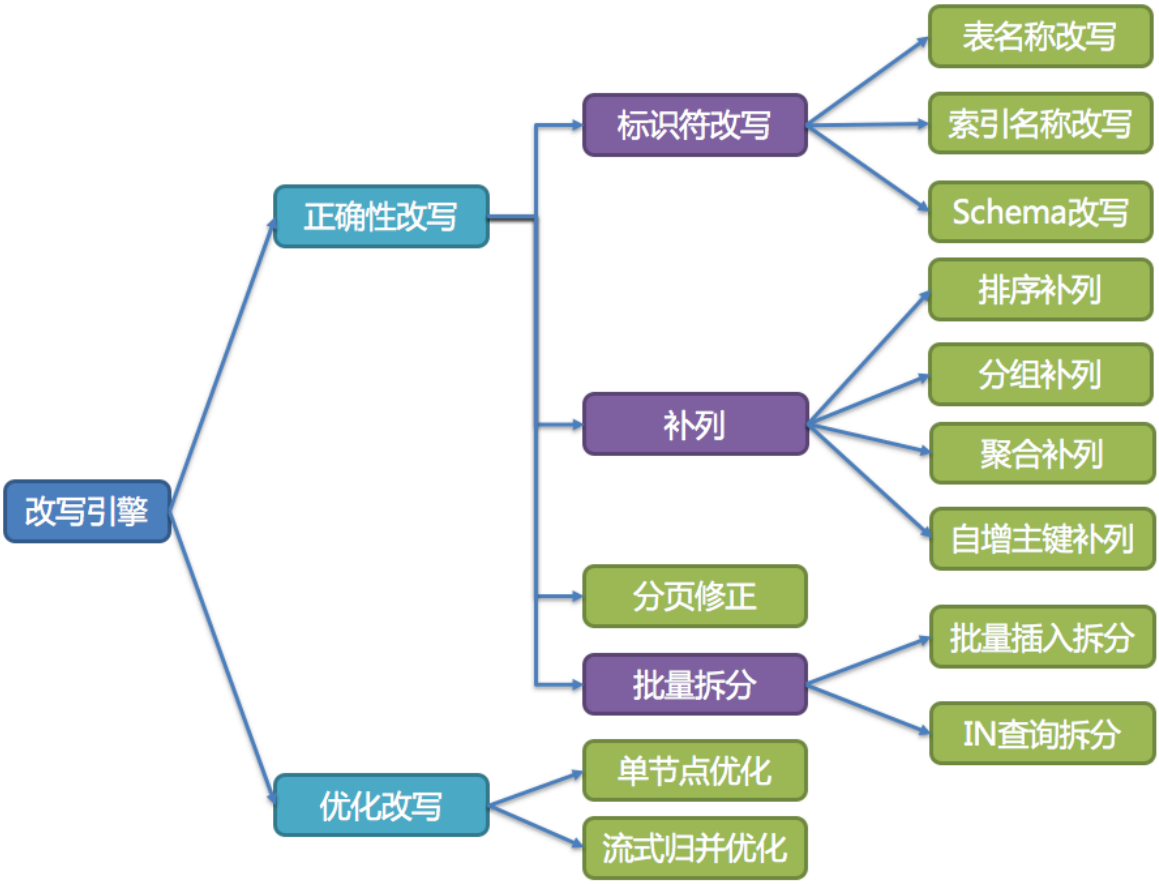
单节点优化

路由至单节点的 SQL，则无需优化改写。当获得一次查询的路由结果后，如果是路由至唯一的数据节点，则无需涉及到结果归并。因此补列和分页信息等改写都没有必要进行。尤其是分页信息的改写，无需将数据从第 1 条开始取，大量的降低了对数据库的压力，并且节省了网络带宽的无谓消耗。

流式归并优化

它仅为包含 `GROUP BY` 的 SQL 增加 `ORDER BY` 以及和分组项相同的排序项和排序顺序，用于将内存归并转化为流式归并。在结果归并的部分中，将对流式归并和内存归并进行详细说明。

改写引擎的整体结构划分如下图所示。



12.4.10 执行引擎

ShardingSphere 采用一套自动化的执行引擎，负责将路由和改写完成之后的真实 SQL 安全且高效发送到底层数据源执行。它不是简单地将 SQL 通过 JDBC 直接发送至数据源执行；也并非直接将执行请求放入线程池去并发执行。它更关注平衡数据源连接创建以及内存占用所产生的消耗，以及最大限度地合理利用并发等问题。执行引擎的目标是自动化的平衡资源控制与执行效率。

连接模式

从资源控制的角度看，业务方访问数据库的连接数量应当有所限制。它能够有效地防止某一业务操作过多的占用资源，从而将数据库连接的资源耗尽，以致于影响其他业务的正常访问。特别是在一个数据库实例中存在较多分表的情况下，一条不包含分片键的逻辑 SQL 将产生落在同库不同表的大量真实 SQL，如果每条真实 SQL 都占用一个独立的连接，那么一次查询无疑将会占用过多的资源。

从执行效率的角度看，为每个分片查询维持一个独立的数据库连接，可以更加有效的利用多线程来提升执行效率。为每个数据库连接开启独立的线程，可以将 I/O 所产生的消耗并行处理。为每个分片维持一个独立的数据库连接，还能够避免过早的将查询结果数据加载至内存。独立的数据库连接，能够持有查询结果集游标位置的引用，在需要获取相应数据时移动游标即可。

以结果集游标下移进行结果归并的方式，称之为流式归并，它无需将结果数据全数加载至内存，可以有效的节省内存资源，进而减少垃圾回收的频次。当无法保证每个分片查询持有一个独立数据库连接时，则

需要在复用该数据库连接获取下一张分表的查询结果集之前，将当前的查询结果集全数加载至内存。因此，即使可以采用流式归并，在此场景下也将退化为内存归并。

一方面是对数据库连接资源的控制保护，一方面是采用更优的归并模式达到对中间件内存资源的节省，如何处理好两者之间的关系，是 ShardingSphere 执行引擎需要解决的问题。具体来说，如果一条 SQL 在经过 ShardingSphere 的分片后，需要操作某数据库实例下的 200 张表。那么，是选择创建 200 个连接并行执行，还是选择创建一个连接串行执行呢？效率与资源控制又应该如何抉择呢？

针对上述场景，ShardingSphere 提供了一种解决思路。它提出了连接模式（Connection Mode）的概念，将其划分为内存限制模式（MEMORY_STRICTLY）和连接限制模式（CONNECTION_STRICTLY）这两种类型。

内存限制模式

使用此模式的前提是，ShardingSphere 对一次操作所耗费的数据库连接数量不做限制。如果实际执行的 SQL 需要对某数据库实例中的 200 张表做操作，则对每张表创建一个新的数据库连接，并通过多线程的方式并发处理，以达成执行效率最大化。并且在 SQL 满足条件情况下，优先选择流式归并，以防止出现内存溢出或避免频繁垃圾回收情况。

连接限制模式

使用此模式的前提是，ShardingSphere 严格控制对一次操作所耗费的数据库连接数量。如果实际执行的 SQL 需要对某数据库实例中的 200 张表做操作，那么只会创建唯一的数据库连接，并对其 200 张表串行处理。如果一次操作中的分片散落在不同的数据库，仍然采用多线程处理对不同库的操作，但每个库的每次操作仍然只创建一个唯一的数据库连接。这样即可以防止对一次请求对数据库连接占用过多所带来的问题。该模式始终选择内存归并。

内存限制模式适用于 OLAP 操作，可以通过放宽对数据库连接的限制提升系统吞吐量；连接限制模式适用于 OLTP 操作，OLTP 通常带有分片键，会路由到单一的分片，因此严格控制数据库连接，以保证在线系统数据库资源能够被更多的应用所使用，是明智的选择。

自动化执行引擎

ShardingSphere 最初将使用何种模式的决定权交由用户配置，让开发者依据自己业务的实际场景需求选择使用内存限制模式或连接限制模式。

这种解决方案将两难的选择的决定权交由用户，使得用户必须要了解这两种模式的利弊，并依据业务场景需求进行选择。这无疑增加了用户对 ShardingSphere 的学习和使用的成本，并非最优方案。

这种一分为二的处理方案，将两种模式的切换交由静态的初始化配置，是缺乏灵活应对能力的。在实际的使用场景中，面对不同 SQL 以及占位符参数，每次的路由结果是不同的。这就意味着某些操作可能需要使用内存归并，而某些操作则可能选择流式归并更优，具体采用哪种方式不应该由用户在 ShardingSphere 启动之前配置好，而是应该根据 SQL 和占位符参数的场景，来动态的决定连接模式。

为了降低用户的使用成本以及连接模式动态化这两个问题，ShardingSphere 提炼出自动化执行引擎的思路，在其内部消化了连接模式概念。用户无需了解所谓的内存限制模式和连接限制模式是什么，而是交由执行引擎根据当前场景自动选择最优的执行方案。

自动化执行引擎将连接模式的选择粒度细化至每一次 SQL 的操作。针对每次 SQL 请求，自动化执行引擎都将根据其路由结果，进行实时的演算和权衡，并自主地采用恰当的连接模式执行，以达到资源控制和效率的最优平衡。针对自动化的执行引擎，用户只需配置 maxConnectionSizePerQuery 即可，该参数表示一次查询时每个数据库所允许使用的最大连接数。

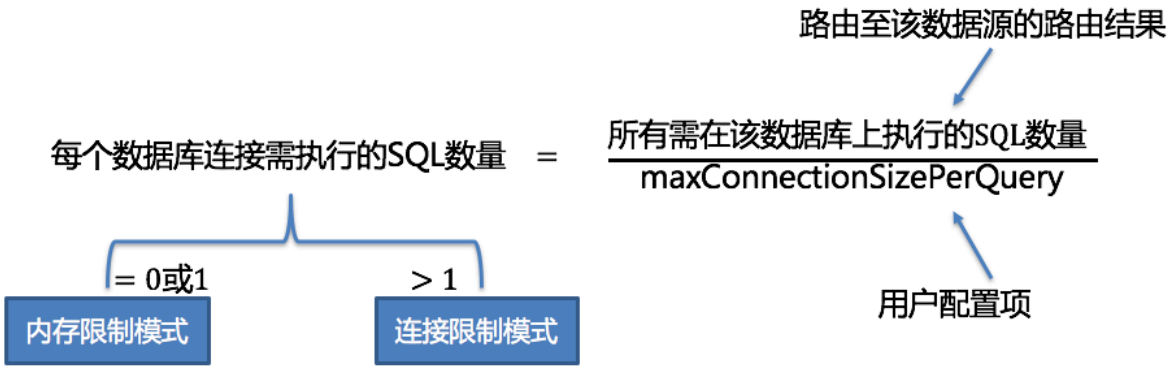
执行引擎分为准备和执行两个阶段。

准备阶段

顾名思义，此阶段用于准备执行的数据。它分为结果集分组和执行单元创建两个步骤。

结果集分组是实现内化连接模式概念的关键。执行引擎根据 maxConnectionSizePerQuery 配置项，结合当前路由结果，选择恰当的连接模式。具体步骤如下：

1. 将 SQL 的路由结果按照数据源的名称进行分组。
2. 通过下图的公式，可以获得每个数据库实例在 `maxConnectionSizePerQuery` 的允许范围内，每个连接需要执行的 SQL 路由结果组，并计算出本次请求的最优连接模式。

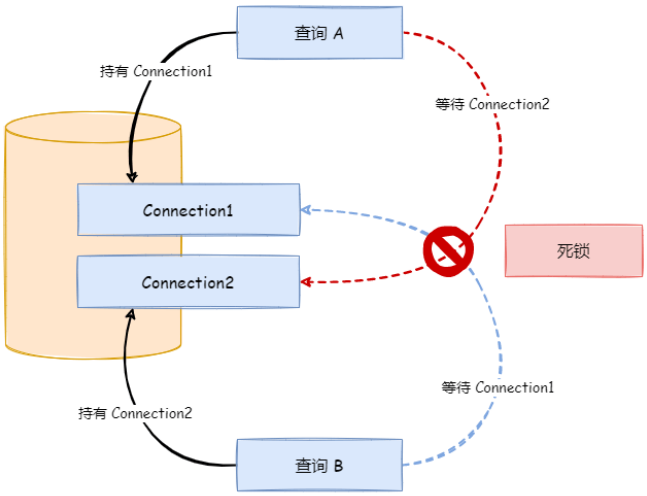


在 maxConnectionSizePerQuery 允许的范围内，当一个连接需要执行的请求数量大于 1 时，意味着当前的数据库连接无法持有相应的数据结果集，则必须采用内存归并；反之，当一个连接需要执行的请求数量等于 1 时，意味着当前的数据库连接可以持有相应的数据结果集，则可以采用流式归并。

每一次的连接模式的选择，是针对每一个物理数据库的。也就是说，在同一次查询中，如果路由至一个以上的数据库，每个数据库的连接模式不一定一样，它们可能是混合存在的形态。

