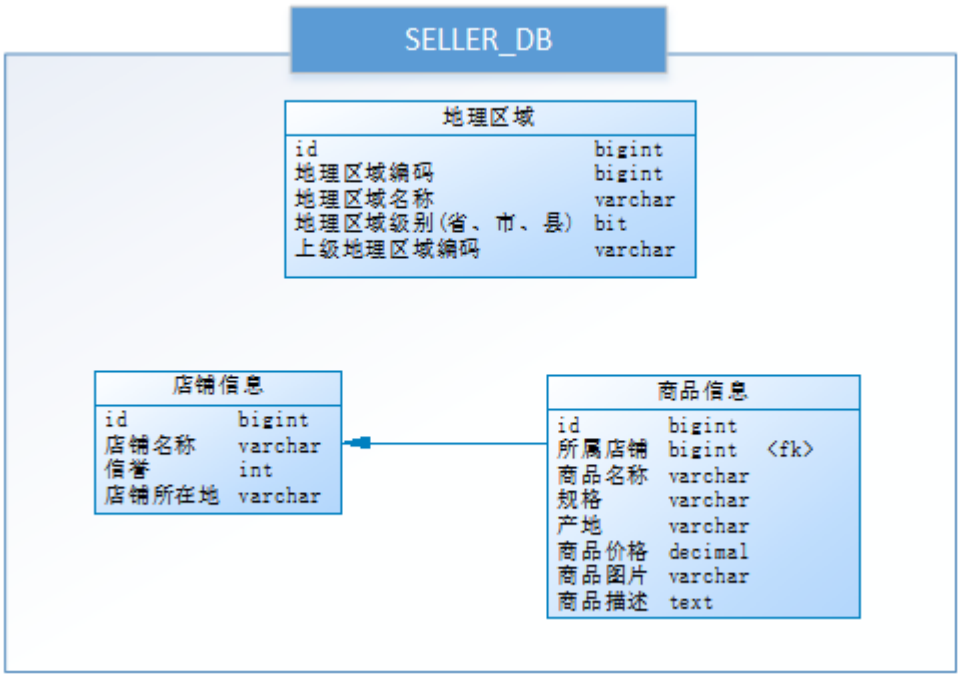
Sharding-JDBC分库分表专题

1.概述

1.1.分库分表是什么

小明是一家初创电商平台的开发人员，他负责卖家模块的功能开发，其中涉及了店铺、商品的相关业务，设计如下数据库：



通过以下SQL能够获取到商品相关的店铺信息、地理区域信息：

```
SELECT p.*,r.[地理区域名称],s.[店铺名称],s.[信誉]
FROM [商品信息] p
LEFT JOIN [地理区域] r ON p.[产地] = r.[地理区域编码]
LEFT JOIN [店铺信息] s ON p.id = s.[所属店铺]
WHERE p.id = ?
```

形成类似以下列表展示：

Vita Coco唯他可可椰子水饮料进口nfc青椰果汁500ml*24瓶原味正品

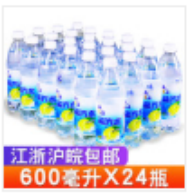
￥298.00 包邮

vitacoco唯他可可旗舰店 北京

绿力冬瓜茶 绿力冬瓜汁饮料310ml *24罐 整箱装冬瓜茶饮料

￥88.80
运费: 18.00

绿力食品旗舰店 上海



上海风味盐汽水柠檬味汽水夏季防暑降温碳酸饮料600ml*24瓶整箱

￥26.80
运费: 8.00

萌大叔食品 上海

沙棘汁野山坡吕梁山西特产饮料整箱生榨果汁纯沙棘汁原浆16瓶装

￥47.88 包邮

tianjiaoshiye 山西 吕梁

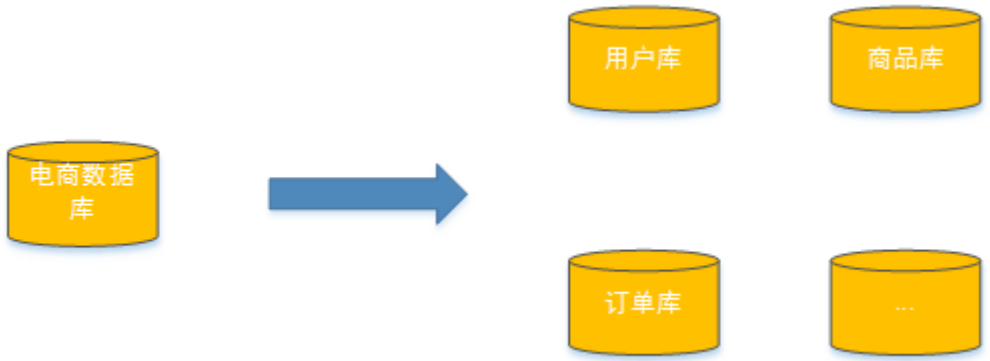
随着公司业务快速发展，数据库中的数据量猛增，访问性能也变慢了，优化迫在眉睫。分析一下问题出现在哪儿呢？ 关系型数据库本身比较容易成为系统瓶颈，单机存储容量、连接数、处理能力都有限。当单表的数据量达到1000W或100G以后，由于查询维度较多，即使添加从库、优化索引，做很多操作时性能仍下降严重。

方案1：

通过提升服务器硬件能力来提高数据处理能力，比如增加存储容量 、CPU等，这种方案成本很高，并且如果瓶颈在MySQL本身那么提高硬件也是有很的。

方案2：

把数据分散在不同的数据库中，使得单一数据库的数据量变小来缓解单一数据库的性能问题，从而达到提升数据库性能的目的，如下图：将电商数据库拆分为若干独立的数据库，并且对于大表也拆分为若干小表，通过这种数据库拆分的方法来解决数据库的性能问题。



分库分表就是为了解决由于数据量过大而导致数据库性能降低的问题，将原来独立的数据库拆分成若干数据库组成，将数据大表拆分成若干数据表组成，使得单一数据库、单一数据表的数据量变小，从而达到提升数据库性能的目的。

1.2.分库分表的方式

分库分表包括分库和分表两个部分，在生产中通常包括：垂直分库、水平分库、垂直分表、水平分表四种方式。

1.2.1.垂直分表

下边通过一个商品查询的案例讲解垂直分表：

通常在商品列表中是不显示商品详情信息的，如下图：

Vita Coco唯他可可椰子水饮料进口nfc青椰果汁500ml*24瓶原味正品

￥298.00 包邮

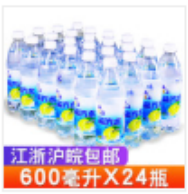
vitacoco唯他可可旗舰店 北京

绿力冬瓜茶 绿力冬瓜汁饮料310ml *24罐 整箱装冬瓜茶饮料

￥88.80

运费: 18.00

绿力食品旗舰店 上海



上海风味盐汽水柠檬味汽水夏季防暑降温碳酸饮料600ml*24瓶整箱

￥26.80

运费: 8.00

萌大叔食品 上海

沙棘汁野山坡吕梁山西特产饮料整箱生榨果汁纯沙棘汁原浆16瓶装

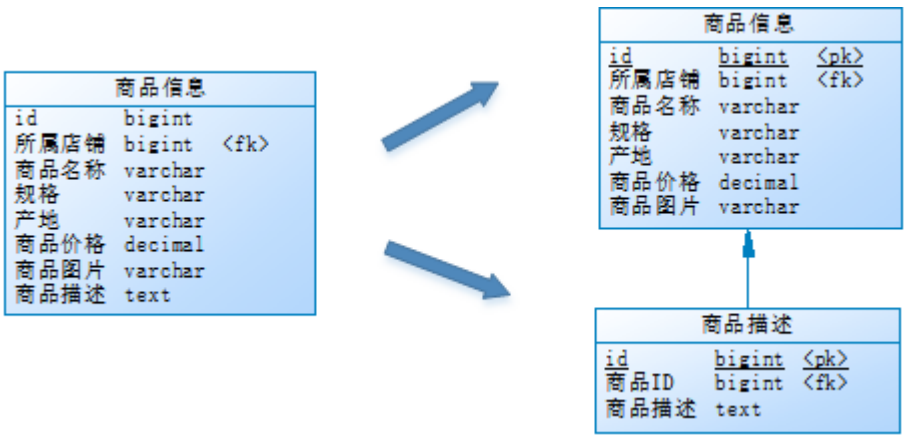
￥47.88 包邮

tianjiaoshiye 山西 吕梁

用户在浏览商品列表时，只有对某商品感兴趣时才会查看该商品的详细描述。因此，商品信息中**商品描述**字段访问频次较低，且该字段存储占用空间较大，访问单个数据IO时间较长；商品信息中商品名称、商品图片、商品价格等其他字段数据访问频次较高。

由于这两种数据的特性不一样，因此他考虑将商品信息表拆分如下：

将访问频次低的商品描述信息单独存放在一张表中，访问频次较高的商品基本信息单独放在一张表中。



商品列表可采用以下sql：

```
SELECT p.*,r.[地理区域名称],s.[店铺名称],s.[信誉]
FROM [商品信息] p
LEFT JOIN [地理区域] r ON p.[产地] = r.[地理区域编码]
LEFT JOIN [店铺信息] s ON p.id = s.[所属店铺]
WHERE...ORDER BY...LIMIT...
```

需要获取商品描述时，再通过以下sql获取：

```
SELECT *
FROM [商品描述]
WHERE [商品ID] = ?
```

小明进行的这一步优化，就叫**垂直分表。**

垂直分表定义：将一个表按照字段分成多表，每个表存储其中一部分字段。

它带来的提升是：

1.为了避免IO争抢并减少锁表的几率，查看详情的用户与商品信息浏览互不影响

2.充分发挥热门数据的操作效率，商品信息的操作的高效率不会被商品描述的低效率所拖累。

一般来说，某业务实体中的各个数据项的访问频次是不一样的，部分数据项可能是占用存储空间比较大的BLOB或是TEXT。例如上例中的**商品描述。**所以，当表数据量很大时，可以**将表按字段切开，将热门字段、冷门字段分开放置在不同库中**，这些库可以放在不同的存储设备上，避免IO争抢。垂直切分带来的性能提升主要集中在热门数据的操作效率上，而且磁盘争用情况减少。