通过上一步骤获得的路由分组结果创建执行的单元。当数据源使用数据库连接池等控制数据库连接数量的技术时，在获取数据库连接时，如果不妥善处理并发，则有一定几率发生死锁。在多个请求相互等待对方释放数据库连接资源时，将会产生饥饿等待，造成交叉的死锁问题。

举例说明，假设一次查询需要在某一数据源上获取两个数据库连接，并路由至同一个数据库的两个分表查询。则有可能出现查询 A 已获取到该数据源的 1 个数据库连接，并等待获取另一个数据库连接；而查询 B 也已经在该数据源上获取到的一个数据库连接，并同样等待另一个数据库连接的获取。如果数据库连接池的允许最大连接数是 2，那么这 2 个查询请求将永久的等待下去。下图描绘了死锁的情况。



ShardingSphere 为了避免死锁的出现，在获取数据库连接时进行了同步处理。它在创建执行单元时，以原子性的方式一次性获取本次 SQL 请求所需的全部数据库连接，杜绝了每次查询请求获取到部分资源的可能。由于对数据库的操作非常频繁，每次获取数据库连接时时都进行锁定，会降低 ShardingSphere 的并发。因此，ShardingSphere 在这里进行了 2 点优化：

1. 避免锁定一次性只需要获取 1 个数据库连接的操作。因为每次仅需要获取 1 个连接，则不会发生两个请求相互等待的场景，无需锁定。对于大部分 OLTP 的操作，都是使用分片键路由至唯一的数据节点，这会使得系统变为完全无锁的状态，进一步提升了并发效率。除了路由至单分片的情况，读写分离也在此范畴之内。
2. 仅针对内存限制模式时才进行资源锁定。在使用连接限制模式时，所有的查询结果集将在装载至内存之后释放掉数据库连接资源，因此不会产生死锁等待的问题。

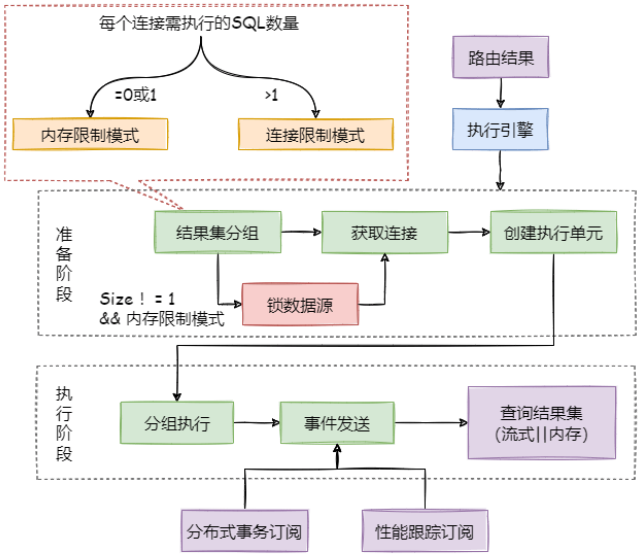
执行阶段

该阶段用于真正的执行 SQL，它分为分组执行和归并结果集生成两个步骤。

分组执行将准备执行阶段生成的执行单元分组下发至底层并发执行引擎，并针对执行过程中的每个关键步骤发送事件。如：执行开始事件、执行成功事件以及执行失败事件。执行引擎仅关注事件的发送，它并不关心事件的订阅者。ShardingSphere 的其他模块，如：分布式事务、调用链路追踪等，会订阅感兴趣的事件，并进行相应的处理。

ShardingSphere 通过在执行准备阶段的获取的连接模式，生成内存归并结果集或流式归并结果集，并将其传递至结果归并引擎，以进行下一步的工作。

执行引擎的整体结构划分如下图所示。



12.4.11 归并引擎

将从各个数据节点获取的多数据结果集，组合成为一个结果集并正确的返回至请求客户端，称为结果归并。

ShardingSphere 支持的结果归并从功能上分为遍历、排序、分组、分页和聚合 5 种类型，它们是组合而非互斥的关系。从结构划分，可分为流式归并、内存归并和装饰者归并。流式归并和内存归并是互斥的，装饰者归并可以在流式归并和内存归并之上做进一步的处理。

由于从数据库中返回的结果集是逐条返回的，并不需要将所有的数据一次性加载至内存中，因此，在进行结果归并时，沿用数据库返回结果集的方式进行归并，能够极大减少内存的消耗，是归并方式的优先选择。

流式归并是指每一次从结果集中获取到的数据，都能够通过逐条获取的方式返回正确的单条数据，它与数据库原生的返回结果集的方式最为契合。遍历、排序以及流式分组都属于流式归并的一种。

内存归并则是需要将结果集的所有数据都遍历并存储在内存中，再通过统一的分组、排序以及聚合等计算之后，再将其封装成为逐条访问的数据结果集返回。

装饰者归并是对所有的结果集归并进行统一的功能增强，目前装饰者归并有分页归并和聚合归并这 2 种类型。

遍历归并

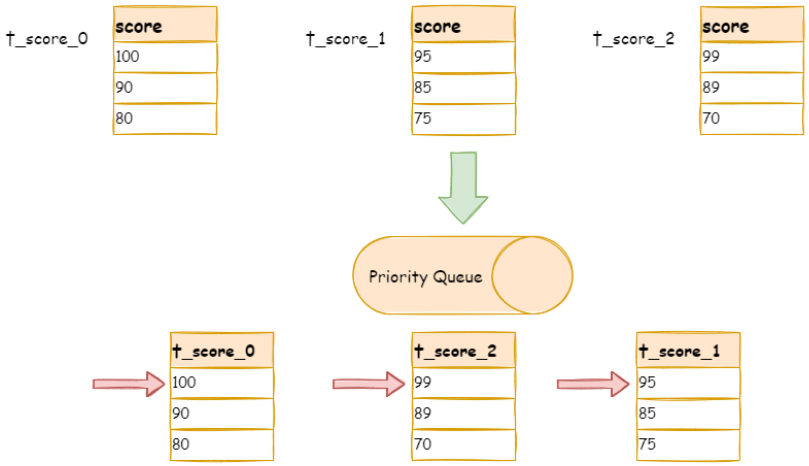
它是最为简单的归并方式。只需将多个数据结果集合并为一个单向链表即可。在遍历完成链表中当前数据结果集之后，将链表元素后移一位，继续遍历下一个数据结果集即可。

排序归并

由于在 SQL 中存在 `ORDER BY` 语句，因此每个数据结果集自身是有序的，因此只需要将数据结果集当前游标指向的数据值进行排序即可。这相当于对多个有序的数组进行排序，归并排序是最适合此场景的排序算法。

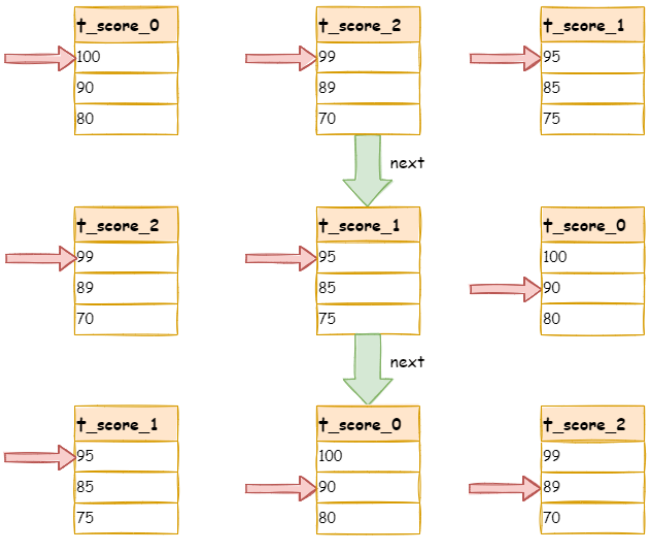
ShardingSphere 在对排序的查询进行归并时，将每个结果集的当前数据值进行比较（通过实现 Java 的 Comparable 接口完成），并将其放入优先级队列。每次获取下一条数据时，只需将队列顶端结果集的游标下移，并根据新游标重新进入优先级排序队列找到自己的位置即可。

通过一个例子来说明 ShardingSphere 的排序归并，下图是一个通过分数进行排序的示例图。图中展示了 3 张表返回的数据结果集，每个数据结果集已经根据分数排序完毕，但是 3 个数据结果集之间是无序的。将 3 个数据结果集的当前游标指向的数据值进行排序，并放入优先级队列，t_score_0 的第一个数据值最大，t_score_2 的第一个数据值次之，t_score_1 的第一个数据值最小，因此优先级队列根据 t_score_0，t_score_2 和 t_score_1 的方式排序队列。



下图则展现了进行 next 调用的时候，排序归并是如何进行的。通过图中我们可以看到，当进行第一次 next 调用时，排在队列首位的 t_score_0 将会被弹出队列，并且将当前游标指向的数据值（也就是 100）返回至查询客户端，并且将游标下移一位之后，重新放入优先级队列。而优先级队列也会根据 t_score_0 的当前数据结果集指向游标的数据值（这里是 90）进行排序，根据当前数值，t_score_0 排列在队列的最后一位。之前队列中排名第二的 t_score_2 的数据结果集则自动排在了队列首位。

在进行第二次 next 时，只需要将目前排列在队列首位的 t_score_2 弹出队列，并且将其数据结果集游标指向的值返回至客户端，并下移游标，继续加入队列排队，以此类推。当一个结果集中已经没有数据了，则无需再次加入队列。



可以看到，对于每个数据结果集中的数据有序，而多数据结果集整体无序的情况下，ShardingSphere 无需将所有的数据都加载至内存即可排序。它使用的是流式归并的方式，每次 next 仅获取唯一正确的一条数据，极大的节省了内存的消耗。

从另一个角度来说，ShardingSphere 的排序归并，是在维护数据结果集的纵轴和横轴这两个维度的有序性。纵轴是指每个数据结果集本身，它是天然有序的，它通过包含 ORDER BY 的 SQL 所获取。横轴是指每个数据结果集当前游标所指向的值，它需要通过优先级队列来维护其正确顺序。每一次数据结果集当前游标的下移，都需要将该数据结果集重新放入优先级队列排序，而只有排列在队列首位的数据结果集才可能发生游标下移的操作。

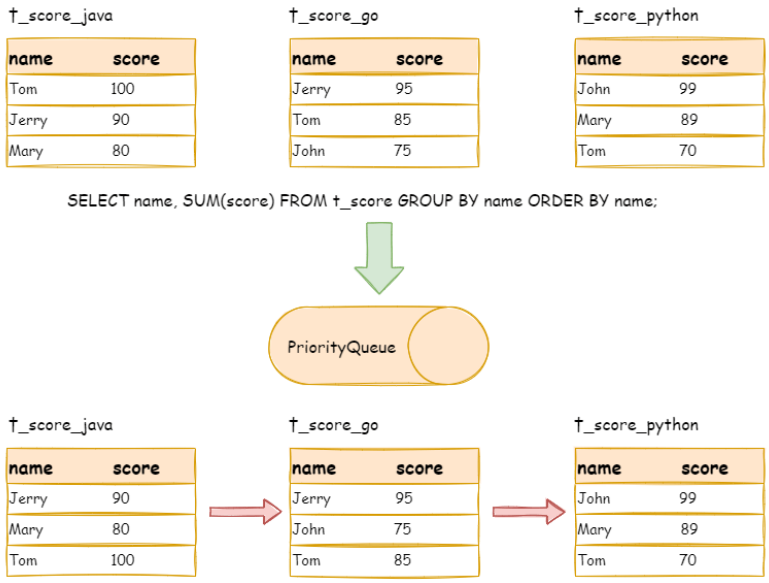
分组归并

分组归并的情况最为复杂，它分为流式分组归并和内存分组归并。流式分组归并要求 SQL 的排序项与分组项的字段以及排序类型（ASC 或 DESC）必须保持一致，否则只能通过内存归并才能保证其数据的正确性。

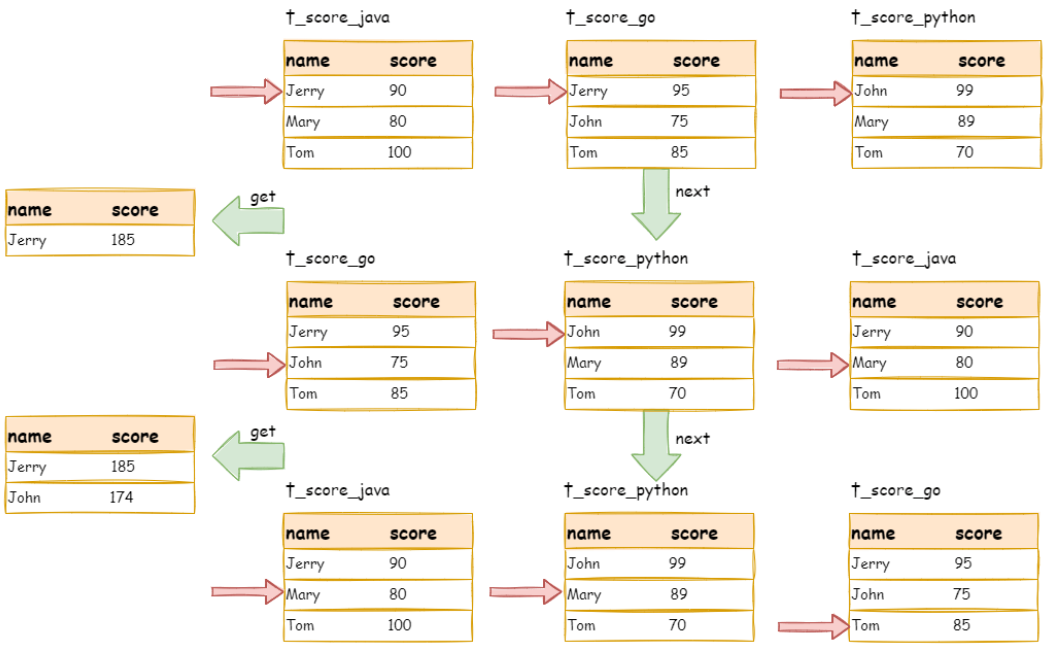
举例说明，假设根据科目分片，表结构中包含考生的姓名（为了简单起见，不考虑重名的情况）和分数。通过 SQL 获取每位考生的总分，可通过如下 SQL：

```
SELECT name, SUM(score) FROM t_score GROUP BY name ORDER BY name;
```

在分组项与排序项完全一致的情况下，取得的数据是连续的，分组所需的数据全数存在于各个数据结果集的当前游标所指向的数据值，因此可以采用流式归并。如下图所示。



进行归并时，逻辑与排序归并类似。下图展现了进行 next 调用的时候，流式分组归并是如何进行的。



通过图中我们可以看到，当进行第一次 next 调用时，排在队列首位的 t_score_java 将会被弹出队列，并且将分组值同为“Jerry”的其他结果集中的数据一同弹出队列。在获取了所有的姓名为“Jerry”的同学的分数之后，进行累加操作，那么，在第一次 next 调用结束后，取出的结果集是“Jerry”的分数总和。与此同时，所有的数据结果集中的游标都将下移至数据值“Jerry”的下一个不同的数据值，并且根据数据结果集当前游标指向的值进行重排序。因此，包含名字顺着第二位的“John”的相关数据结果集则排在的队列的前列。

流式分组归并与排序归并的区别仅仅在于两点：

1. 它会一次性的将多个数据结果集中的分组项相同的数据全数取出。
2. 它需要根据聚合函数的类型进行聚合计算。

对于分组项与排序项不一致的情况，由于需要获取分组的相关的数据值并非连续的，因此无法使用流式归并，需要将所有的结果集数据加载至内存中进行分组和聚合。例如，若通过以下 SQL 获取每位考生的总分并按照分数从高至低排序：

```
SELECT name, SUM(score) FROM t_score GROUP BY name ORDER BY score DESC;
```

那么各个数据结果集中取出的数据与排序归并那张图的上半部分的表结构的原始数据一致，是无法进行流式归并的。

当 SQL 中只包含分组语句时，根据不同数据库的实现，其排序的顺序不一定与分组顺序一致。但由于排序语句的缺失，则表示此 SQL 并不在意排序顺序。因此，ShardingSphere 通过 SQL 优化的改写，自动增加与分组项一致的排序项，使其能够从消耗内存的内存分组归并方式转化为流式分组归并方案。

聚合归并

无论是流式分组归并还是内存分组归并，对聚合函数的处理都是一致的。除了分组的 SQL 之外，不进行分组的 SQL 也可以使用聚合函数。因此，聚合归并是在之前介绍的归并类的之上追加的归并能力，即装饰者模式。聚合函数可以归类为比较、累加和求平均值这 3 种类型。

比较类型的聚合函数是指 `MAX` 和 `MIN`。它们需要对每一个同组的结果集数据进行比较，并且直接返回其最大或最小值即可。

累加类型的聚合函数是指 `SUM` 和 `COUNT`。它们需要将每一个同组的结果集数据进行累加。

求平均值的聚合函数只有 `AVG`。它必须通过 SQL 改写的 `SUM` 和 `COUNT` 进行计算，相关内容已在 SQL 改写的内容中涵盖，不再赘述。

分页归并

上文所述的所有归并类型都可能进行分页。分页也是追加在其他归并类型之上的装饰器，ShardingSphere 通过装饰者模式来增加对数据结果集进行分页的能力。分页归并负责将无需获取的数据过滤掉。

ShardingSphere 的分页功能比较容易让使用者误解，用户通常认为分页归并会占用大量内存。在分布式的场景中，将 `LIMIT 10000000, 10` 改写为 `LIMIT 0, 10000010`，才能保证其数据的正确性。用户非常容易产生 ShardingSphere 会将大量无意义的数据加载至内存中，造成内存溢出风险的错觉。其实，通过流式归并的原理可知，会将数据全部加载到内存中的只有内存分组归并这一种情况。而通常来说，进行 OLAP 的分组 SQL，不会产生大量的结果数据，它更多的用于大量的计算，以及少量结果产出的场景。除了内存分组归并这种情况之外，其他情况都通过流式归并获取数据结果集，因此 ShardingSphere 会通过结果集的 next 方法将无需取出的数据全部跳过，并不会将其存入内存。

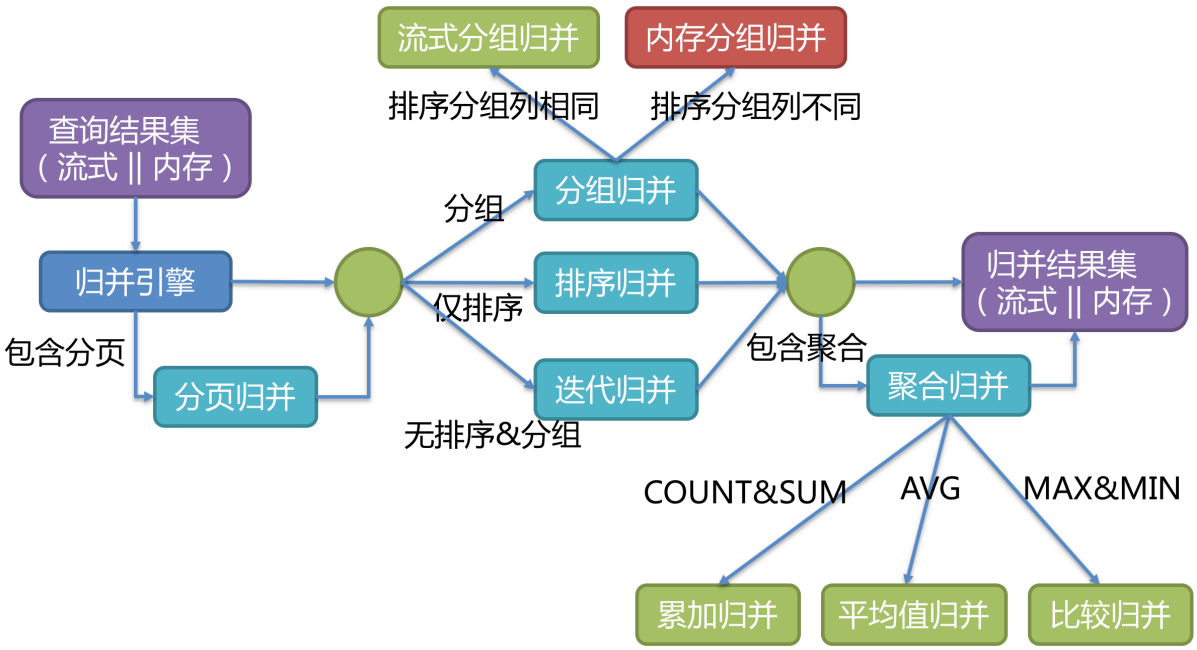
但同时需要注意的是，由于排序的需要，大量的数据仍然需要传输到 ShardingSphere 的内存空间。因此，采用 LIMIT 这种方式分页，并非最佳实践。由于 LIMIT 并不能通过索引查询数据，因此如果可以保证 ID 的连续性，通过 ID 进行分页是比较好的解决方案，例如：

```
SELECT * FROM t_order WHERE id > 100000 AND id <= 100010 ORDER BY id;
```

或通过记录上次查询结果的最后一条记录的 ID 进行下一页的查询，例如：

```
SELECT * FROM t_order WHERE id > 10000000 LIMIT 10;
```

归并引擎的整体结构划分如下图。



12.5 分布式事务

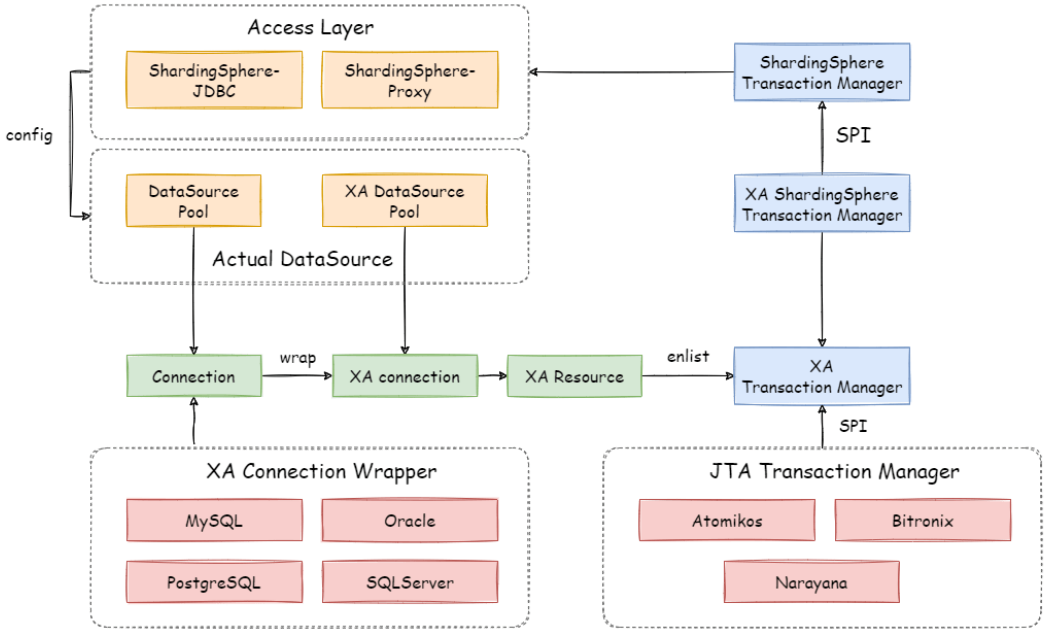
12.5.1 导览

本小节主要介绍 Apache ShardingSphere 分布式事务的实现原理

- 基于 XA 协议的两阶段事务
- 基于 Seata 的柔性事务

12.5.2 XA 事务

`XAShardingSphereTransactionManager` 为 Apache ShardingSphere 的分布式事务的 XA 实现类。它主要负责对多数据源进行管理和适配，并且将相应事务的开启、提交和回滚操作委托给具体的 XA 事务管理器。



开启全局事务

收到接入端的 `set autoCommit=0` 时，`XAShardingSphereTransactionManager` 将调用具体的 XA 事务管理器开启 XA 全局事务，以 XID 的形式进行标记。

执行真实分片 SQL

`XAShardingSphereTransactionManager` 将数据库连接所对应的 XAResource 注册到当前 XA 事务中之后，事务管理器会在此阶段发送 `XAResource.start` 命令至数据库。数据库在收到 `XAResource.end` 命令之前的所有 SQL 操作，会被标记为 XA 事务。

例如:

```
XAResource1.start              ## Enlist 阶段执行
statement.execute("sql1");     ## 模拟执行一个分片 SQL1
statement.execute("sql2");     ## 模拟执行一个分片 SQL2
XAResource1.end                ## 提交阶段执行
```

示例中的 `sql1` 和 `sql2` 将会被标记为 XA 事务。

提交或回滚事务

`XAShardingSphereTransactionManager` 在接收到接入端的提交命令后，会委托实际的 XA 事务管理进行提交动作，事务管理器将收集到的当前线程中所有注册的 XAResource，并发送 `XAResource.end` 指令，用以标记此 XA 事务边界。接着会依次发送 `prepare` 指令，收集所有参与 XAResource 投票。若所有 XAResource 的反馈结果均为正确，则调用 `commit` 指令进行最终提交；若有任意 XAResource 的反馈结果不正确，则调用 `rollback` 指令进行回滚。在事务管理器发出提交指令后，任何 XAResource 产生的异常都会通过恢复日志进行重试，以保证提交阶段的操作原子性，和数据强一致性。