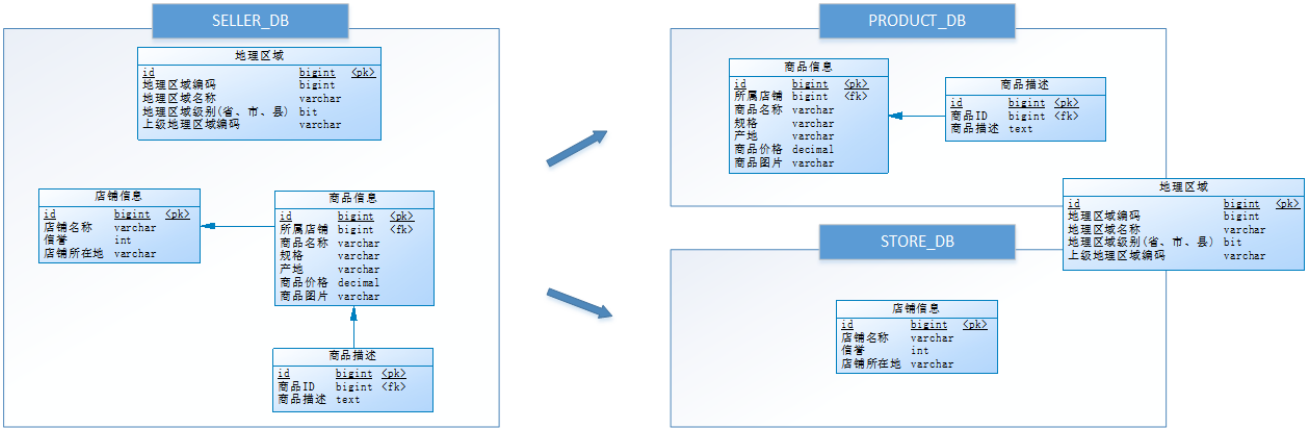
通常我们按以下原则进行垂直拆分:

1. 把不常用的字段单独放在一张表;
2. 把text，blob等大字段拆分出来放在附表中;
3. 经常组合查询的列放在一张表中;

1.2.2.垂直分库

通过垂直分表性能得到了一定程度的提升，但是还没有达到要求，并且磁盘空间也快不够了，因为数据还是始终限制在一台服务器，库内垂直分表只解决了单一表数据量过大的问题，但没有将表分布到不同的服务器上，因此每个表还是竞争同一个物理机的CPU、内存、网络IO、磁盘。

经过思考，他把原有的SELLER_DB(卖家库)，分为了PRODUCT_DB(商品库)和STORE_DB(店铺库)，并把这两个库分散到不同服务器，如下图：



由于**商品信息**与**商品描述**业务耦合度较高，因此一起被存放在PRODUCT_DB(商品库)；而**店铺信息**相对独立，因此单独被存放在STORE_DB(店铺库)。

小明进行的这一步优化，就叫**垂直分库**。

垂直分库是指按照业务将表进行分类，分布到不同的数据库上面，每个库可以放在不同的服务器上，它的核心理念是专库专用。

它带来的提升是：

- 解决业务层面的耦合，业务清晰
- 能对不同业务的数据进行分级管理、维护、监控、扩展等
- 高并发场景下，垂直分库一定程度的提升IO、数据库连接数、降低单机硬件资源的瓶颈

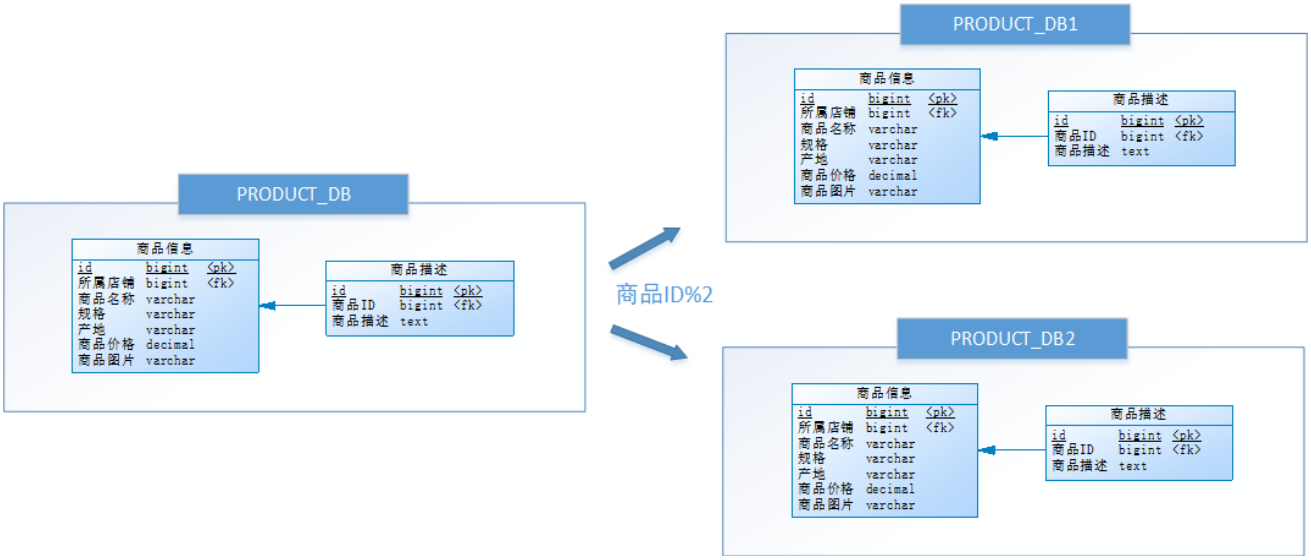
垂直分库通过将表按业务分类，然后分布在不同数据库，并且可以将这些数据库部署在不同服务器上，从而达到多个服务器共同分摊压力的效果，但是依然没有解决单表数据量过大的问题。

1.2.3.水平分库

经过**垂直分库**后，数据库性能问题得到一定程度的解决，但是随着业务量的增长，PRODUCT_DB(商品库)单库存储数据已经超出预估。粗略估计，目前有8w店铺，每个店铺平均150个不同规格的商品，再算上增长，那商品数量得往1500w+上预估，并且PRODUCT_DB(商品库)属于访问非常频繁的资源，单台服务器已经无法支撑。此时该如何优化？

再次分库？但是从业务角度分析，目前情况已经无法再次垂直分库。

尝试水平分库，将店铺ID为单数的和店铺ID为双数的商品信息分别放在两个库中。



也就是说，要操作某条数据，先分析这条数据所属的店铺ID。如果店铺ID为双数，将此操作映射至RRODUCT_DB1(商品库1)；如果店铺ID为单数，将操作映射至RRODUCT_DB2(商品库2)。此操作要访问数据库名称的表达式为**RRODUCT_DB[店铺ID%2 + 1]** 。

小明进行的这一步优化，就叫**水平分库**。

水平分库是把同一个表的数据按一定规则拆到不同的数据库中，每个库可以放在不同的服务器上。

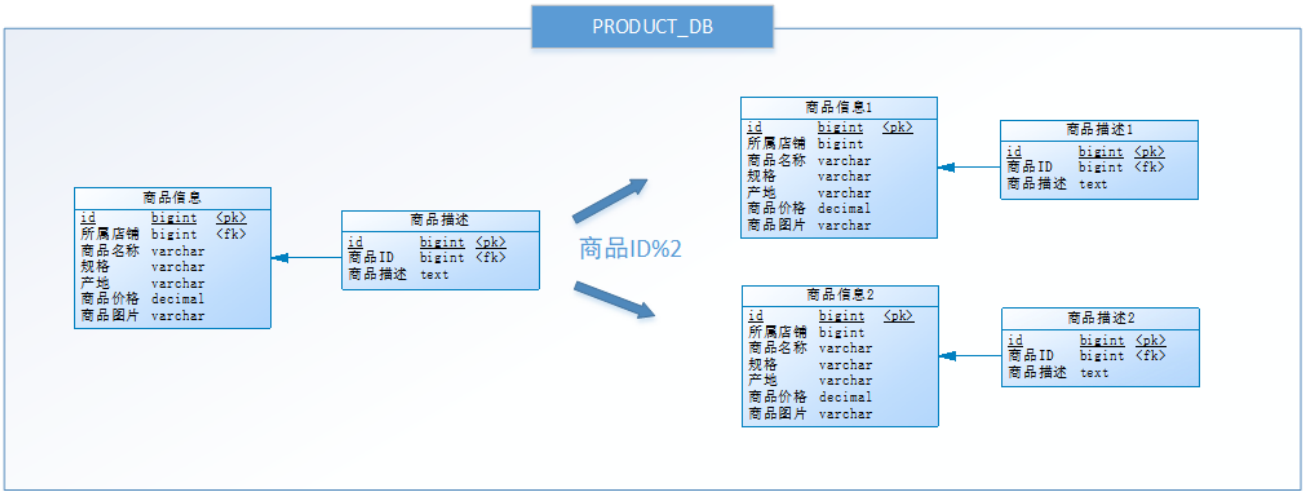
它带来的提升是：

- 解决了单库大数据，高并发的性能瓶颈。
- 提高了系统的稳定性及可用性。

当一个应用难以再细粒度的垂直切分，或切分后数据量行数巨大，存在单库读写、存储性能瓶颈，这时候就需要进行**水平分库**了，经过水平切分的优化，往往能解决单库存储量及性能瓶颈。但由于同一个表被分配在不同的数据库，需要额外进行数据操作的路由工作，因此大大提升了系统复杂度。

1.2.4.水平分表

按照水平分库的思路对他把PRODUCT_DB_X(商品库)内的表也可以进行水平拆分，其目的也是为解决单表数据量大的问题，如下图：



与水平分库的思路类似，不过这次操作的目标是表，商品信息及商品描述被分成了两套表。如果商品ID为双数，将此操作映射至商品信息1表；如果商品ID为单数，将操作映射至商品信息2表。此操作要访问表名称的表达式为**商品信息[商品ID%2 + 1]** 。

小明进行的这一步优化，就叫**水平分表**。

水平分表是在同一个数据库内，把同一个表的数据按一定规则拆到多个表中。

它带来的提升是：

- 优化单一表数据量过大而产生的性能问题
- 避免IO争抢并减少锁表的几率

库内的水平分表，解决了单一表数据量过大的问题，分出来的小表中只包含一部分数据，从而使得单个表的数据量变小，提高检索性能。

1.2.5 小结

本章介绍了分库分表的各种方式，它们分别是垂直分表、垂直分库、水平分库和水平分表：

垂直分表：可以把一个宽表的字段按访问频次、是否是大字段的原则拆分为多个表，这样既能使业务清晰，还能提升部分性能。拆分后，尽量从业务角度避免联查，否则性能方面将得不偿失。

垂直分库：可以把多个表按业务耦合松紧归类，分别存放在不同的库，这些库可以分布在不同服务器，从而使访问压力被多服务器负载，大大提升性能，同时能提高整体架构的业务清晰度，不同的业务库可根据自身情况定制优化方案。但是它需要解决跨库带来的所有复杂问题。

水平分库：可以把一个表的数据(按数据行)分到多个不同的库，每个库只有这个表的部分数据，这些库可以分布在不同服务器，从而使访问压力被多服务器负载，大大提升性能。它不仅需要解决跨库带来的所有复杂问题，还要解决数据路由的问题(数据路由问题后边介绍)。

水平分表：可以把一个表的数据(按数据行)分到多个同一个数据库的多张表中，每个表只有这个表的部分数据，这样做能小幅提升性能，它仅仅作为水平分库的一个补充优化。

一般来说，在系统设计阶段就应该根据业务耦合松紧来确定垂直分库，垂直分表方案，在数据量及访问压力不是特别大的情况，首先考虑缓存、读写分离、索引技术等方案。若数据量极大，且持续增长，再考虑水平分库水平分表方案。

1.3.分库分表带来的问题

分库分表能有效的缓解了单机和单库带来的性能瓶颈和压力，突破网络IO、硬件资源、连接数的瓶颈，同时也带来了一些问题。

1.3.1.事务一致性问题

由于分库分表把数据分布在不同库甚至不同服务器，不可避免会带来**分布式事务**问题。

1.3.2.跨节点关联查询

在没有分库前，我们检索商品时可以通过以下SQL对店铺信息进行关联查询：

```
SELECT p.*,r.[地理区域名称],s.[店铺名称],s.[信誉]
FROM [商品信息] p
LEFT JOIN [地理区域] r ON p.[产地] = r.[地理区域编码]
LEFT JOIN [店铺信息] s ON p.id = s.[所属店铺]
WHERE...ORDER BY...LIMIT...
```

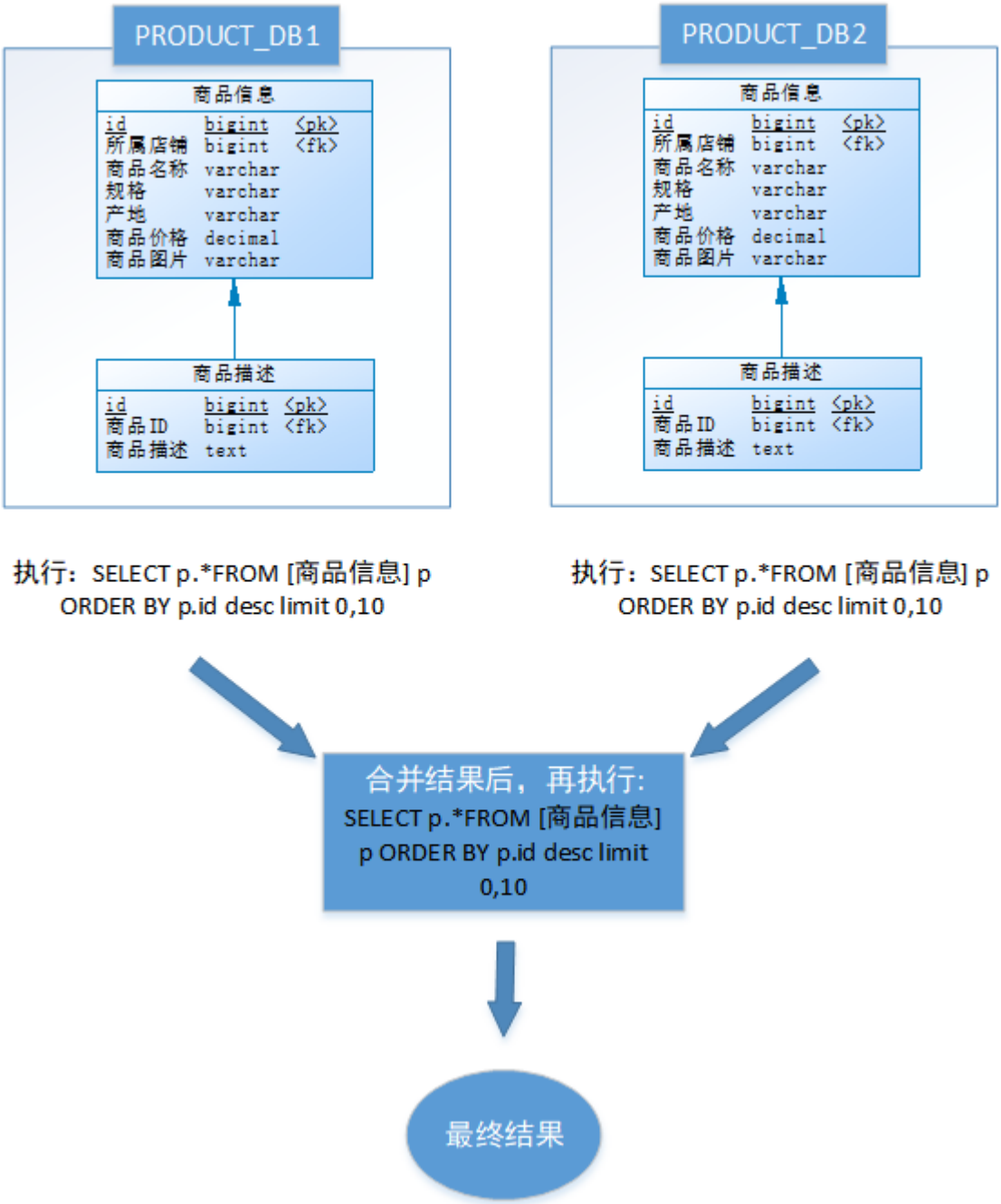
但垂直分库后**[商品信息]**和**[店铺信息]**不在一个数据库，甚至不在一台服务器，无法进行关联查询。

可将原关联查询分为两次查询，第一次查询的结果集中找出关联数据id，然后根据id发起第二次请求得到关联数据，最后将获得到的数据进行拼装。

1.3.3.跨节点分页、排序函数

跨节点多库进行查询时，limit分页、order by排序等问题，就变得比较复杂了。需要先在不同的分片节点中将数据进行排序并返回，然后将不同分片返回的结果集进行汇总和再次排序。

如，进行水平分库后的商品库，按ID倒序排序分页，取第一页：

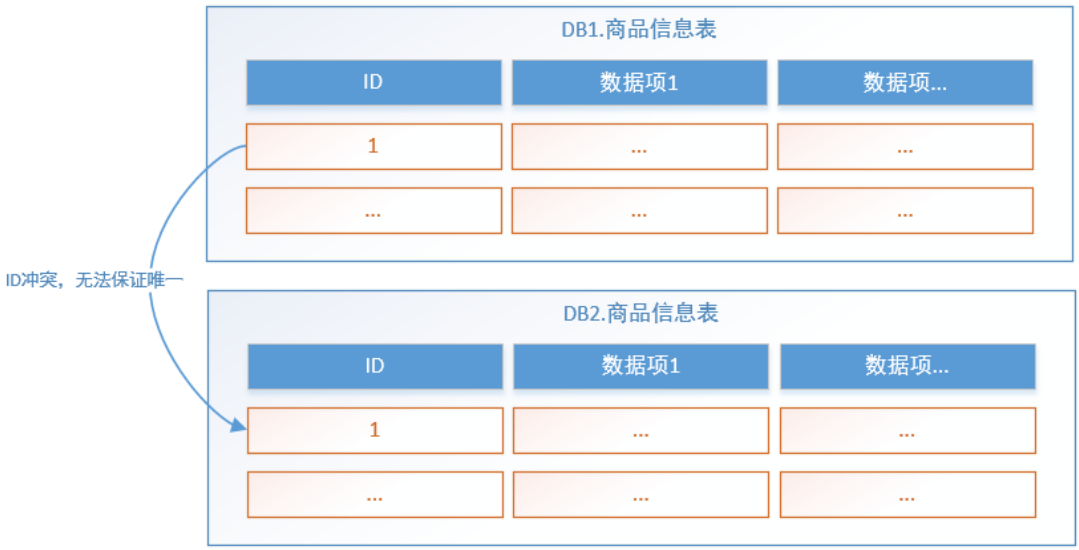


以上流程是取第一页的数据，性能影响不大，但由于商品信息的分布在各数据库的数据可能是随机的，如果是取第N页，需要将所有节点前N页数据都取出来合并，再进行整体的排序，操作效率可想而知。所以请求页数越大，系统的性能也会越差。

在使用Max、Min、Sum、Count之类的函数进行计算的时候，与排序分页同理，也需要先在每个分片上执行相应的函数，然后将各个分片的结果集进行汇总和再次计算，最终将结果返回。

1.3.4.主键避重

在分库分表环境中，由于表中数据同时存在不同数据库中，主键值平时使用的自增长将无用武之地，某个分区数据库生成的ID无法保证全局唯一。因此需要单独设计全局主键，以避免跨库主键重复问题。



1.3.5.公共表

实际的应用场景中，参数表、数据字典表等都是数据量较小，变动少，而且属于高频联合查询的依赖表。例子中**地理区域表**也属于此类型。

可以将这类表在每个数据库都保存一份，所有对公共表的更新操作都同时发送到所有分库执行。

由于分库分表之后，数据被分散在不同的数据库、服务器。因此，对数据的操作也就无法通过常规方式完成，并且它还带来了一系列的问题。好在，这些问题不是所有都需要我们在应用层面上解决，市面上有很多中间件可供我们选择，其中Sharding-JDBC使用流行度较高，我们来了解一下它。

1.4 Sharding-JDBC介绍

1.4.1 Sharding-JDBC介绍

Sharding-JDBC是当当网研发的开源分布式数据库中间件，从 3.0 开始Sharding-JDBC被包含在 Sharding-Sphere 中，之后该项目进入进入Apache孵化器，4.0版本之后的版本为Apache版本。

ShardingSphere是一套开源的分布式数据库中间件解决方案组成的生态圈，它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（计划中）这3款相互独立的产品组成。 他们均提供标准化的数据分片、分布式事务和数据库治理功能，可适用于如Java同构、异构语言、容器、云原生等各种多样化的应用场景。

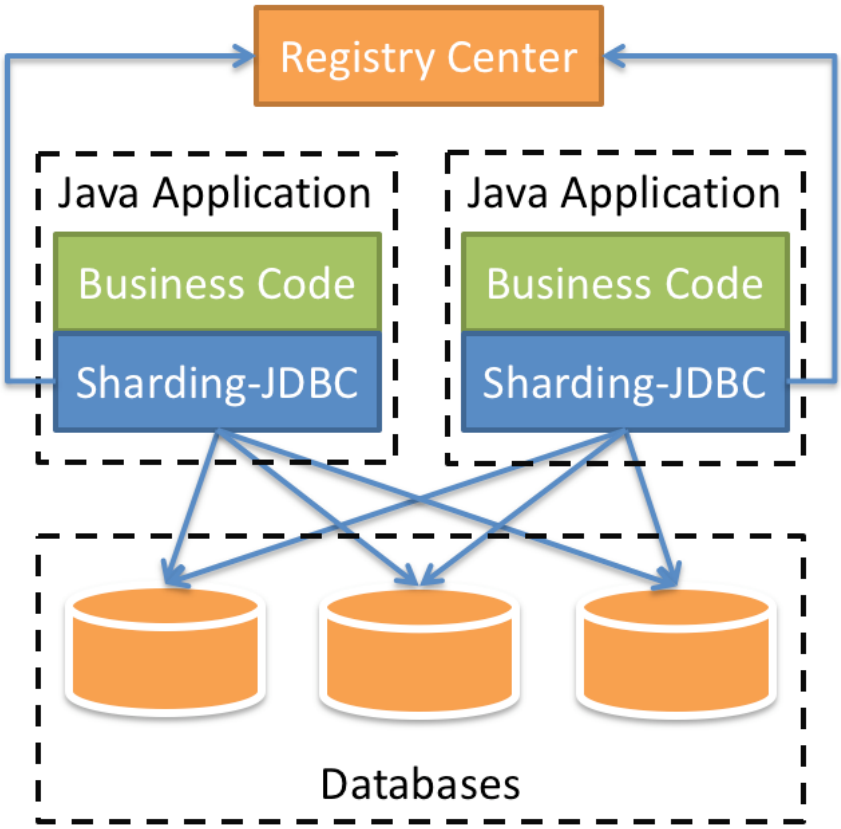
官方地址：https://shardingsphere.apache.org/document/current/cn/overview/