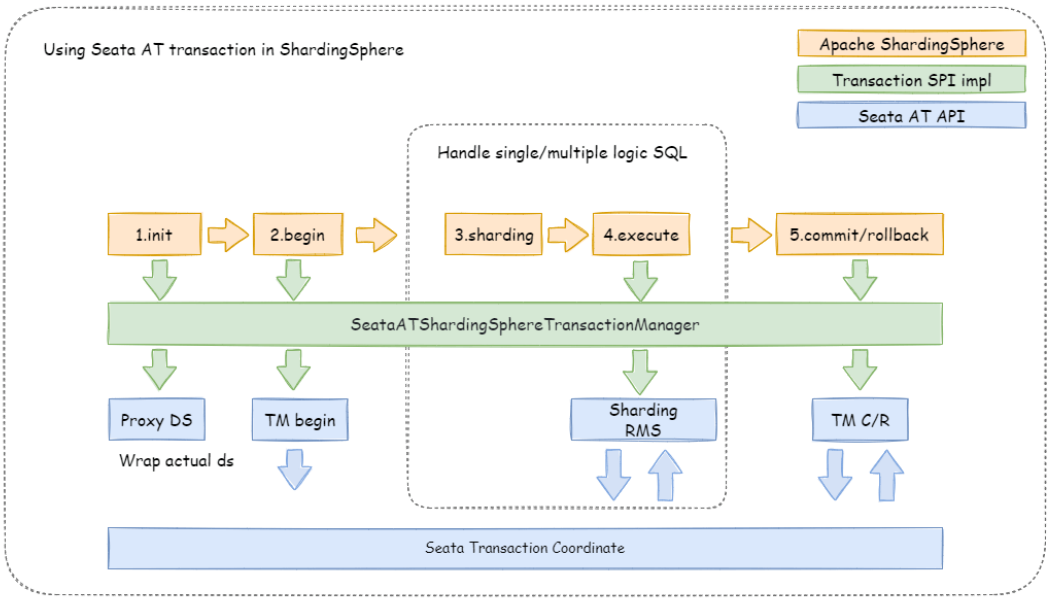
例如:

```
XAResource1.prepare           ## ack: yes
XAResource2.prepare           ## ack: yes
XAResource1.commit
XAResource2.commit

XAResource1.prepare           ## ack: yes
XAResource2.prepare           ## ack: no
XAResource1.rollback
XAResource2.rollback
```

12.5.3 Seata 柔性事务

整合 Seata AT 事务时，需要将 TM，RM 和 TC 的模型融入 Apache ShardingSphere 的分布式事务生态中。在数据库资源上，Seata 通过对接 `DataSource` 接口，让 JDBC 操作可以同 TC 进行远程通信。同样，Apache ShardingSphere 也是面向 `DataSource` 接口，对用户配置的数据源进行聚合。因此，将 `DataSource` 封装为基于 Seata 的 `DataSource` 后，就可以将 Seata AT 事务融入到 Apache ShardingSphere 的分片生态中。



引擎初始化

包含 Seata 柔性事务的应用启动时，用户配置的数据源会根据 `seata.conf` 的配置，适配为 Seata 事务所需的 `DataSourceProxy`，并且注册至 RM 中。

开启全局事务

TM 控制全局事务的边界，TM 通过向 TC 发送 Begin 指令，获取全局事务 ID，所有分支事务通过此全局事务 ID，参与到全局事务中；全局事务 ID 的上下文存放在当前线程变量中。

执行真实分片 SQL

处于 Seata 全局事务中的分片 SQL 通过 RM 生成 undo 快照，并且发送 `participate` 指令至 TC，加入到全局事务中。由于 Apache ShardingSphere 的分片物理 SQL 采取多线程方式执行，因此整合 Seata AT 事务时，需要在主线程和子线程间进行全局事务 ID 的上下文传递。

提交或回滚事务

提交 Seata 事务时，TM 会向 TC 发送全局事务的提交或回滚指令，TC 根据全局事务 ID 协调所有分支事务进行提交或回滚。

12.6 数据迁移

12.6.1 原理说明

目前的数据迁移解决方案为：使用一个全新的数据库集群作为迁移目标库。

这种实现方式有以下优点：

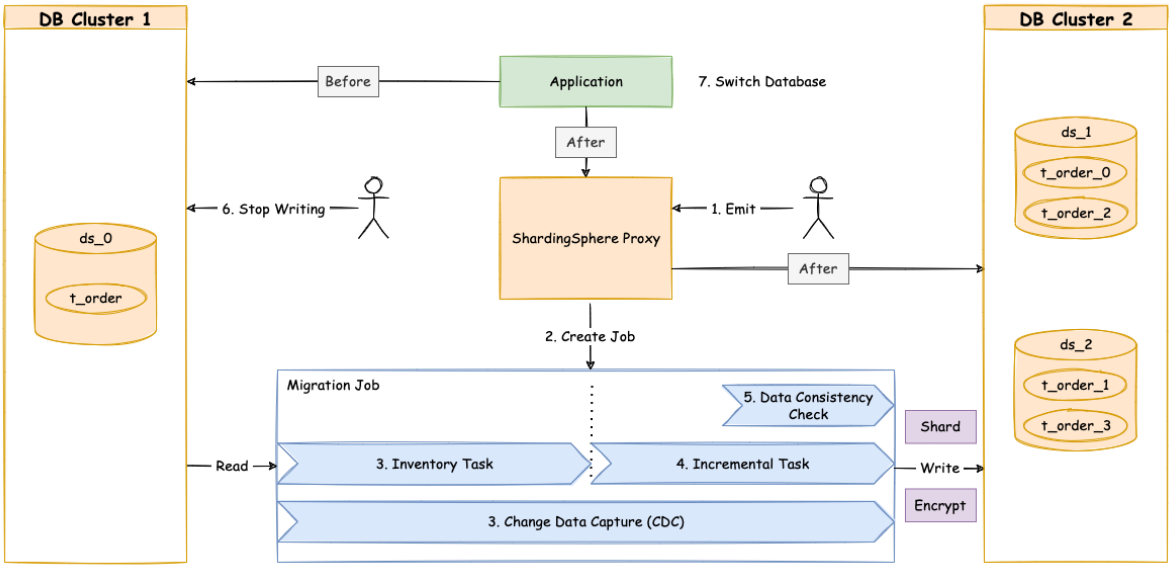
1. 迁移过程中，原始数据没有任何影响；
2. 迁移失败无风险；
3. 不受分片策略限制。

同时也存在一定的缺点：

1. 在一定时间内存在冗余服务器；
2. 所有数据都需要移动。

一次数据迁移包括以下几个主要阶段：

1. 准备阶段；
2. 存量数据迁移阶段；
3. 增量数据同步阶段；
4. 流量切换阶段。



12.6.2 执行阶段说明

准备阶段

在准备阶段，数据迁移模块会进行数据源连通性及权限的校验，同时进行存量数据的统计、日志位点的记录，最后根据数据量和用户设置的并行度，对任务进行分片。

存量数据迁移阶段

执行在准备阶段拆分好的存量数据迁移任务，存量迁移阶段采用 JDBC 查询的方式，直接从源端读取数据，基于配置的分片等规则写入到目标端。

增量数据同步阶段

由于存量数据迁移耗费的时间受到数据量和并行度等因素影响，此时需要对这段时间内业务新增的数据进行同步。不同的数据库使用的技术细节不同，但总体上均为基于复制协议或 WAL 日志实现的变更数据捕获功能。

- MySQL：订阅并解析 binlog；
- PostgreSQL：采用官方逻辑复制 test_decoding。

这些捕获的增量数据，同样会由数据迁移模块写入到新数据节点中。当增量数据基本同步完成时（由于业务系统未停止，增量数据是不断的），则进入流量切换阶段。

流量切换阶段

在此阶段，可能存在一定时间的业务只读窗口期，通过设置数据库只读、控制源头写流量等方式，让源端数据节点中的数据短暂静态，确保增量同步完全完成。

这个只读窗口期时长取决于用户是否需要对数据进行一致性校验以及数据量。一致性校验是独立的任务，支持单独启停，支持断点续传。

确认完成后，数据迁移完成。然后用户可以把读流量或者写流量切换到 Apache ShardingSphere。

12.6.3 相关参考

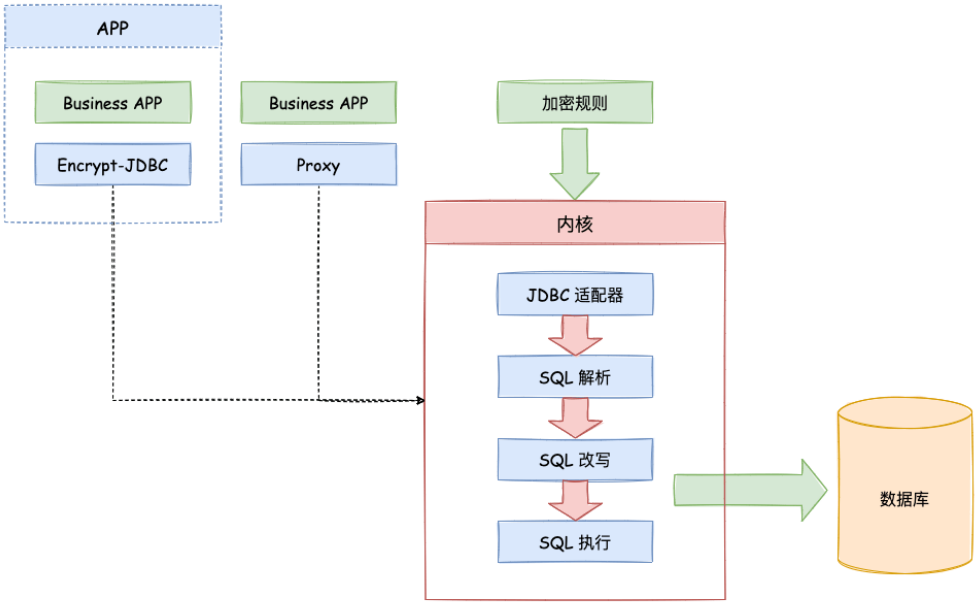
数据迁移的配置

12.7 数据加密

12.7.1 处理流程详解

Apache ShardingSphere 通过对用户输入的 SQL 进行解析，并依据用户提供的加密规则对 SQL 进行改写，从而实现对原文数据进行加密。在用户查询数据时，它仅从数据库中取出密文数据，并对其解密，最终将解密后的原始数据返回给用户。Apache ShardingSphere 自动化 & 透明化了数据加密过程，让用户无需关注数据加密的实现细节，像使用普通数据那样使用加密数据。

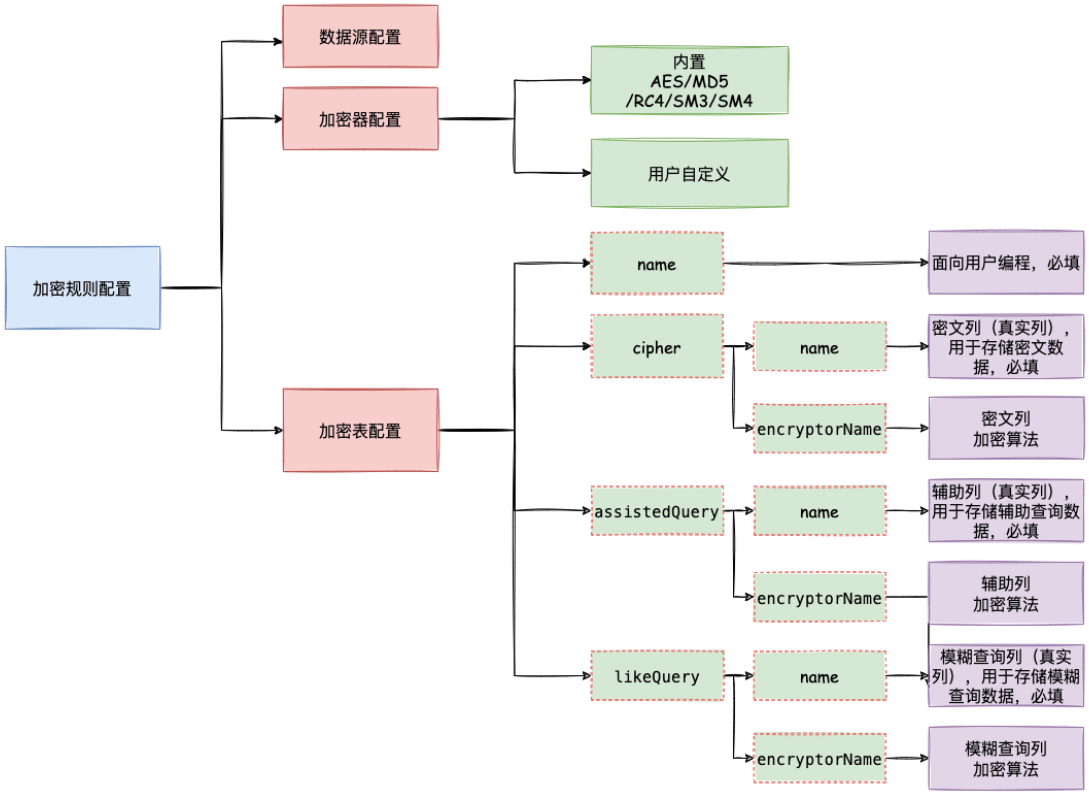
整体架构



加密模块将用户发起的 SQL 进行拦截，并通过 SQL 语法解析器进行解析、理解 SQL 行为，再依据用户传入的加密规则，找出需要加密的字段和所使用的加解密算法对目标字段进行加解密处理后，再与底层数据库进行交互。Apache ShardingSphere 会将用户请求的明文进行加密后存储到底层数据库；并在用户查询时，将密文从数据库中取出进行解密后返回给终端用户。通过屏蔽对数据的加密处理，使用户无需感知解析 SQL、数据加密、数据解密的处理过程，就像在使用普通数据一样使用加密数据。