咱们目前只需关注Sharding-JDBC，它定位为轻量级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。

Sharding-JDBC的核心功能为**数据分片**和**读写分离**，通过Sharding-JDBC，应用可以**透明**的使用jdbc访问已经分库分表、读写分离的多个数据源，而不用关心数据源的数量以及数据如何分布。

- 适用于任何基于Java的ORM框架，如： Hibernate, Mybatis, Spring JDBC Template或直接使用JDBC。

- 基于任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, Druid, HikariCP等。
- 支持任意实现JDBC规范的数据库。目前支持MySQL，Oracle，SQLServer和PostgreSQL。



上图展示了Sharding-Jdbc的工作方式，使用Sharding-Jdbc前需要人工对数据库进行分库分表，在应用程序中加入Sharding-Jdbc的Jar包，应用程序通过Sharding-Jdbc操作分库分表后的数据库和数据表，由于Sharding-Jdbc是对Jdbc驱动的增强，使用Sharding-Jdbc就像使用Jdbc驱动一样，在应用程序中是无需指定具体要操作的分库和分表的。

1.4.2 与jdbc性能对比

1. 性能损耗测试：服务器资源充足、并发数相同，比较JDBC和Sharding-JDBC性能损耗，Sharding-JDBC相对JDBC损耗不超过7%。

基准测试性能对比

业务场景	JDBC	Sharding-JDBC1.5.2	Sharding-JDBC1.5.2/JDBC损耗
单库单表查询	493	470	4.7%
单库单表更新	6682	6303	5.7%
单库单表插入	6855	6375	7%

业务场景	业务平均响应时间(ms)	业务TPS
JDBC单库单表查询	7	493
Sharding-JDBC 1.5.2单库单表查询	8	470

2. 性能对比测试：服务器资源使用到极限，相同的场景JDBC与Sharding-JDBC的吞吐量相当。
3. 性能对比测试：服务器资源使用到极限，Sharding-JDBC采用分库分表后，Sharding-JDBC吞吐量较JDBC不分表有接近2倍的提升。

JDBC单库两库表与Sharding-JDBC两库各两表对比

业务场景	JDBC单库两表	Sharding-JDBC两库各两表	性能提升至
查询	1736	3331	192%
更新	9170	17997	196%
插入	11574	23043	199%

JDBC单库单表与Sharding-JDBC两库各一表对比

业务场景	JDBC单库单表	Sharding-JDBC两库各一表	性能提升至
查询	1586	2944	185%
更新	9548	18561	194%
插入	11182	21414	192%

2.Sharding-JDBC快速入门

2.1 需求说明

本章节使用Sharding-JDBC完成对订单表的水平分表，通过快速入门程序的开发，快速体验Sharding-JDBC的使用方法。

人工创建两张表，t_order_1和t_order_2，这两张表是订单表拆分后的表，通过Sharding-Jdbc向订单表插入数据，按照一定的分片规则，主键为偶数的进入t_order_1，另一部分数据进入t_order_2，通过Sharding-Jdbc 查询数据，根据 SQL语句的内容从t_order_1或t_order_2查询数据。

2.2.环境搭建

2.2.1 环境说明

- 操作系统：Win10
- 数据库：MySQL-5.7.25
- JDK：64位 jdk1.8.0_201
- 应用框架：spring-boot-2.1.3.RELEASE，Mybatis3.5.0
- Sharding-JDBC：sharding-jdbc-spring-boot-starter-4.0.0-RC1

2.2.2 创建数据库

创建订单库order_db

```
CREATE DATABASE `order_db` CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

在order_db中创建t_order_1、t_order_2表

```
DROP TABLE IF EXISTS `t_order_1`;
CREATE TABLE `t_order_1`  (
  `order_id` bigint(20) NOT NULL COMMENT '订单id',
  `price` decimal(10, 2) NOT NULL COMMENT '订单价格',
  `user_id` bigint(20) NOT NULL COMMENT '下单用户id',
  `status` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单状态',
  PRIMARY KEY (`order_id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;

DROP TABLE IF EXISTS `t_order_2`;
CREATE TABLE `t_order_2`  (
  `order_id` bigint(20) NOT NULL COMMENT '订单id',
  `price` decimal(10, 2) NOT NULL COMMENT '订单价格',
  `user_id` bigint(20) NOT NULL COMMENT '下单用户id',
  `status` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单状态',
  PRIMARY KEY (`order_id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

2.2.3.引入maven依赖

引入 sharding-jdbc和SpringBoot整合的Jar包：

```
<dependency>
  <groupId>org.apache.shardingsphere</groupId>
  <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
  <version>4.0.0-RC1</version>
</dependency>
```

具体spring boot相关依赖及配置请参考资料中dbsharding/sharding-jdbc-simple工程，本指引只说明与Sharding-JDBC相关的内容。

2.3 编写程序

2.3.1 分片规则配置

分片规则配置是sharding-jdbc进行对分库分表操作的重要依据，配置内容包括：数据源、主键生成策略、分片策略等。

在application.properties中配置

```
server.port=56081

spring.application.name = sharding-jdbc-simple-demo
```

```
server.servlet.context-path = /sharding-jdbc-simple-demo
spring.http.encoding.enabled = true
spring.http.encoding.charset = UTF-8
spring.http.encoding.force = true

spring.main.allow-bean-definition-overriding = true

mybatis.configuration.map-underscore-to-camel-case = true
# 以下是分片规则配置
# 定义数据源
spring.shardingsphere.datasource.names = m1

spring.shardingsphere.datasource.m1.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m1.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m1.url = jdbc:mysql://localhost:3306/order_db?useUnicode=true
spring.shardingsphere.datasource.m1.username = root
spring.shardingsphere.datasource.m1.password = root

# 指定t_order表的数据分布情况，配置数据节点
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes = m1.t_order_$->{1..2}

# 指定t_order表的主键生成策略为SNOWFLAKE
spring.shardingsphere.sharding.tables.t_order.key-generator.column=order_id
spring.shardingsphere.sharding.tables.t_order.key-generator.type=SNOWFLAKE



# 指定t_order表的分片策略，分片策略包括分片键和分片算法
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.sharding-column = order_id
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.algorithm-expression =
t_order_$->{order_id % 2 + 1}

# 打开sql输出日志
spring.shardingsphere.props.sql.show = true

swagger.enable = true

logging.level.root = info
logging.level.org.springframework.web = info
logging.level.com.itheima.dbsharding  = debug
logging.level.druid.sql = debug
```

1.首先定义数据源m1，并对m1进行实际的参数配置。

2.指定t_order表的数据分布情况，他分布在m1.t_order_1，m1.t_order_2

3.指定t_order表的主键生成策略为SNOWFLAKE，SNOWFLAKE是一种分布式自增算法，保证id全局唯一

4.定义t_order分片策略，order_id为偶数的数据落在t_order_1，为奇数的落在t_order_2，分表策略的表达式为t_order_$->{order_id % 2 + 1}

2.3.2.数据操作

```
@Mapper
@Component
public interface OrderDao {

    /**
     * 新增订单
     * @param price 订单价格
     * @param userId 用户id
     * @param status 订单状态
     * @return
     */
    @Insert("insert into t_order(price,user_id,status) value(#{price},#{userId},#{status})")
    int insertOrder(@Param("price") BigDecimal price, @Param("userId")Long userId,
@Param("status")String status);

    /**
     * 根据id列表查询多个订单
     * @param orderIds 订单id列表
     * @return
     */
    @Select({"<script>" +
            "select " +
            " * " +
            " from t_order t" +
            " where t.order_id in " +
            "<foreach collection='orderIds' item='id' open='(' separator=',' close=')'>" +
            " #{id} " +
            "</foreach>"+
            "</script>"})
    List<Map> selectOrderbyIds(@Param("orderIds")List<Long> orderIds);

}
```

2.3.3.测试

编写单元测试：

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {ShardingJdbcSimpleDemoBootstrap.class})
public class OrderDaoTest {
    @Autowired
    private OrderDao orderDao;
    @Test
    public void testInsertOrder(){
        for (int i = 0 ; i<10; i++){
            orderDao.insertOrder(new BigDecimal((i+1)*5),1L,"WAIT_PAY");
        }

    }
```

```
    @Test
    public void testSelectOrderbyIds(){
        List<Long> ids = new ArrayList<>();
        ids.add(373771636085620736L);
        ids.add(373771635804602369L);
        List<Map> maps = orderDao.selectOrderbyIds(ids);
        System.out.println(maps);
    }
}
```

执行testInsertOrder：

```
Started OrderDaoTest in 1.641 seconds (JVM running for 2.121)
==>  Preparing: insert into t_order(price,user_id,status) value(?,?,?)
==> Parameters: 5(BigDecimal), 1(Long), WAIT_PAY(String)
Rule Type: sharding
Logic SQL: insert into t_order(price,user_id,status) value(?,?,?)
SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optiona
Actual SQL: m1 ::: insert into t_order_2 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [5, 1, WAIT_PAY, 370964610859139073]
<==    Updates: 1
==>  Preparing: insert into t_order(price,user_id,status) value(?,?,?)
==> Parameters: 10(BigDecimal), 1(Long), WAIT_PAY(String)
Rule Type: sharding
Logic SQL: insert into t_order(price,user_id,status) value(?,?,?)
SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optiona
Actual SQL: m1 ::: insert into t_order_1 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [10, 1, WAIT_PAY, 370964611354066944]
<==    Updates: 1
```

通过日志可以发现order_id为奇数的被插入到t_order_2表，为偶数的被插入到t_order_1表，达到预期目标。

执行testSelectOrderbyIds：

```
: Started OrderDaoTest in 1.664 seconds (JVM running for 2.144)
: ==>  Preparing: select * from t_order t where t.order_id in ( ? , ? )
: ==> Parameters: 370969501279191040(Long), 370969500767485953(Long)
: Rule Type: sharding
: Logic SQL: select  *  from t_order t  where t.order_id in  (  ?  ,  ?  )
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_order, alias=Optional.of
: Actual SQL: m1 ::: select  *  from t_order_1 t  where t.order_id in  (  ?  ,  ?  ) ::: [370969501279191040, 370969500767485953]
: Actual SQL: m1 ::: select  *  from t_order_2 t  where t.order_id in  (  ?  ,  ?  ) ::: [370969501279191040, 370969500767485953]
: <==      Total: 2
```

通过日志可以发现，根据传入order_id的奇偶不同，sharding-jdbc分别去不同的表检索数据，达到预期目标。

2.4.流程分析

通过日志分析，Sharding-JDBC在拿到用户要执行的sql之后干了哪些事儿：

（1）解析sql，获取片键值，在本例中是order_id

（2）Sharding-JDBC通过规则配置 t_order_$->{order_id % 2 + 1}，知道了当order_id为偶数时，应该往t_order_1表插数据，为奇数时，往t_order_2插数据。

（3）于是Sharding-JDBC根据order_id的值改写sql语句，改写后的SQL语句是真实所要执行的SQL语句。

（4）执行改写后的真实sql语句

（5）将所有真正执行sql的结果进行汇总合并，返回。

2.5.其他集成方式

Sharding-JDBC不仅可以与spring boot良好集成，它还支持其他配置方式，共支持以下四种集成方式。

Spring Boot Yaml 配置

定义application.yml，内容如下：

```yaml
server:
  port: 56081
  servlet:
    context-path: /sharding-jdbc-simple-demo
spring:
  application:
    name: sharding-jdbc-simple-demo
  http:
    encoding:
      enabled: true
      charset: utf-8
      force: true
  main:
    allow-bean-definition-overriding: true
  shardingsphere:
    datasource:
      names: m1
      m1:
        type: com.alibaba.druid.pool.DruidDataSource
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/order_db?useUnicode=true
        username: root
        password: mysql
    sharding:
      tables:
        t_order:
          actualDataNodes: m1.t_order_$->{1..2}
          tableStrategy:
            inline:
              shardingColumn: order_id
              algorithmExpression: t_order_$->{order_id % 2 + 1}
          keyGenerator:
            type: SNOWFLAKE
            column: order_id
    props:
      sql:
        show: true
mybatis:
  configuration:
    map-underscore-to-camel-case: true
swagger:
  enable: true
logging:
  level:

    root: info
```

```
    org.springframework.web: info
    com.itheima.dbsharding: debug
    druid.sql: debug
```

如果使用application.yml则需要屏蔽原来的application.properties文件。

Java 配置

添加配置类：

```
 @Configuration
public class ShardingJdbcConfig {

    // 定义数据源
    Map<String, DataSource> createDataSourceMap() {
        DruidDataSource dataSource1 = new DruidDataSource();
        dataSource1.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource1.setUrl("jdbc:mysql://localhost:3306/order_db?useUnicode=true");
        dataSource1.setUsername("root");
        dataSource1.setPassword("root");
        Map<String, DataSource> result = new HashMap<>();
        result.put("m1", dataSource1);
        return result;
    }
    // 定义主键生成策略
    private static KeyGeneratorConfiguration getKeyGeneratorConfiguration() {
        KeyGeneratorConfiguration result = new
KeyGeneratorConfiguration("SNOWFLAKE","order_id");
        return result;
    }

    // 定义t_order表的分片策略
    TableRuleConfiguration getOrderTableRuleConfiguration() {
        TableRuleConfiguration result = new TableRuleConfiguration("t_order","m1.t_order_$->
{1..2}");
        result.setTableShardingStrategyConfig(new
InlineShardingStrategyConfiguration("order_id", "t_order_$->{order_id % 2 + 1}"));
        result.setKeyGeneratorConfig(getKeyGeneratorConfiguration());

        return result;
    }
    // 定义sharding-Jdbc数据源
    @Bean
    DataSource getShardingDataSource() throws SQLException {
        ShardingRuleConfiguration shardingRuleConfig = new ShardingRuleConfiguration();
        shardingRuleConfig.getTableRuleConfigs().add(getOrderTableRuleConfiguration());
        //spring.shardingsphere.props.sql.show = true
        Properties properties = new Properties();
        properties.put("sql.show","true");
        return ShardingDataSourceFactory.createDataSource(createDataSourceMap(),
```

```
shardingRuleConfig,properties);
    }
}
```

由于采用了配置类所以需要屏蔽原来application.properties文件中spring.shardingsphere开头的配置信息。

还需要在SpringBoot启动类中屏蔽使用spring.shardingsphere配置项的类：

```
@SpringBootApplication(exclude = {SpringBootConfiguration.class})
public class ShardingJdbcSimpleDemoBootstrap {....}
```

Spring Boot properties配置

此方式同快速入门程序。

```
# 定义数据源
spring.shardingsphere.datasource.names = m1

spring.shardingsphere.datasource.m1.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m1.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m1.url = jdbc:mysql://localhost:3306/order_db?useUnicode=true
spring.shardingsphere.datasource.m1.username = root
spring.shardingsphere.datasource.m1.password = root


# 指定t_order表的主键生成策略为SNOWFLAKE
spring.shardingsphere.sharding.tables.t_order.key-generator.column=order_id
spring.shardingsphere.sharding.tables.t_order.key-generator.type=SNOWFLAKE

# 指定t_order表的数据分布情况
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes = m1.t_order_$->{1..2}

# 指定t_order表的分表策略
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.sharding-column = order_id
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.algorithm-expression =
t_order_$->{order_id % 2 + 1}
```

Spring命名空间配置

此方式使用xml方式配置，不推荐使用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:sharding="http://shardingsphere.apache.org/schema/shardingsphere/sharding"
```

```
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans.xsd
                            http://shardingsphere.apache.org/schema/shardingsphere/sharding

 http://shardingsphere.apache.org/schema/shardingsphere/sharding/sharding.xsd
                            http://www.springframework.org/schema/context
                            http://www.springframework.org/schema/context/spring-context.xsd
                            http://www.springframework.org/schema/tx
                            http://www.springframework.org/schema/tx/spring-tx.xsd">
    <context:annotation-config />

    <!--定义多个数据源-->
    <bean id="m1" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/order_db_1?useUnicode=true" />
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>

   <!--定义分库策略-->
    <sharding:inline-strategy id="tableShardingStrategy" sharding-column="order_id" algorithm-
expression="t_order_$->{order_id % 2 + 1}" />

   <!--定义主键生成策略-->
    <sharding:key-generator id="orderKeyGenerator" type="SNOWFLAKE" column="order_id" />

    <!--定义sharding-Jdbc数据源-->
    <sharding:data-source id="shardingDataSource">
        <sharding:sharding-rule data-source-names="m1">
            <sharding:table-rules>
                <sharding:table-rule logic-table="t_order"  table-strategy-
ref="tableShardingStrategy" key-generator-ref="orderKeyGenerator" />
            </sharding:table-rules>
        </sharding:sharding-rule>
    </sharding:data-source>
</beans>
```

3.Sharding-JDBC执行原理

3.1 基本概念

在了解Sharding-JDBC的执行原理前，需要了解以下概念：

逻辑表

水平拆分的数据表的总称。例：订单数据表根据主键尾数拆分为10张表，分别是`t_order_0`、`t_order_1`到`t_order_9`，他们的逻辑表名为`t_order`。

真实表

在分片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`。

数据节点

数据分片的最小物理单元。由数据源名称和数据表组成，例：`ds_0.t_order_0`。

绑定表

指分片规则一致的主表和子表。例如：`t_order`表和`t_order_item`表，均按照`order_id`分片,绑定表之间的分区键完全相同，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);
```

广播表

指所有的分片数据源中都存在的表，表结构和表中的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。

分片键

用于分片的数据库字段，是将数据库(表)水平拆分的关键字段。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，Sharding-Jdbc也支持根据多个字段进行分片。

分片算法

通过分片算法将数据分片，支持通过`=`、`BETWEEN`和`IN`分片。分片算法需要应用方开发者自行实现，可实现的灵活度非常高。包括：精确分片算法 、范围分片算法 ，复合分片算法 等。例如：where order_id = ? 将采用精确分片算法，where order_id in (?,?,?)将采用精确分片算法，where order_id BETWEEN ? and ? 将采用范围分片算法，复合分片算法用于分片键有多个复杂情况。

分片策略

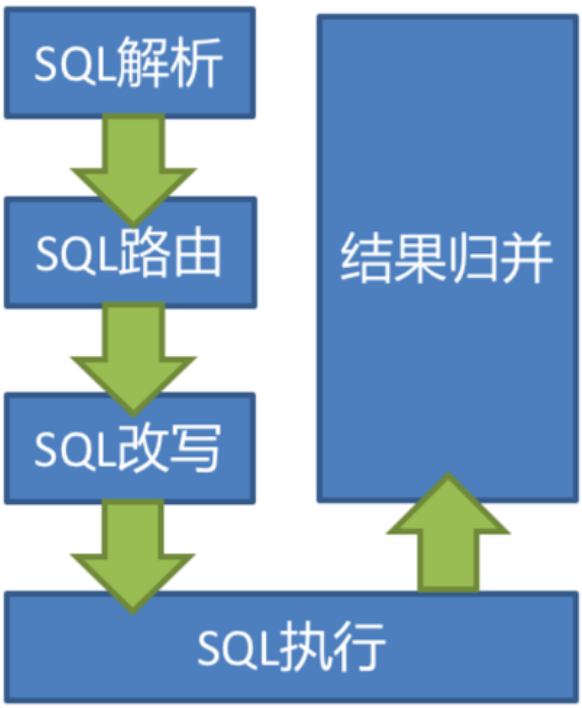
包含分片键和分片算法，由于分片算法的独立性，将其独立抽离。真正可用于分片操作的是分片键 + 分片算法，也就是分片策略。内置的分片策略大致可分为尾数取模、哈希、范围、标签、时间等。由用户方配置的分片策略则更加灵活，常用的使用行表达式配置分片策略，它采用Groovy表达式表示，如: `t_user_$->{u_id % 8}` 表示t_user表根据u_id模8，而分成8张表，表名称为`t_user_0`到`t_user_7`。

自增主键生成策略

通过在客户端生成自增主键替换以数据库原生自增主键的方式，做到分布式主键无重复。

3.2.SQL解析

当Sharding-JDBC接受到一条SQL语句时，会陆续执行`SQL解析 => 查询优化 => SQL路由 => SQL改写 => SQL执行 => 结果归并`，最终返回执行结果。

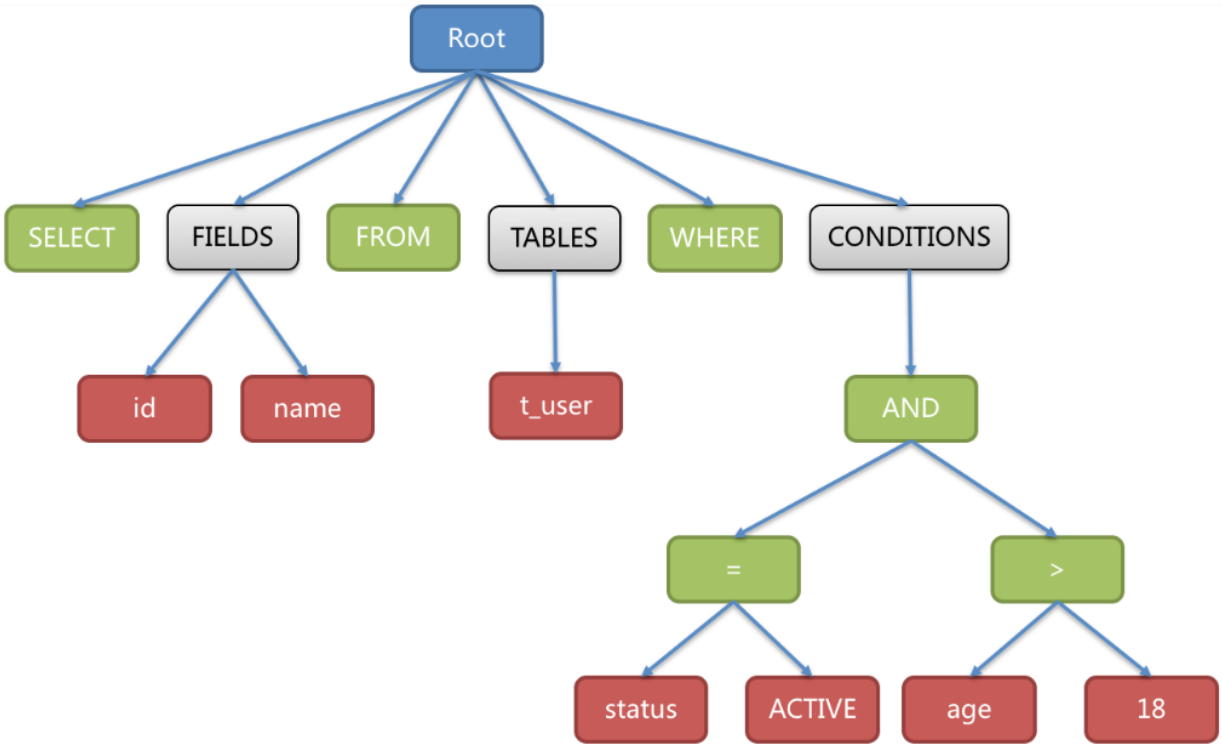


SQL解析过程分为**词法解析**和**语法解析**。 词法解析器用于将SQL拆解为不可再分的原子符号，称为Token。并根据不同数据库方言所提供的字典，将其归类为关键字，表达式，字面量和操作符。 再使用语法解析器将SQL转换为抽象语法树。

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，通过对抽象语法树的遍历去提炼分片所需的**上下文**，并标记有可能需要**SQL改写(后边介绍)**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）。

3.3.SQL路由

SQL路由就是把针对**逻辑表**的数据操作映射到对数据结点操作的过程。

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，根据分片键操作符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。根据分片键进行路由的场景可分为直接路由、标准路由、笛卡尔路由等。

标准路由

标准路由是Sharding-Jdbc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。

全库表路由

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。其中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(数据查询)和DML（数据操纵），以及DDL（数据定义）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);
```