加密规则

在详解整套流程之前，我们需要先了解下加密规则与配置，这是认识整套流程的基础。加密配置主要分为三部分：数据源配置，加密算法配置，加密表配置，其详情如下图所示：



数据源配置：指数据源配置。

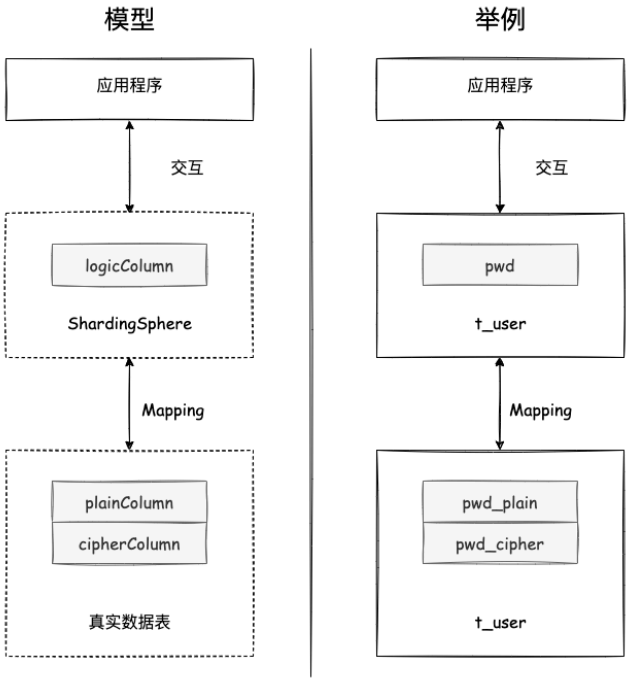
加密算法配置：指使用什么加密算法进行加解密。目前 ShardingSphere 内置了三种加解密算法：AES，MD5 和 RC4。用户还可以通过实现 ShardingSphere 提供的接口，自行实现一套加解密算法。

加密表配置：用于告诉 ShardingSphere 数据表里哪个列用于存储密文数据（cipherColumn）、使用什么算法加解密（encryptorName）、哪个列用于存储辅助查询数据（assistedQueryColumn）、使用什么算法加解密（assistedQueryEncryptorName）以及用户想使用哪个列进行 SQL 编写（logicColumn）。

> 如何理解 用户想使用哪个列进行 SQL 编写（logicColumn）？
>
> 我们可以从加密模块存在的意义来理解。加密模块最终目的是希望屏蔽底层对数据的加密处理，也就是说我们不希望用户知道数据是如何被加解密的、如何将密文数据存储到 cipherColumn，将辅助查询数据存储到 assistedQueryColumn。换句话说，我们不希望用户知道 cipherColumn 和 assistedQueryColumn 的存在和使用。所以，我们需要给用户提供一个概念意义上的列，这个列可以脱离底层数据库的真实列，它可以是数据库表里的一个真实列，也可以不是，从而使得用户可以随意改变底层数据库的 cipherColumn 和 assistedQueryColumn 的列名。只要用户的 SQL 面向这个逻辑列进行编写，并在加密规则里给出 logicColumn、cipherColumn、assistedQueryColumn 之间正确的映射关系即可。
>
> 为什么要这么做呢？答案在文章后面，即为了让已上线的业务能无缝、透明、安全地进行数据加密迁移。

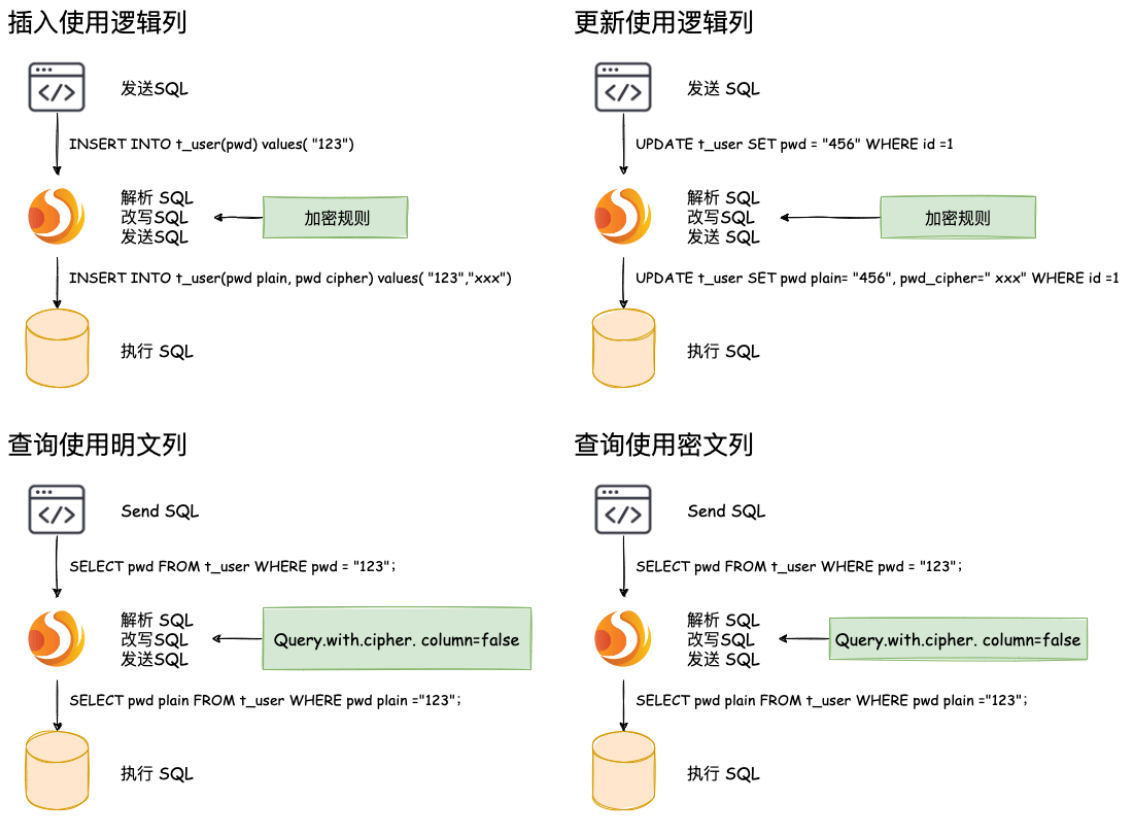
加密处理过程

举例说明，假如数据库里有一张表叫做 `t_user`，这张表里实际有两个字段 `pwd_cipher`，用于存放密文数据、`pwd_assisted_query`，用于存放辅助查询数据，同时定义 logicColumn 为 pwd。那么，用户在编写 SQL 时应该面向 logicColumn 进行编写，即 `INSERT INTO t_user SET pwd = '123'`。Apache ShardingSphere 接收到该 SQL，通过用户提供的加密配置，发现 pwd 是 logicColumn，于是便对逻辑列及其对应的明文数据进行加密处理。**Apache ShardingSphere 将面向用户的逻辑列与面向底层数据库的密文列进行了列名以及数据的加密映射转换**。如下图所示：



即依据用户提供的加密规则，将用户 SQL 与底层数据表结构割裂开来，使得用户的 SQL 编写不再依赖于真实的数据库表结构。而用户与底层数据库之间的衔接、映射、转换交由 Apache ShardingSphere 进行处理。

下方图片展示了使用加密模块进行增删改查时，其中的处理流程和转换逻辑，如下图所示。



12.7.2 解决方案详解

在了解了 Apache ShardingSphere 加密处理流程后，即可将加密配置、加密处理流程与实际场景进行结合。所有的设计开发都是为了解决业务场景遇到的痛点。那么面对之前提到的业务场景需求，又应该如何使用 Apache ShardingSphere 这把利器来满足业务需求呢?

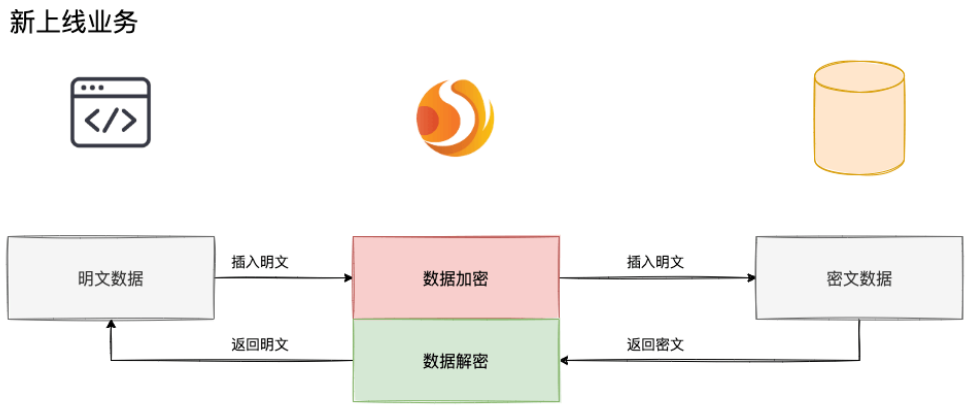
业务场景分析：新上线业务由于一切从零开始，不存在历史数据清洗问题，所以相对简单。

解决方案说明：选择合适的加密算法，如 AES 后，只需配置逻辑列（面向用户编写 SQL ）和密文列（数据表存密文数据）即可，**逻辑列和密文列可以相同也可以不同**。建议配置如下（YAML 格式展示）:

```
-!ENCRYPT
  encryptors:
    aes_encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
        digest-algorithm-name: SHA-1
  tables:
    t_user:
      columns:
        pwd:
          cipher:
            name: pwd_cipher
            encryptorName: aes_encryptor
```

```
        assistedQuery:
          name: pwd_assisted_query
          encryptorName: pwd_assisted_query_cipher
```

使用这套配置，Apache ShardingSphere 只需将 logicColumn 和 cipherColumn，assistedQueryColumn 进行转换，底层数据表不存储明文，只存储了密文，这也是安全审计部分的要求所在。整体处理流程如下图所示：



12.7.3 中间件加密服务优势

1. 自动化 & 透明化数据加密过程，用户无需关注加密中间实现细节。
2. 提供多种内置、第三方（AKS）的加密算法，用户仅需简单配置即可使用。
3. 提供加密算法 API 接口，用户可实现接口，从而使用自定义加密算法进行数据加密。
4. 支持切换不同的加密算法。

12.7.4 加密算法解析

Apache ShardingSphere 提供了加密算法用于数据加密，即 `EncryptAlgorithm`。

一方面，Apache ShardingSphere 为用户提供了内置的加解密实现类，用户只需进行配置即可使用；另一方面，为了满足用户不同场景的需求，我们还开放了相关加解密接口，用户可依据这两种类型的接口提供具体实现类。再进行简单配置，即可让 Apache ShardingSphere 调用用户自定义的加解密方案进行数据加密。

EncryptAlgorithm

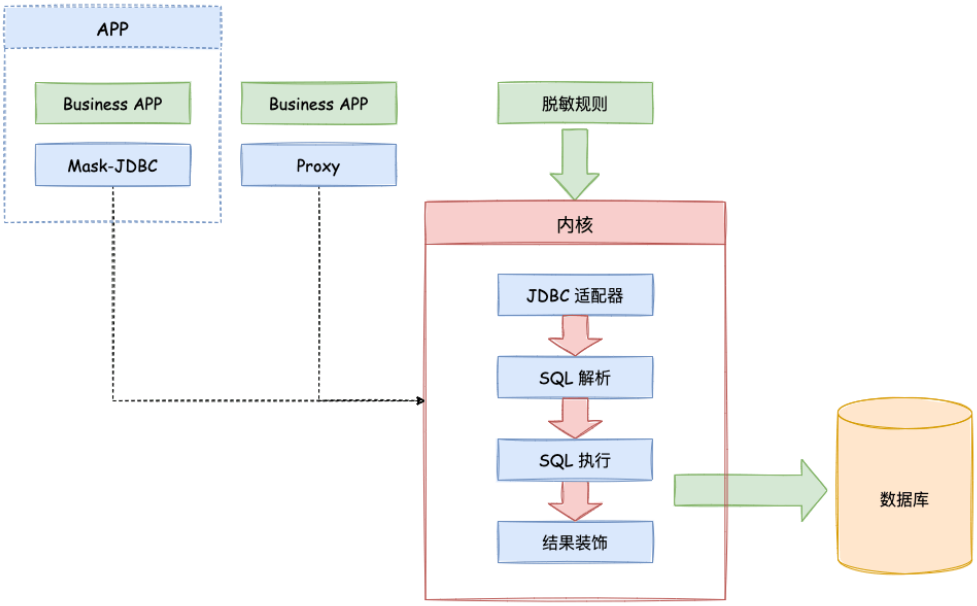
该解决方案通过提供 `encrypt()`，`decrypt()` 两种方法对需要加密的数据进行加解密。在用户进行 `INSERT`，`DELETE`，`UPDATE` 时，ShardingSphere 会按照用户配置，对 SQL 进行解析、改写、路由，并调用 `encrypt()` 将数据加密后存储到数据库，而在 `SELECT` 时，则调用 `decrypt()` 方法将从数据库中取出的加密数据进行逆向解密，最终将原始数据返回给用户。