3.4.SQL改写

工程师面向**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。

如一个简单的例子，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;
```

再比如，Sharding-JDBC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;
```

3.5.SQL执行

Sharding-JDBC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，他能在以下两种模式自适应切换：

内存限制模式

使用此模式的前提是，Sharding-JDBC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。

连接限制模式

使用此模式的前提是，Sharding-JDBC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。

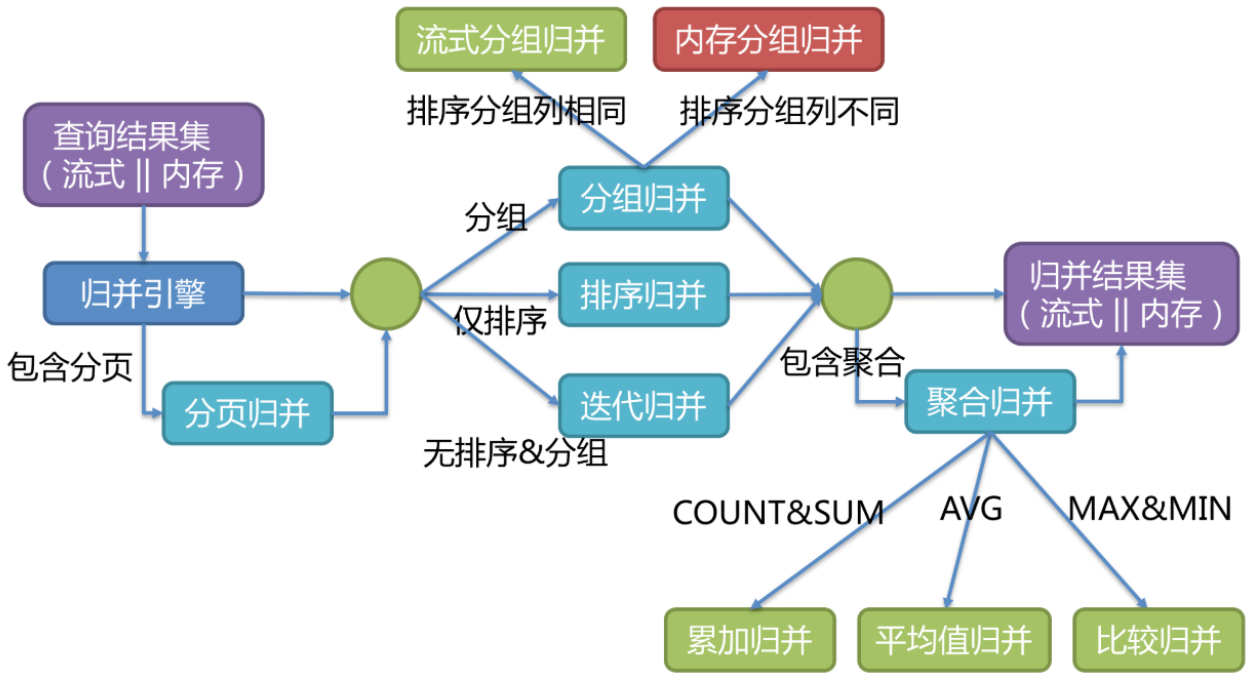
内存限制模式适用于OLAP操作，可以通过放宽对数据库连接的限制提升系统吞吐量； 连接限制模式适用于OLTP操作，OLTP通常带有分片键，会路由到单一的分片，因此严格控制数据库连接，以保证在线系统数据库资源能够被更多的应用所使用，是明智的选择。

3.6.结果归并

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。

Sharding-JDBC支持的结果归并从**功能**上可分为**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。

归并引擎的整体结构划分如下图。

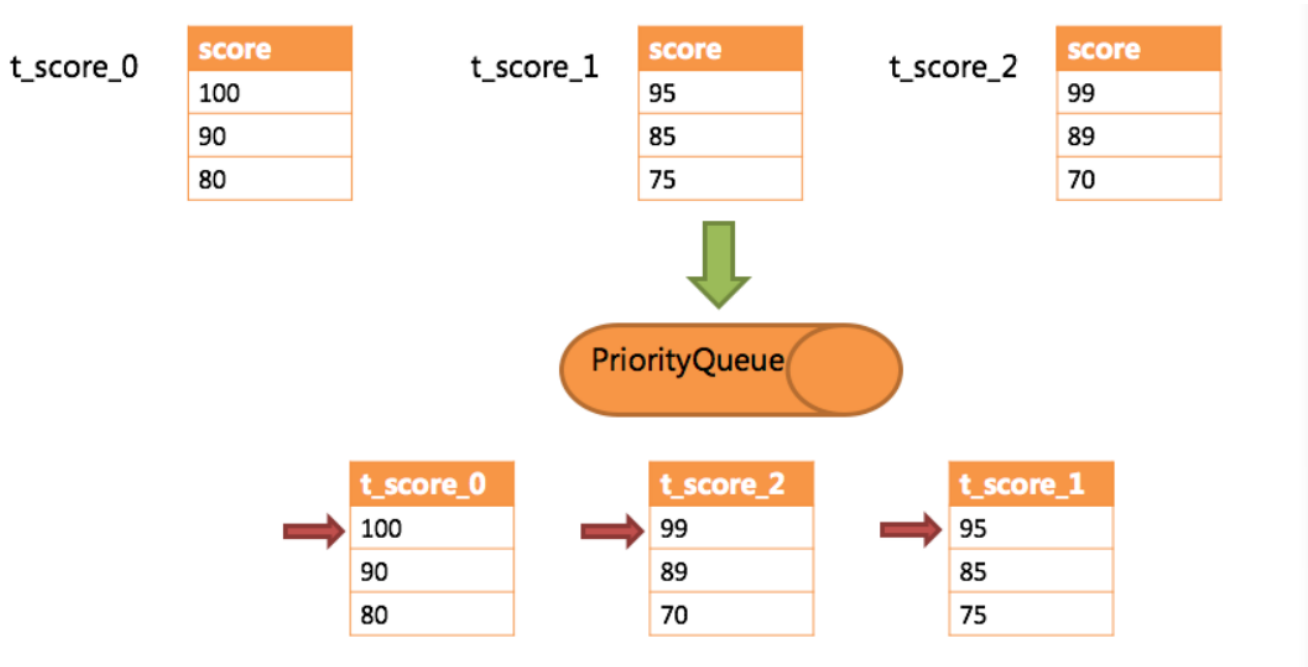


结果归并从结构划分可分为**流式归并**、**内存归并**和**装饰者归并**。流式归并和内存归并是互斥的，装饰者归并可以在流式归并和内存归并之上做进一步的处理。

内存归并很容易理解，他是将所有分片结果集的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。

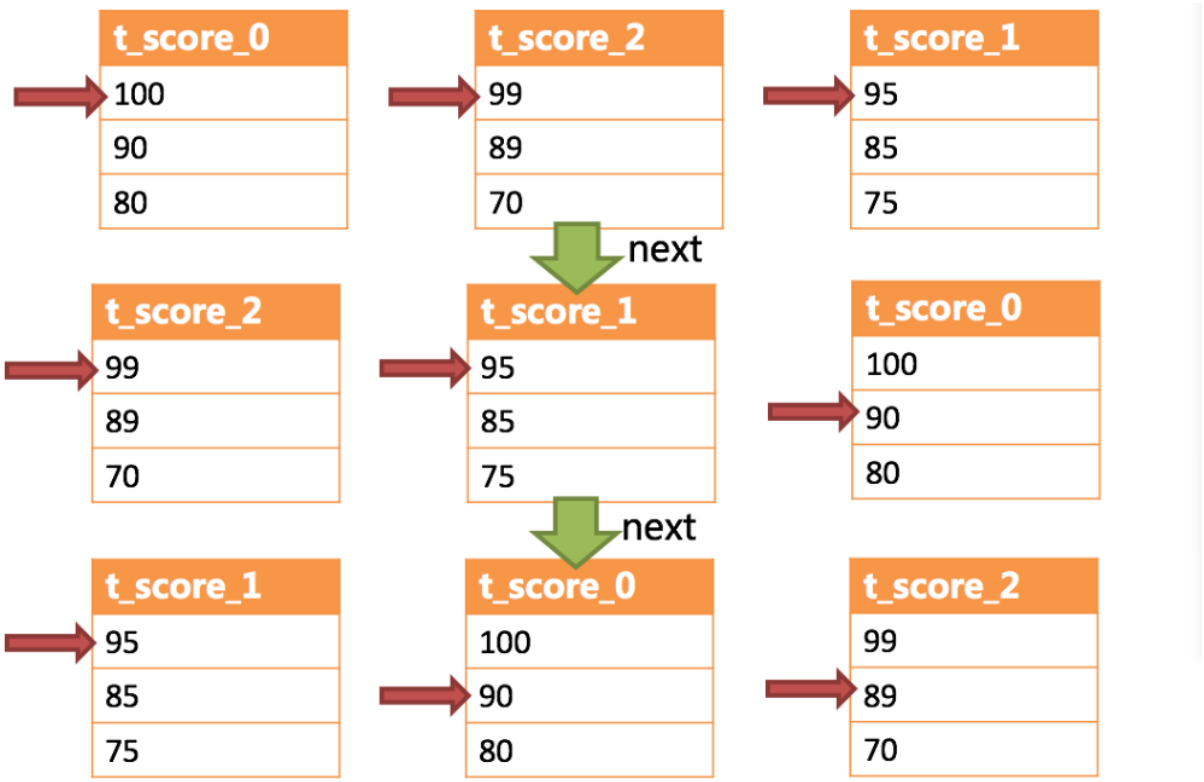
流式归并是指每一次从数据库结果集中获取到的数据，都能够通过游标逐条获取的方式返回正确的单条数据，它与数据库原生的返回结果集的方式最为契合。

下边举例说明排序归并的过程，如下图是一个通过分数进行排序的示例图，它采用流式归并方式。 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，Sharding-JDBC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。

装饰者归并是对所有的结果集归并进行统一的功能增强，比如归并时需要聚合SUM前，在进行聚合计算前，都会通过内存归并或流式归并查询出结果集。因此，聚合归并是在之前介绍的归并类型之上追加的归并能力，即装饰者模式。

3.7 总结

通过以上内容介绍，相信大家已经了解到Sharding-JDBC基础概念、核心功能以及执行原理。

基础概念：逻辑表，真实表，数据节点，绑定表，广播表，分片键，分片算法，分片策略，主键生成策略

核心功能：数据分片，读写分离

执行流程：`SQL解析 => 查询优化 => SQL路由 => SQL改写 => SQL执行 => 结果归并`

接下来我们将通过一个个demo，来演示Sharding-JDBC实际使用方法。

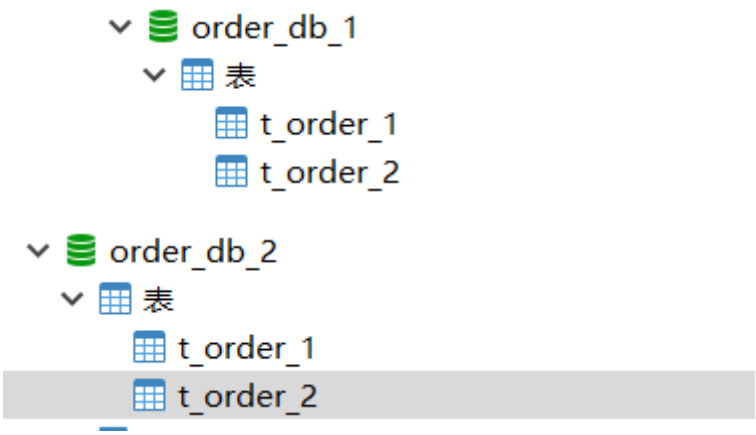
4.水平分表

前面已经介绍过，**水平分表**是在同一个数据库内，把同一个表的数据按一定规则拆到多个表中。在**快速入门**里，我们已经对水平分库进行实现，这里不再重复介绍。

5.水平分库

前面已经介绍过，**水平分库**是把同一个表的数据按一定规则拆到不同的数据库中，每个库可以放在不同的服务器上。接下来看一下如何使用Sharding-JDBC实现水平分库，咱们继续对**快速入门**中的例子进行完善。

(1)将原有order_db库拆分为order_db_1、order_db_2



(2)分片规则修改

由于数据库拆分了两个，这里需要配置两个数据源。

分库需要配置分库的策略，和分表策略的意义类似，通过分库策略实现数据操作针对分库的数据库进行操作。

```
# 定义多个数据源
spring.shardingsphere.datasource.names = m1,m2

spring.shardingsphere.datasource.m1.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m1.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m1.url = jdbc:mysql://localhost:3306/order_db_1?useUnicode=true
spring.shardingsphere.datasource.m1.username = root
spring.shardingsphere.datasource.m1.password = root

spring.shardingsphere.datasource.m2.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m2.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m2.url = jdbc:mysql://localhost:3306/order_db_2?useUnicode=true
spring.shardingsphere.datasource.m2.username = root
spring.shardingsphere.datasource.m2.password = root
...
# 分库策略，以user_id为分片键，分片策略为user_id % 2 + 1，user_id为偶数操作m1数据源，否则操作m2。
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.sharding-column = user_id
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.algorithm-expression =
m$->{user_id % 2 + 1}
```

分库策略定义方式如下：

```
#分库策略，如何将一个逻辑表映射到多个数据源
spring.shardingsphere.sharding.tables.<逻辑表名称>.database-strategy.<分片策略>.<分片策略属性名>= #分片策略属性值

#分表策略，如何将一个逻辑表映射为多个实际表
spring.shardingsphere.sharding.tables.<逻辑表名称>.table-strategy.<分片策略>.<分片策略属性名>= #分片策略属性值
```

Sharding-JDBC支持以下几种分片策略：

不管理分库还是分表，策略基本一样。

- **standard**：标准分片策略，对应StandardShardingStrategy。提供对SQL语句中的=, IN和BETWEEN AND的分片操作支持。StandardShardingStrategy只支持单分片键，提供PreciseShardingAlgorithm和RangeShardingAlgorithm两个分片算法。PreciseShardingAlgorithm是必选的，用于处理=和IN的分片。RangeShardingAlgorithm是可选的，用于处理BETWEEN AND分片，如果不配置RangeShardingAlgorithm，SQL中的BETWEEN AND将按照全库路由处理。
- **complex**：符合分片策略，对应ComplexShardingStrategy。复合分片策略。提供对SQL语句中的=, IN和BETWEEN AND的分片操作支持。ComplexShardingStrategy支持多分片键，由于多分片键之间的关系复杂，因此并未进行过多的封装，而是直接将分片键值组合以及分片操作符透传至分片算法，完全由应用开发者实现，提供最大的灵活度。
- **inline**：行表达式分片策略，对应InlineShardingStrategy。使用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`。
- **hint**：Hint分片策略，对应HintShardingStrategy。通过Hint而非SQL解析的方式分片的策略。对于分片字段非SQL决定，而由其他外置条件决定的场景，可使用SQL Hint灵活的注入分片字段。例：内部系统，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(待实现)两种方式使用。
- **none**：不分片策略，对应NoneShardingStrategy。不分片的策略。

目前例子中都使用inline分片策略，若对其他分片策略细节若感兴趣，请查阅官方文档：
https://shardingsphere.apache.org

(3)插入测试

修改testInsertOrder方法，插入数据中包含不同的user_id

```
@Test
public void testInsertOrder(){
    for (int i = 0 ; i<10; i++){
        orderDao.insertOrder(new BigDecimal((i+1)*5),1L,"WAIT_PAY");
    }
    for (int i = 0 ; i<10; i++){
        orderDao.insertOrder(new BigDecimal((i+1)*10),2L,"WAIT_PAY");
    }
}
```

执行testInsertOrder:

```
: ==>  Preparing: insert into t_order(price,user_id,status) value(?,?,?)
: ==> Parameters: 50(BigDecimal), 1(Long), WAIT_PAY(String)
: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) value(?,?,?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent())]), routeCo
: Actual SQL: m2 ::: insert into t_order_1 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [50, 1, WAIT_PAY, 370976701154328576]
: <==    Updates: 1
: ==>  Preparing: insert into t_order(price,user_id,status) value(?,?,?)
: ==> Parameters: 10(BigDecimal), 2(Long), WAIT_PAY(String)
: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) value(?,?,?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent())]), routeCo
: Actual SQL: m1 ::: insert into t_order_2 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [10, 2, WAIT_PAY, 370976701246603265]
: <==    Updates: 1
```

通过日志可以看出，根据user_id的奇偶不同，数据分别落在了不同数据源，达到目标。

（4）查询测试

调用快速入门的查询接口进行测试：

```
List<Map> selectOrderbyIds(@Param("orderIds")List<Long> orderIds);
```

通过日志发现，sharding-jdbc将sql路由到m1和m2：

```
Logic SQL: select  *  from t_order t  where t.order_id in  (  ?  ,  ?  )
SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_orde
Actual SQL: m1 ::: select  *  from t_order_1 t  where t.order_id in  (  ?  ,  ?  ) ::: [373067982407991296, 37306798240799
Actual SQL: m2 ::: select  *  from t_order_1 t  where t.order_id in  (  ?  ,  ?  ) ::: [373067982407991296, 37306798240799
```

问题分析：

由于查询语句中并没有使用分片键user_id，所以sharding-jdbc将广播路由到每个数据结点。

下边我们在sql中添加分片键进行查询。

在OrderDao中定义接口：

```
@Select({"<script>",
        " select",
        " * ",
        " from t_order t ",
        "where t.order_id in",
        "<foreach collection='orderIds' item='id' open='(' separator=',' close=')'>",
        "#{id}",
        "</foreach>",
        " and t.user_id = #{userId} ",
        "</script>"
})
List<Map> selectOrderbyUserAndIds(@Param("userId") Integer userId,@Param("orderIds")List<Long>
orderIds);
```

编写测试方法：

```
@Test
public void testSelectOrderbyUserAndIds(){
    List<Long> orderIds = new ArrayList<>();
    orderIds.add(373422416644276224L);
    orderIds.add(373422415830581248L);
    //查询条件中包括分库的键user_id
    int user_id = 1;
    List<Map> orders = orderDao.selectOrderbyUserAndIds(user_id,orderIds);
    JSONArray jsonOrders = new JSONArray(orders);
    System.out.println(jsonOrders);
}
```

执行testSelectOrderbyUserAndIds:

```
Logic SQL: select  *  from t_order t  where t.order_id in  (  ?  ,  ?  )  and t.user_id = ?
SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[
Actual SQL: m2 ::: select  *  from t_order_1 t  where t.order_id in  (  ?  ,  ?  )  and t.user_id = ? :::
<==      Total: 2
```

查询条件user_id为1，根据分片策略m$->{user_id % 2 + 1}计算得出m2，此sharding-jdbc将sql路由到m2，见上图日志。

6.垂直分库

前面已经介绍过，**垂直分库**是指按照业务将表进行分类，分布到不同的数据库上面，每个库可以放在不同的服务器上，它的核心理念是专库专用。接下来看一下如何使用Sharding-JDBC实现垂直分库。

(1)创建数据库

创建数据库user_db

```
CREATE DATABASE `user_db` CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

在user_db中创建t_user表

```
DROP TABLE IF EXISTS `t_user`;
CREATE TABLE `t_user`  (
  `user_id` bigint(20) NOT NULL COMMENT '用户id',
  `fullname` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '用户姓名',
  `user_type` char(1) DEFAULT NULL COMMENT '用户类型',
  PRIMARY KEY (`user_id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