当前，Apache ShardingSphere 针对这种类型的加密解决方案提供了三种具体实现类，分别是 MD5（不可逆），AES（可逆），RC4（可逆），用户只需配置即可使用这三种内置的方案。

12.8 数据脱敏

12.8.1 处理流程详解

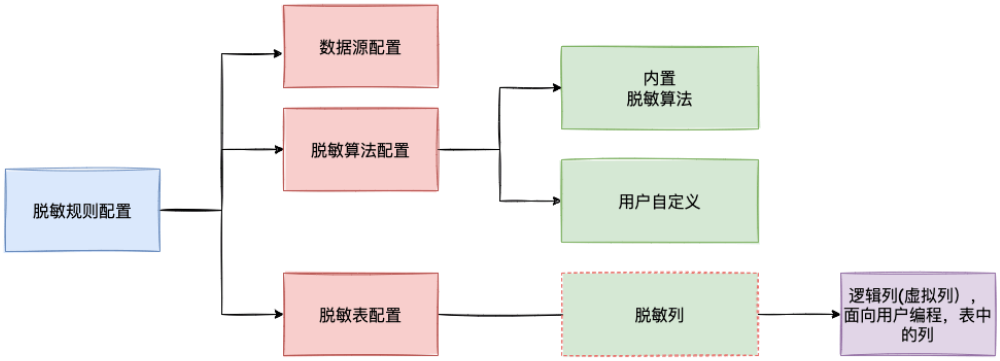
Apache ShardingSphere 通过对用户查询的 SQL 进行解析，并依据用户提供的脱敏规则对 SQL 执行结果进行装饰，从而实现对原文数据进行脱敏。### 整体架构



脱敏模块将用户发起的 SQL 进行拦截，并通过 SQL 语法解析器进行解析、执行，再依据用户传入的脱敏规则，找出需要脱敏的字段和所使用的脱敏算法对查询结果进行装饰，返回给客户端。

脱敏规则

在详解整套流程之前，我们需要先了解下脱敏规则与配置，这是认识整套流程的基础。脱敏配置主要分为三部分：数据源配置，脱敏算法配置，脱敏表配置：



数据源配置：指数据源配置。

脱敏算法配置：指使用什么脱敏算法。目前 ShardingSphere 内置了多种脱敏算法：MD5、KEEP_FIRST_N_LAST_M、KEEP_FROM_X_TO_Y、MASK_FIRST_N_LAST_M、MASK_FROM_X_TO_Y、MASK_BEFORE_SPECIAL_CHARS、MASK_AFTER_SPECIAL_CHARS 和 GENERIC_TABLE_RANDOM_REPLACE。用户还可以通过实现 ShardingSphere 提供的接口，自行实现一套脱敏算法。

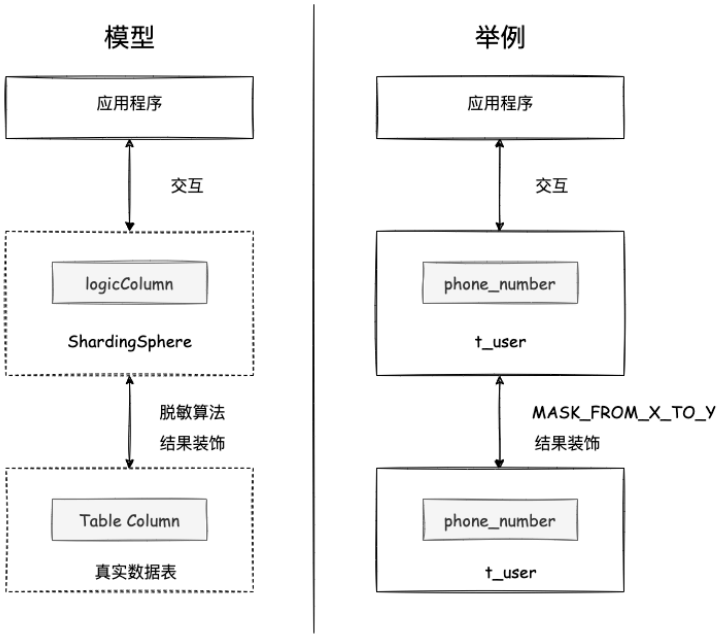
脱敏表配置：用于告诉 ShardingSphere 数据表里哪个列用于数据脱敏、使用什么算法脱敏。

脱敏规则创建即可生效

脱敏处理过程

举例说明，假如数据库里有一张表叫做 `t_user`，这张表里有一个字段 `phone_number` 使用 `MASK_FROM_X_TO_Y` 进行算法处理，Apache ShardingSphere 不会改变数据存储，只会按脱敏算法对结果进行装饰，从而达到脱敏的效果

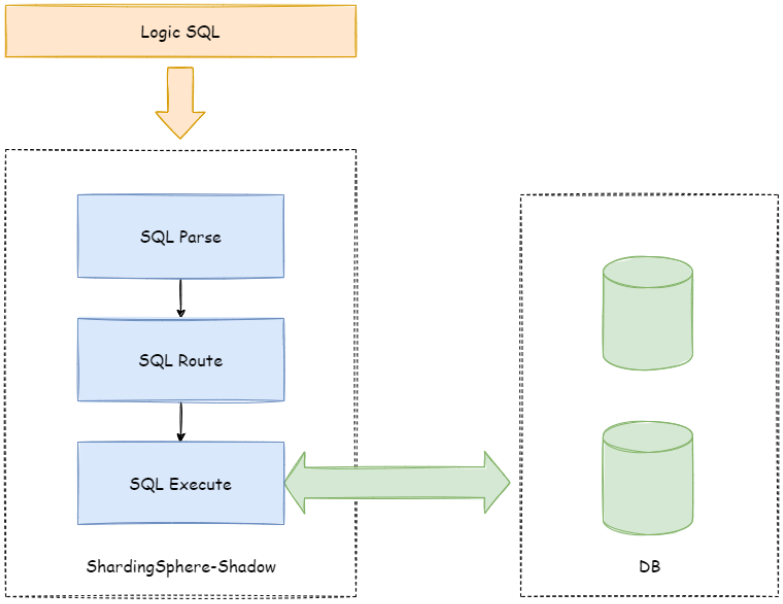
如下图所示：



12.9 影子库

12.9.1 原理介绍

Apache ShardingSphere 通过解析 SQL，对传入的 SQL 进行影子判定，根据配置文件中用户设置的影子规则，路由到生产库或者影子库。



以 INSERT 语句为例，在写入数据时，Apache ShardingSphere 会对 SQL 进行解析，再根据配置文件中的规则，构造一条路由链。在当前版本的功能中，影子功能处于路由链中的最后一个执行单元，即，如果有其他需要路由的规则存在，如分片，Apache ShardingSphere 会首先根据分片规则，路由到某一个数据库，再执行影子路由判定流程，判定执行 SQL 满足影子规则的配置，数据路由到与之对应的影子库，生产数据则维持不变。

DML 语句

支持两种算法。影子判定会首先判断执行 SQL 相关表与配置的影子表是否有交集。如果有交集，依次判定交集部分影子表关联的影子算法，有任何一个判定成功。SQL 语句路由到影子库。影子表没有交集或者影子算法判定不成功，SQL 语句路由到生产库。

DDL 语句

仅支持注解影子算法。在压测场景下，DDL 语句一般不需要测试。主要在初始化或者修改影子库中影子表时使用。影子判定会首先判断执行 SQL 是否包含注解。如果包含注解，影子规则中配置的 HINT 影子算法依次判定。有任何一个判定成功。SQL 语句路由到影子库。执行 SQL 不包含注解或者 HINT 影子算法判定不成功，SQL 语句路由到生产库。

12.9.2 相关参考

JAVA API：影子库配置

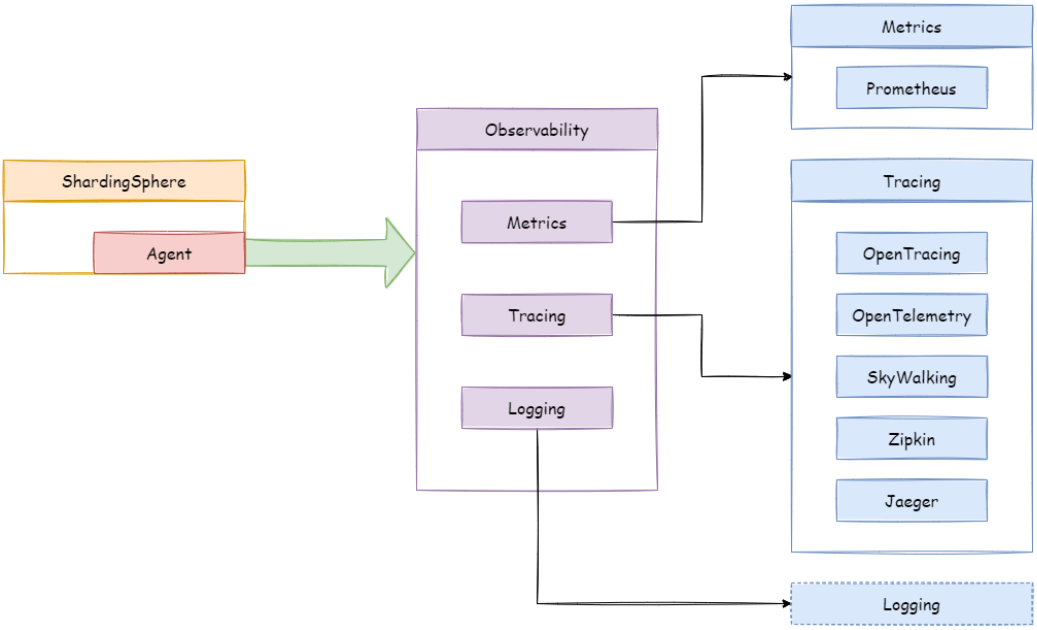
YAML 配置：影子库配置

12.10 可观察性

12.10.1 原理说明

ShardingSphere-Agent 模块为 ShardingSphere 提供了可观察性的框架，它是基于 Java Agent 技术实现的。

Metrics、Tracing 和 Logging 等功能均通过插件的方式集成在 Agent 中，如图：



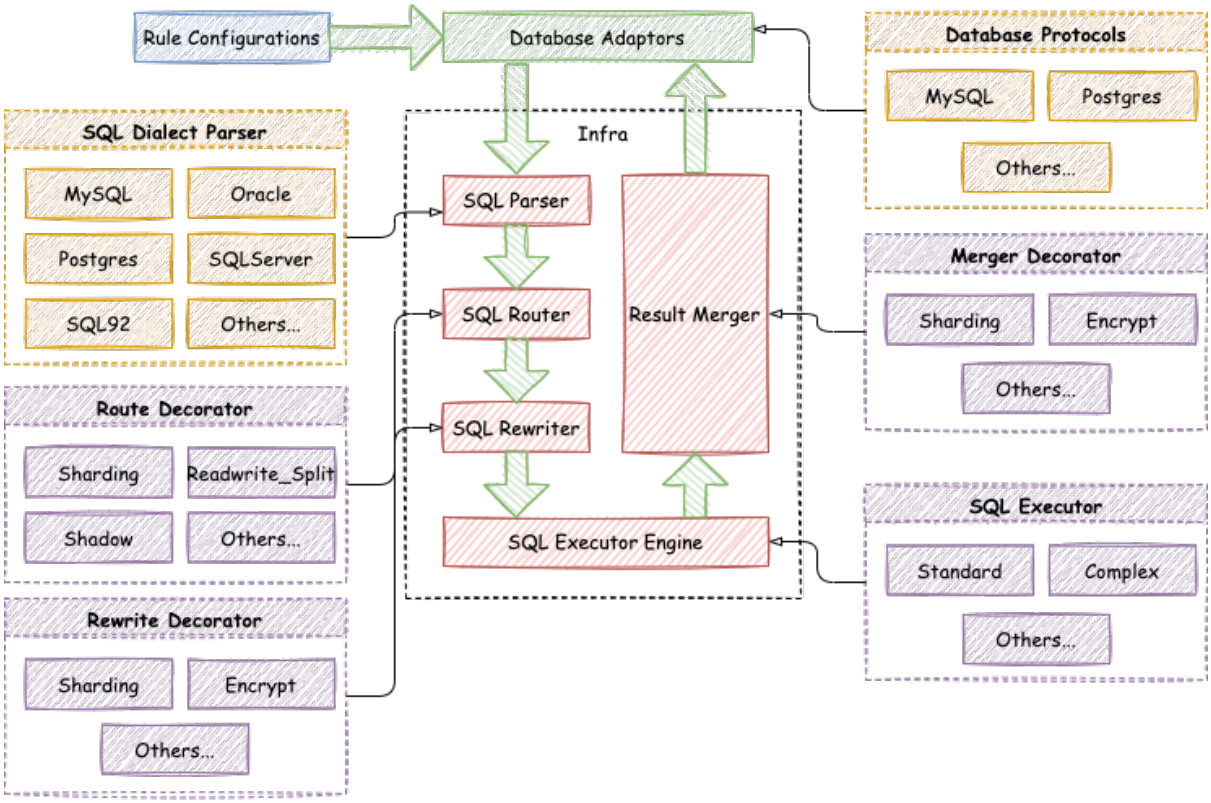
- Metrics 插件用于收集和展示整个集群的统计指标。Apache ShardingSphere 默认提供了对 Prometheus 的支持。
- Tracing 插件用于获取 SQL 解析与 SQL 执行的链路跟踪信息。Apache ShardingSphere 默认提供了导出 tracing 数据到 Jaeger、和 Zipkin 的支持，也支持用户通过插件化的方式开发自定义的 Tracing 组件。
- 默认的 Logging 插件展示了如何在 ShardingSphere 中记录额外的日志，实际应用中需要用户根据自己的需求进行探索。