(2)在Sharding-JDBC规则中修改

```
# 新增m0数据源，对应user_db
spring.shardingsphere.datasource.names = m0,m1,m2
...
spring.shardingsphere.datasource.m0.type = com.alibaba.druid.pool.DruidDataSource

spring.shardingsphere.datasource.m0.driver-class-name = com.mysql.jdbc.Driver
```

```
spring.shardingsphere.datasource.m0.url = jdbc:mysql://localhost:3306/user_db?useUnicode=true
spring.shardingsphere.datasource.m0.username = root
spring.shardingsphere.datasource.m0.password = root

....
# t_user分表策略，固定分配至m0的t_user真实表
spring.shardingsphere.sharding.tables.t_user.actual-data-nodes = m$->{0}.t_user
spring.shardingsphere.sharding.tables.t_user.table-strategy.inline.sharding-column = user_id
spring.shardingsphere.sharding.tables.t_user.table-strategy.inline.algorithm-expression = t_user
```

(3)数据操作

新增UserDao:

```
@Mapper
@Component
public interface UserDao {

    /**
     * 新增用户
     * @param userId 用户id
     * @param fullname 用户姓名
     * @return
     */
    @Insert("insert into t_user(user_id, fullname) value(#{userId},#{fullname})")
    int insertUser(@Param("userId")Long userId,@Param("fullname")String fullname);

    /**
     * 根据id列表查询多个用户
     * @param userIds 用户id列表
     * @return
     */
    @Select({"<script>",
            " select",
            " * ",
            " from t_user t ",
            " where t.user_id in",
            "<foreach collection='userIds' item='id' open='(' separator=',' close=')'>",
            "#{id}",
            "</foreach>",
            "</script>"
    })
    List<Map> selectUserbyIds(@Param("userIds")List<Long> userIds);

}
```

(4)测试

新增单元测试方法：

```
@Test
```

```
public void testInsertUser(){
    for (int i = 0 ; i<10; i++){
        Long id = i + 1L;
        userDao.insertUser(id,"姓名"+ id );
    }

}
@Test
public void testSelectUserbyIds(){
    List<Long> userIds = new ArrayList<>();
    userIds.add(1L);
    userIds.add(2L);
    List<Map> users = userDao.selectUserbyIds(userIds);
    System.out.println(users);
}
```

执行testInsertUser:

```
: Started OrderDaoTest in 1.795 seconds (JVM running for 2.3)
: ==>  Preparing: insert into t_user(user_id, fullname) value(?,?)
: ==> Parameters: 1(Long), 姓名1(String)
: Rule Type: sharding
: Logic SQL: insert into t_user(user_id, fullname) value(?,?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_user, alias=Optional.absent(
: Actual SQL: m0 ::: insert into t_user (user_id, fullname) VALUES (?, ?) ::: [1, 姓名1]
: <==    Updates: 1
```

通过日志可以看出t_user表的数据被落在了m0数据源，达到目标。

执行testSelectUserbyIds:

```
: Started OrderDaoTest in 1.665 seconds (JVM running for 2.135)
: ==>  Preparing: select * from t_user t where t.user_id in ( ? , ? )
: ==> Parameters: 1(Long), 2(Long)
: Rule Type: sharding
: Logic SQL: select  *  from t_user t  where t.user_id in  (  ?  ,  ?  )
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Ta
: Actual SQL: m0 ::: select  *  from t_user t  where t.user_id in  (  ?  ,  ?  ) ::: [1, 2]
: <==      Total: 2
```

通过日志可以看出t_user表的查询操作被落在了m0数据源，达到目标。

7.公共表

公共表属于系统中数据量较小，变动少，而且属于高频联合查询的依赖表。参数表、数据字典表等属于此类型。可以将这类表在每个数据库都保存一份，所有更新操作都同时发送到所有分库执行。接下来看一下如何使用Sharding-JDBC实现公共表。

(1)创建数据库

分别在user_db、order_db_1、order_db_2中创建t_dict表：

```
CREATE TABLE `t_dict`  (
  `dict_id` bigint(20) NOT NULL COMMENT '字典id',
  `type` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '字典类型',
  `code` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '字典编码',
  `value` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '字典值',
  PRIMARY KEY (`dict_id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

(2)在Sharding-JDBC规则中修改

```
# 指定t_dict为公共表
spring.shardingsphere.sharding.broadcast-tables=t_dict
```

(3)数据操作

新增DictDao：

```
@Mapper
@Component
public interface DictDao {

    /**
     * 新增字典
     * @param type 字典类型
     * @param code 字典编码
     * @param value 字典值
     * @return
     */
    @Insert("insert into t_dict(dict_id,type,code,value) value(#{dictId},#{type},#{code},#
{value})")
    int insertDict(@Param("dictId") Long dictId,@Param("type") String type, @Param("code")String
code, @Param("value")String value);

    /**
     * 删除字典
     * @param dictId 字典id
     * @return
     */
    @Delete("delete from t_dict where dict_id = #{dictId}")
    int deleteDict(@Param("dictId") Long dictId);

}
```

(4)字典操作测试

新增单元测试方法：

```
@Test
public void testInsertDict(){
        dictDao.insertDict(1L,"user_type","0","管理员");
        dictDao.insertDict(2L,"user_type","1","操作员");
}
@Test
public void testDeleteDict(){
        dictDao.deleteDict(1L);
        dictDao.deleteDict(2L);
}
```

执行testInsertDict：

```
: ==>  Preparing: insert into t_dict(dict_id,type,code,value) value(?,?,?,?)
: ==> Parameters: 1(Long), menu_type(String), 0(String), 父级菜单(String)
: Rule Type: sharding
: Logic SQL: insert into t_dict(dict_id,type,code,value) value(?,?,?,?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_d
: Actual SQL: m0 ::: insert into t_dict (dict_id, type, code, value) VALUES (?, ?, ?, ?) ::: [1, menu_type, 0, 父级菜单]
: Actual SQL: m1 ::: insert into t_dict (dict_id, type, code, value) VALUES (?, ?, ?, ?) ::: [1, menu_type, 0, 父级菜单]
: Actual SQL: m2 ::: insert into t_dict (dict_id, type, code, value) VALUES (?, ?, ?, ?) ::: [1, menu_type, 0, 父级菜单]
: <==    Updates: 1
```

通过日志可以看出，对t_dict的表的操作被广播至所有数据源。

测试删除字典，观察是否把所有数据源中该 公共表的记录删除。

（5）字典关联查询测试

字典表已在各各分库存在，各业务表即可和字典表关联查询。

定义用户关联查询dao：

在UserDao中定义：

```
/**
 * 根据id列表查询多个用户，关联查询字典表
 * @param userIds 用户id列表
 * @return
 */
@Select({"<script>",
        " select",
        " * ",
        " from t_user t ,t_dict b",
        " where t.user_type = b.code and t.user_id in",
        "<foreach collection='userIds' item='id' open='(' separator=',' close=')'>",
        "#{id}",
        "</foreach>",
        "</script>"
})
List<Map> selectUserInfobyIds(@Param("userIds")List<Long> userIds);
```

定义测试方法：

```
@Test
public void testSelectUserInfobyIds(){
    List<Long> userIds = new ArrayList<>();
    userIds.add(1L);
    userIds.add(2L);
    List<Map> users = userDao.selectUserInfobyIds(userIds);
    JSONArray jsonUsers = new JSONArray(users);
    System.out.println(jsonUsers);

}
```

执行测试方法，查看日志，成功关联查询字典表：

```
Logic SQL: select  *  from t_user t ,t_dict b  where t.user_type = b.code and t.user_id in  (  ?  ,  ?  )
SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t
Actual SQL: m0 ::: select  *  from t_user t ,t_dict b  where t.user_type = b.code and t.user_id in  (  ?  ,  ?  ) :::
<==      Total: 2
'操作员"},{"dict_id":2,"user_type":"1","code":"1","user_id":2,"fullname":"姓名2","type":"user_type","value":"操作员"}]
```

附 SQL支持说明

详细参考：https://shardingsphere.apache.org/document/current/cn/features/sharding/use-norms/sql/

说明：以下为官方显示内容，具体是否适用以实际测试为准 。

支持的SQL

SQL	必要条件
SELECT * FROM tbl_name	
SELECT * FROM tbl_name WHERE (col1 = ? or col2 = ?) and col3 = ?	
SELECT * FROM tbl_name WHERE col1 = ? ORDER BY col2 DESC LIMIT ?	
SELECT COUNT(*), SUM(col1), MIN(col1), MAX(col1), AVG(col1) FROM tbl_name WHERE col1 = ?	
SELECT COUNT(col1) FROM tbl_name WHERE col2 = ? GROUP BY col1 ORDER BY col3 DESC LIMIT ?, ?	
INSERT INTO tbl_name (col1, col2,...) VALUES (?, ?, ...)	
INSERT INTO tbl_name VALUES (?, ?,....)	
INSERT INTO tbl_name (col1, col2, ...) VALUES (?, ?, ...), (?, ?, ...)	
UPDATE tbl_name SET col1 = ? WHERE col2 = ?	
DELETE FROM tbl_name WHERE col1 = ?	
CREATE TABLE tbl_name (col1 int, ...)	
ALTER TABLE tbl_name ADD col1 varchar(10)	
DROP TABLE tbl_name	
TRUNCATE TABLE tbl_name	
CREATE INDEX idx_name ON tbl_name	
DROP INDEX idx_name ON tbl_name	
DROP INDEX idx_name	
SELECT DISTINCT * FROM tbl_name WHERE col1 = ?	
SELECT COUNT(DISTINCT col1) FROM tbl_name	

不支持的SQL

SQL	不支持原因
INSERT INTO tbl_name (col1, col2, ...) VALUES(1+2, ?, ...)	VALUES语句不支持运算表达式
INSERT INTO tbl_name (col1, col2, ...) SELECT col1, col2, ... FROM tbl_name WHERE col3 = ?	INSERT .. SELECT
SELECT COUNT(col1) as count_alias FROM tbl_name GROUP BY col1 HAVING count_alias > ?	HAVING
SELECT * FROM tbl_name1 UNION SELECT * FROM tbl_name2	UNION
SELECT * FROM tbl_name1 UNION ALL SELECT * FROM tbl_name2	UNION ALL
SELECT * FROM ds.tbl_name1	包含schema
SELECT SUM(DISTINCT col1), SUM(col1) FROM tbl_name	详见DISTINCT支持情况详细说明

DISTINCT支持情况详细说明

支持的SQL

SQL
SELECT DISTINCT * FROM tbl_name WHERE col1 = ?
SELECT DISTINCT col1 FROM tbl_name
SELECT DISTINCT col1, col2, col3 FROM tbl_name
SELECT DISTINCT col1 FROM tbl_name ORDER BY col1
SELECT DISTINCT col1 FROM tbl_name ORDER BY col2
SELECT DISTINCT(col1) FROM tbl_name
SELECT AVG(DISTINCT col1) FROM tbl_name
SELECT SUM(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1) FROM tbl_name GROUP BY col1
SELECT COUNT(DISTINCT col1 + col2) FROM tbl_name
SELECT COUNT(DISTINCT col1), SUM(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1), col1 FROM tbl_name GROUP BY col1
SELECT col1, COUNT(DISTINCT col1) FROM tbl_name GROUP BY col1

不支持的SQL

SQL	不支持原因
SELECT SUM(DISTINCT col1), SUM(col1) FROM tbl_name	同时使用普通聚合函数和DISTINCT聚合函数