12.11 基础架构

让开发者能够像使用积木一样定制属于自己的独特系统，是 Apache ShardingSphere 可插拔架构的设计目标。

可插拔架构对程序架构设计的要求非常高，需要将各个模块相互独立，互不感知，并且通过一个可插拔内核，以叠加的方式将各种功能组合使用。设计一套将功能开发完全隔离的架构体系，既可以最大限度的将开源社区的活力激发出来，也能够保障项目的质量。

Apache ShardingSphere 5.x 版本开始致力于可插拔架构，项目的功能组件能够灵活的以可插拔的方式进行扩展。目前，数据分片、读写分离、数据库高可用、数据加密、影子库压测等功能，以及对 MySQL、PostgreSQL、SQLServer、Oracle 等 SQL 与协议的支持，均通过插件的方式织入项目。Apache ShardingSphere 目前已提供数十个 SPI（Service Provider Interface） 作为系统的扩展点，而且仍在不断增加中。



13 FAQ

13.1 JDBC

13.1.1 引入 shardingsphere-transaction-xa-core 后，如何避免 spring-boot 自动加载默认的 JtaTransactionManager?

回答:

1. 需要在 spring-boot 的引导类中添加 `@SpringBootApplication(exclude = JtaAutoConfiguration.class)`。

13.1.2 Oracle 表名、字段名配置大小写在加载 metadata 元数据时结果不正确?

回答：需要注意，Oracle 表名和字段名，默认元数据都是大写，除非建表语句中带双引号，如 `CREATE TABLE "TableName"("Id" number)` 元数据为双引号中内容，可参考以下 SQL 查看元数据的具体情况:

```
SELECT OWNER, TABLE_NAME, COLUMN_NAME, DATA_TYPE FROM ALL_TAB_COLUMNS WHERE TABLE_
NAME IN ('TableName')
```

ShardingSphere 使用 `OracleTableMetaDataLoader` 对 Oracle 元数据进行加载，配置时需确保表名、字段名的大小写配置与数据库中的一致。ShardingSphere 查询元数据关键 SQL:

```
private String getTableMetaDataSQL(final Collection<String> tables, final
DatabaseMetaData metaData) throws SQLException {
    StringBuilder stringBuilder = new StringBuilder(28);
    if (versionContainsIdentityColumn(metaData)) {
        stringBuilder.append(", IDENTITY_COLUMN");
    }
    if (versionContainsCollation(metaData)) {
        stringBuilder.append(", COLLATION");
    }
    String collation = stringBuilder.toString();
```

```
    return tables.isEmpty() ? String.format(TABLE_META_DATA_SQL, collation)
            : String.format(TABLE_META_DATA_SQL_IN_TABLES, collation, tables.
stream().map(each -> String.format("'%s'", each)).collect(Collectors.joining(",
")));
}
```

13.1.3 使用 MySQL XA 事务时报 SQLException: Unable to unwrap to interface com.mysql.jdbc.Connection 异常

回答:

多个 MySQL 驱动之间不兼容。由于优先加载了类路径下 MySQL5 版本的驱动类，当试图调用 MySQL8 驱动里的 unwrap 方法时，类型转换异常。

解决方案：检查类路径下是否同时存在 MySQL5 和 MySQL8 的驱动，只保留对应版本的一个驱动包即可。

异常堆栈如下:

```
Caused by: java.sql.SQLException: Unable to unwrap to interface com.mysql.jdbc.
Connection
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:129)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:97)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:89)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:63)
    at com.mysql.cj.jdbc.ConnectionImpl.unwrap(ConnectionImpl.java:2650)
    at com.zaxxer.hikari.pool.ProxyConnection.unwrap(ProxyConnection.java:481)
    at org.apache.shardingsphere.transaction.xa.jta.connection.dialect.
MySQLXAConnectionWrapper.wrap(MySQLXAConnectionWrapper.java:46)
    at org.apache.shardingsphere.transaction.xa.jta.datasource.
XATransactionDataSource.getConnection(XATransactionDataSource.java:89)
    at org.apache.shardingsphere.transaction.xa.XAShardingSphereTransactionManager.
getConnection(XAShardingSphereTransactionManager.java:96
```

13.2 Proxy

13.2.1 Windows 环境下，运行 ShardingSphere-Proxy，找不到或无法加载主类 org.apache.shardingsphere.proxy.Bootstrap，如何解决?

回答:

某些解压缩工具在解压 ShardingSphere-Proxy 二进制包时可能将文件名截断，导致找不到某些类。解决方案：打开 cmd.exe 并执行下面的命令:

```
tar zxvf apache-shardingsphere-${RELEASE.VERSION}-shardingsphere-proxy-bin.tar.gz
```

13.2.2 在使用 ShardingSphere-Proxy 的时候，如何动态在添加新的逻辑库？

回答：

使用 ShardingSphere-Proxy 时，可以通过 `DistSQL` 动态的创建或移除逻辑库，语法如下：

```
CREATE DATABASE  [IF NOT EXISTS] databaseName;
DROP DATABASE  [IF EXISTS] databaseName;
```

例：

```
CREATE DATABASE sharding_db;
DROP DATABASE sharding_db;
```

13.2.3 在使用 ShardingSphere-Proxy 时，怎么使用合适的工具连接到 ShardingSphere-Proxy？

回答：

1. ShardingSphere-Proxy 可以看做是一个 database server，所以首选支持 SQL 命令连接和操作。
2. 如果使用其他第三方数据库工具，可能由于不同工具的特定实现导致出现异常。
3. 目前已测试的第三方数据库工具如下：
   - DataGrip：2020.1、2021.1（使用 IDEA/DataGrip 时打开 `introspect using JDBC metadata` 选项）。
   - MySQLWorkBench：8.0.25。

13.2.4 使用第三方数据库工具连接 ShardingSphere-Proxy 时，如果 ShardingSphere-Proxy 没有创建 Database 或者没有注册 Storage Unit，连接失败？

回答：

1. 第三方数据库工具在连接 ShardingSphere-Proxy 时会发送一些 SQL 查询元数据，当 ShardingSphere-Proxy 没有创建 `database` 或者没有注册 `storage unit` 时，ShardingSphere-Proxy 无法执行 SQL。
2. 推荐先创建 `database` 并注册 `storage unit` 之后再使用第三方数据库工具连接。
3. 有关 `storage unit` 的详情请参考。相关介绍

13.3 分片

13.3.1 Cloud not resolve placeholder …in string value …异常的解决方法?

回答:

使用 `InlineExpressionParser` SPI 的默认实现的行表达式标识符可以使用 `${...}` 或 `$->{...}`,但前者与 Spring 本身的属性文件占位符冲突，因此在 Spring 环境中使用行表达式标识符建议使用 `$->{...}`。

13.3.2 inline 表达式返回结果为何出现浮点数?

回答:

Java 的整数相除结果是整数，但是对于 inline 表达式中的 Groovy 语法则不同，整数相除结果是浮点数。想获得除法整数结果需要将 A/B 改为 A.intdiv(B)。

13.3.3 如果只有部分数据库分库分表，是否需要将不分库分表的表也配置在分片规则中?

回答:

不分库分表的表在 ShardingSphere 中叫做单表，可以使用 LOAD 语句或者 SINGLE 规则配置需要加载的单表。

13.3.4 指定了泛型为 Long 的 SingleKeyTableShardingAlgorithm，遇到 ClassCastException: Integer can not cast to Long?

回答:

必须确保数据库表中该字段和分片算法该字段类型一致，如：数据库中该字段类型为 int(11)，泛型所对应的分片类型应为 Integer，如果需要配置为 Long 类型，请确保数据库中该字段类型为 bigint。

13.3.5 [分片、PROXY] 实现 StandardShardingAlgorithm 自定义算法时，指定了 Comparable 的具体类型为 Long, 且数据库表中字段类型为 bigint，出现 ClassCastException: Integer can not cast to Long 异常。

回答:

实现 `doSharding` 方法时，不建议指定方法声明中 `Comparable` 具体的类型，而是在 `doSharding` 方法实现中对类型进行转换，可以参考 `ModShardingAlgorithm#doSharding` 方法

13.3.6 ShardingSphere 提供的默认分布式自增主键策略为什么是不连续的，且尾数大多为偶数?

回答:

ShardingSphere 采用 snowflake 算法作为默认的分布式自增主键策略，用于保证分布式的情况下可以无中心化的生成不重复的自增序列。因此自增主键可以保证递增，但无法保证连续。而 snowflake 算法的最后 4 位是在同一毫秒内的访问递增值。因此，如果毫秒内并发度不高，最后 4 位为零的几率则很大。因此并发度不高的应用生成偶数主键的几率会更高。在 3.1.0 版本中，尾数大多为偶数的问题已彻底解决，参见：https://github.com/apache/shardingsphere/issues/1617

13.3.7 如何在 inline 分表策略时，允许执行范围查询操作（BETWEEN AND、>、<、>=、<=）?

回答:

1. 需要使用 4.1.0 或更高版本。
2. 调整以下配置项（需要注意的是，此时所有的范围查询将会使用广播的方式查询每一个分表）:

- 4.x 版本：`allow.range.query.with.inline.sharding` 设置为 true 即可（默认为 false）。
- 5.x 版本：在 InlineShardingStrategy 中将 `allow-range-query-with-inline-sharding` 设置为 true 即可（默认为 false）。

13.3.8 为什么我实现了 `KeyGenerateAlgorithm` 接口，也配置了 `Type`，但是自定义的分布式主键依然不生效?

回答:

Service Provider Interface (SPI) 是一种为了被第三方实现或扩展的 API，除了实现接口外，还需要在 META-INF/services 中创建对应文件来指定 SPI 的实现类，JVM 才会加载这些服务。具体的 SPI 使用方式，请大家自行搜索。与分布式主键 `KeyGenerateAlgorithm` 接口相同，其他 ShardingSphere 的扩展功能也需要用相同的方式注入才能生效。

13.3.9 ShardingSphere 除了支持自带的分布式自增主键之外，还能否支持原生的自增主键?

回答:

是的，可以支持。但原生自增主键有使用限制，即不能将原生自增主键同时作为分片键使用。由于 ShardingSphere 并不知晓数据库的表结构，而原生自增主键是不包含在原始 SQL 中内的，因此 ShardingSphere 无法将该字段解析为分片字段。如自增主键非分片键，则无需关注，可正常返回；若自增主键同时作为分片键使用，ShardingSphere 无法解析其分片值，导致 SQL 路由至多张表，从而影响应用的正确性。而原生自增主键返回的前提条件是 INSERT SQL 必须最终路由至一张表，因此，面对返回多表的 INSERT SQL，自增主键则会返回零。

13.4 单表

13.4.1 Table or view %s does not exist. 异常如何解决?

回答:

在 ShardingSphere 5.4.0 之前的版本，单表采用了自动加载的方式，这种方式在实际使用中存在诸多问题:

1. 逻辑库中注册大量数据源后，自动加载的单表数量过多会导致 ShardingSphere-Proxy/JDBC 启动变慢;
2. 用户通过 DistSQL 方式使用时，通常会按照：**注册存储单元 -> 创建分片、加密、读写分离等规则 -> 创建表**的顺序进行操作。由于单表自动加载机制的存在，会导致操作过程中多次访问数据库进行加载，并且在多个规则混合使用时会导致单表元数据的错乱;
3. 自动加载全部数据源中的单表，用户无法排除不想被 ShardingSphere 管理的单表或废弃表。

为了解决以上问题，从 ShardingSphere 5.4.0 版本开始，调整了单表的加载方式，用户需要通过 YAML 配置或者 DistSQL 的方式手动加载数据库中的单表。需要注意的是，使用 DistSQL LOAD 语句加载单表时，需要保证所有数据源完成注册，所以规则创建完成后，再基于逻辑数据源（不存在逻辑数据源则使用物理数据源）进行单表 LOAD 操作。

- YAML 加载单表示例:

```
rules:
  - !SINGLE
    tables:
      - "*.*"
  - !READWRITE_SPLITTING
     dataSourceGroups:
      readwrite_ds:
        writeDataSourceName: write_ds
        readDataSourceNames:
          - read_ds_0
          - read_ds_1
        loadBalancerName: random
    loadBalancers:
      random:
        type: RANDOM
```

更多加载单表 YAML 配置请参考单表。

- DistSQL 加载单表示例:

```
LOAD SINGLE TABLE *.*;
```

更多 LOAD 单表 DistSQL 请参考单表加载。

13.5 DistSQL

13.5.1 使用 DistSQL 添加数据源时，如何设置自定义的 JDBC 连接参数或连接池属性？

回答：

1. 如需自定义 JDBC 参数，请使用 `urlSource` 的方式定义 `dataSource`。
2. ShardingSphere 预置了必要的连接池参数，如 `maxPoolSize`、`idleTimeout` 等。如需增加或覆盖参数配置，请在 `dataSource` 中通过 `PROPERTIES` 指定。
3. 以上规则请参考 相关介绍。

13.5.2 使用 DistSQL 删除 storage unit 时，出现 Storage unit [xxx] is still used by [SingleRule]。

回答：

1. 被规则引用的 `storage unit` 将无法被删除。
2. 若 `storage unit` 只被 `single rule` 引用，且用户确认可以忽略该限制，则可以添加可选参数 ignore single tables 进行强制删除。

13.5.3 使用 DistSQL 添加数据源时，出现 Failed to get driver instance for jdbcURL=xxx。

回答：

ShardingSphere-Proxy 在部署过程中没有添加 jdbc 驱动，需要将 jdbc 驱动放入 ShardingSphere-Proxy 解压后的 ext-lib 目录，例如：`mysql-connector`。

13.6 其他

13.6.1 如果 SQL 在 ShardingSphere 中执行不正确，该如何调试？

回答：

在 ShardingSphere-Proxy 以及 ShardingSphere-JDBC 1.5.0 版本之后提供了 `sql.show` 的配置，可以将解析上下文和改写后的 SQL 以及最终路由至的数据源的细节信息全部打印至 info 日志。`sql.show` 配置默认关闭，如果需要请通过配置开启。> 注意：5.x 版本以后，`sql.show` 参数调整为 `sql-show`。

13.6.2 阅读源码时为什么会出现编译错误? IDEA 不索引生成的代码?

回答:

ShardingSphere 使用 lombok 实现极简代码。关于更多使用和安装细节，请参考 lombok 官网。`org.apache.shardingsphere.sql.parser.autogen` 包下的代码由 ANTLR 生成，可以执行以下命令快速生成:

```
./mvnw -DskipITs -DskipTests install -T1C
```

生成的代码例如 `org.apache.shardingsphere.sql.parser.autogen.PostgreSQLStatementParser` 等 Java 文件由于较大，默认配置的 IDEA 可能不会索引该文件。可以调整 IDEA 的属性: `idea.max.intellisense.filesize=10000`。

13.6.3 使用 SQLSever 和 PostgreSQL 时，聚合列不加别名会抛异常?

回答:

SQLServer 和 PostgreSQL 获取不加别名的聚合列会改名。例如，如下 SQL:

```
SELECT SUM(num), SUM(num2) FROM tablexxx;
```

SQLServer 获取到的列为空字符串和 (2)，PostgreSQL 获取到的列为空 sum 和 sum(2)。这将导致 ShardingSphere 在结果归并时无法找到相应的列而出错。正确的 SQL 写法应为:

```
SELECT SUM(num) AS sum_num, SUM(num2) AS sum_num2 FROM tablexxx;
```

13.6.4 Oracle 数据库使用 Timestamp 类型的 Order By 语句抛出异常提示“Order by value must implements Comparable”?

回答:

针对上面问题解决方式有两种: 1. 配置启动 JVM 参数“-oracle.jdbc.J2EE13Compliant=true” 2. 通过代码在项目初始化时设置 System.getProperties().setProperty(“oracle.jdbc.J2EE13Compliant”, “true”); 原因如下: `org.apache.shardingsphere.sharding.merge.dql.orderby.OrderByValue#getOrderValues()` 方法如下:

```
private List<Comparable<?>> getOrderValues() throws SQLException {
    List<Comparable<?>> result = new ArrayList<>(orderByItems.size());
    for (OrderItem each : orderByItems) {
        Object value = resultSet.getObject(each.getIndex());
        Preconditions.checkState(null == value || value instanceof Comparable,
"Order by value must implements Comparable");
        result.add((Comparable<?>) value);
    }
    return result;
}
```

使用了 resultSet.getObject(int index) 方法，针对 TimeStamp oracle 会根据 oracle.jdbc.J2EE13Compliant 属性判断返回 java.sql.TimeStamp 还是自定义 oracle.sql.TIMESTAMP 详见 ojdbc 源码 oracle.jdbc.driver.TimestampAccessor#getObject(int var1) 方法:

```
Object getObject(int var1) throws SQLException {
    Object var2 = null;
    if(this.rowSpaceIndicator == null) {
        DatabaseError.throwSqlException(21);
    }
    if(this.rowSpaceIndicator[this.indicatorIndex + var1] != -1) {
        if(this.externalType != 0) {
            switch(this.externalType) {
            case 93:
                return this.getTimestamp(var1);
            default:
                DatabaseError.throwSqlException(4);
                return null;
            }
        }
        if(this.statement.connection.j2ee13Compliant) {
            var2 = this.getTimestamp(var1);
        } else {
            var2 = this.getTIMESTAMP(var1);
        }
    }
    return var2;
}
```

13.6.5 Windows 环境下，通过 Git 克隆 ShardingSphere 源码时为什么提示文件名过长，如何解决?

回答:

为保证源码的可读性，ShardingSphere 编码规范要求类、方法和变量的命名要做到顾名思义，避免使用缩写，因此可能导致部分源码文件命名较长。由于 Windows 版本的 Git 是使用 msys 编译的，它使用了旧版本的 Windows Api，限制文件名不能超过 260 个字符。解决方案如下：打开 cmd.exe（你需要将 git 添加到环境变量中）并执行下面的命令，可以让 git 支持长文件名:

```
git config --global core.longpaths true
```

如果是 Windows 10，还需要通过注册表或组策略，解除操作系统的文件名长度限制（需要重启） ：> 在注册表编辑器中创建 `HKLM\SYSTEM\CurrentControlSet\Control\FileSystem LongPathsEnabled`，类型为 `REG_DWORD`，并设置为 1。> 或者从系统菜单点击设置图标，输入“编辑组策略”，然后在打开的窗口依次进入“计算机管理”>“管理模板”>“系统”>“文件系统”，在右侧双击“启用 win32 长路径”。参考资料：https://docs.microsoft.com/zh-cn/windows/desktop/FileIO/naming-a-file https://ourcodeworld.com/articles/read/109/how-to-solve-filename-too-long-error-in-git-powershell-and-github-application-for-windows

13.6.6 Type is required 异常的解决方法?

回答:

ShardingSphere 中很多功能实现类的加载方式是通过 SPI 注入的方式完成的，如分布式主键，注册中心等；这些功能通过配置中 type 类型来寻找对应的 SPI 实现，因此必须在配置文件中指定类型。

13.6.7 服务启动时如何加快 metadata 加载速度?

回答:

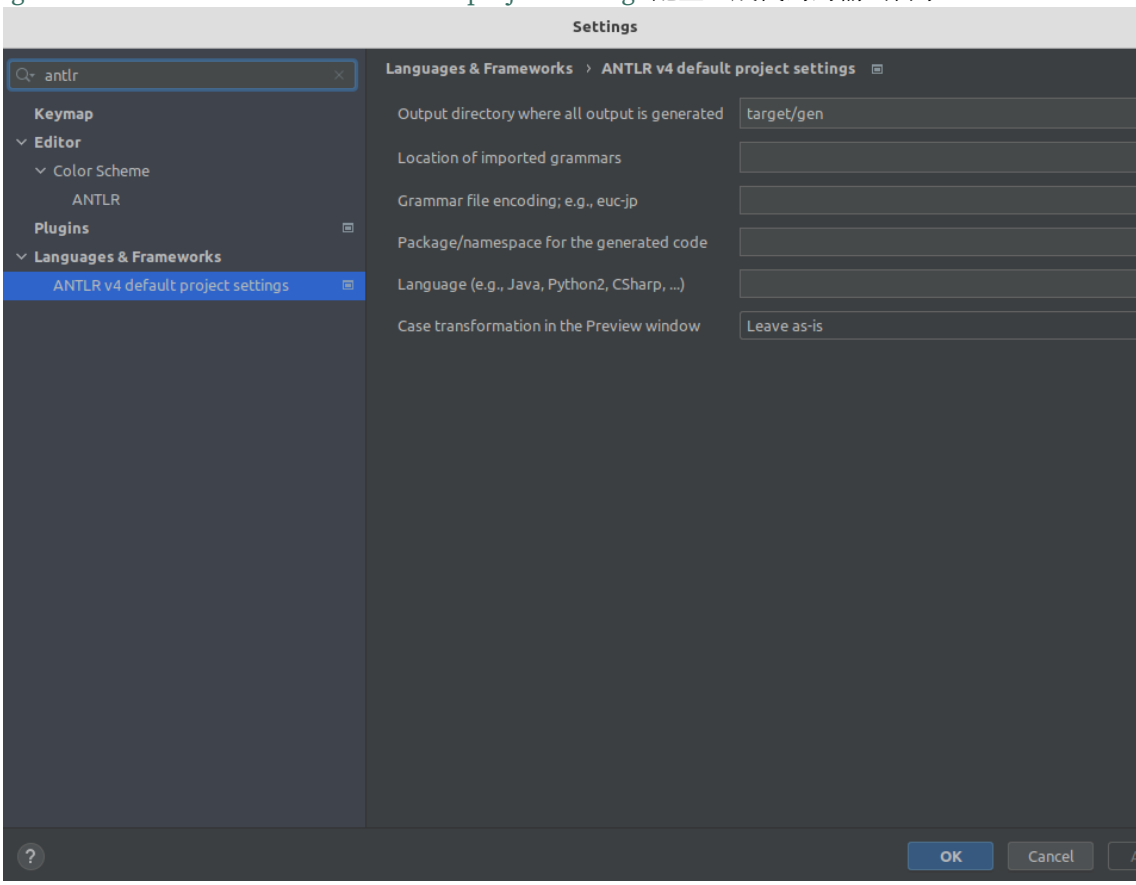
1. 升级到 `4.0.1` 以上的版本，以提高 metadata 的加载速度。
2. 参照你采用的连接池，将:

- 配置项 `max.connections.size.per.query`(默认值为 1)调高(版本 >= 3.0.0.M3 且低于 5.0.0)。
- 配置项 `max-connections-size-per-query`（默认值为 1）调高（版本 >= 5.0.0)。

13.6.8 ANTLR 插件在 src 同级目录下生成代码，容易误提交，如何避免?

回答:

进入 Settings -> Languages & Frameworks -> ANTLR v4 default project settings 配置生成代码的输出目录为 `target/gen`,如图:



13.6.9 使用 Proxool 时分库结果不正确?

回答:

使用 Proxool 配置多个数据源时，应该为每个数据源设置 alias，因为 Proxool 在获取连接时会判断连接池中是否包含已存在的 alias，不配置 alias 会造成每次都只从一个数据源中获取连接。以下是 Proxool 源码中 ProxoolDataSource 类 getConnection 方法的关键代码:

```
if(!ConnectionPoolManager.getInstance().isPoolExists(this.alias)) {
    this.registerPool();
}
```

更多关于 alias 使用方法请参考 Proxool 官网。PS：sourceforge 网站需要翻墙访问。

14 下载

14.1 最新版本

Apache ShardingSphere 的发布版包括源码包及其对应的二进制包。由于下载内容分布在镜像服务器上，所以下载后应该进行 GPG 或 SHA-512 校验，以此来保证内容没有被篡改。

14.1.1 Apache ShardingSphere - 版本: 5.5.2 (发布日期: January 16th, 2025)

- 源码: SRC (ASC, SHA512)
- ShardingSphere-JDBC 二进制包: TAR (ASC, SHA512)
- ShardingSphere-Proxy 二进制包: TAR (ASC, SHA512)
- ShardingSphere-Agent 二进制包: TAR (ASC, SHA512)

14.2 全部版本

全部版本请到 Archive repository 查看。全部孵化器版本请到 Archive incubator repository 查看。

14.3 校验版本

PGP 签名文件

使用 PGP 或 SHA 签名验证下载文件的完整性至关重要。可以使用 GPG 或 PGP 验证 PGP 签名。请下载 KEYS 以及发布的 asc 签名文件。建议从主发布目录而不是镜像中获取这些文件。

```
gpg -i KEYS
```

或者

```
pgpk -a KEYS
```

或者

```
pgp -ka KEYS
```

要验证二进制文件或源代码，您可以从主发布目录下载相关的 asc 文件，并按照以下指南进行操作。

```
gpg --verify apache-shardingsphere-********.asc apache-shardingsphere-*********
```

或者

```
pgpv apache-shardingsphere-********.asc
```

或者

```
pgp apache-shardingsphere-********.asc
```