# 软件架构与中间件



涂志莹
tzy_hit@hit.edu.cn

哈尔滨工业大学

# 软件架构与中间件

# Software Architecture and Middleware


## 第4章
## 数据层的软件架构技术

# 第4章 数据层的软件架构技术

4.1 数据驱动的软件架构演化

4.2 数据读写与主从分离

4.3 数据分库分表

4.4 数据缓存

# 4.3 数据分库分表

1、分库分表的基本概念

2、分库分表的解决方案

3、分库分表的架构设计

4、分库分表的中间件简介

# 4.3.1 分库分表的基本概念

- 什么是分库分表
- 分库分表的发展阶段
- 什么情况下需要分库分表
- 分库分表的典型实例

# 什么是分库分表

# 数据库和数据库实例

- 数据库是文件的集合，是依照某种数据模型组织起来并存放于二级存储器中的数据集合；
- 数据库实例是程序，是位于用户和操作系统之间的一层数据管理软件，用户对数据库中的数据做任何的操作，包括数据定义、数据查询、数据维护、数据库运行控制等等都是在数据库实例下进行的，应用程序只有通过数据库实例才能和数据库进行交互。

- 数据库是由文件组成（一般来说都是二进制文件）的，一般不可能直接对这些二进制文件进行操作，以实现数据库的SELECT、UPDATE、INSERT和DELETE操作，需要数据库实例来完成对数据库的操作。

# 分库分表的基本定义

- 分库分表的本质是数据拆分，是对数据进行分而治之的通用概念。
- 为了分散数据库的压力，采用分库分表将一个表结构分为多个表，或者将一个表的数据分片后放入多个表，这些表可以放在同一个库里，也可以放到不同的库里，甚至可以放在不同的数据库实例上。

- 数据拆分主要分为：垂直拆分和水平拆分
  - 垂直拆分：根据业务的维度，将原本的一个库（表）拆分为多个库（表），每个库（表）与原有的结构不同。
  - 水平拆分：根据分片（sharding）算法，将一个库（表）拆分为多个库（表），每个库（表）依旧保留原有的结构。

# 分库分表的发展阶段

# 分库分表的发展阶段

- 单库单表
  - 例如，将所有用户的信息都存放在同一数据库里的USER表中
- 单库多表
  - 单表内数据越来越多，查询性能下降，有锁表的可能，并阻塞所有其他的操作
  - 例如，将USER表中的数据水平切分，产生多个结构完全一样的表，如User0，User1……UserN，所有表的数据加起来为原有全量的数据
- 多库多表
  - 单台数据库的存储空间不够用，增加和减少索引消耗时间过长
  - 对数据库进行水平切分，将切分的数据库和表水平地分散到不同的数据库实例上

# 什么情况下需要分库分表

# 分库分表的操作时机

- 如果在数据库中表的数量达到了一定量级，则需要进行分表，分解单表的大数据量对索引查询带来的压力，并方便对索引和表结构的变更
- 如果数据库的吞吐量达到了瓶颈，就需要增加数据库实例，利用多个数据库实例来分解大量的数据库请求带来的系统压力
- 如果希望在扩容时对应用层的配置改变最少，就需要在每个数据库实例中预留足够的数据库数量

# 分库分表的典型实例

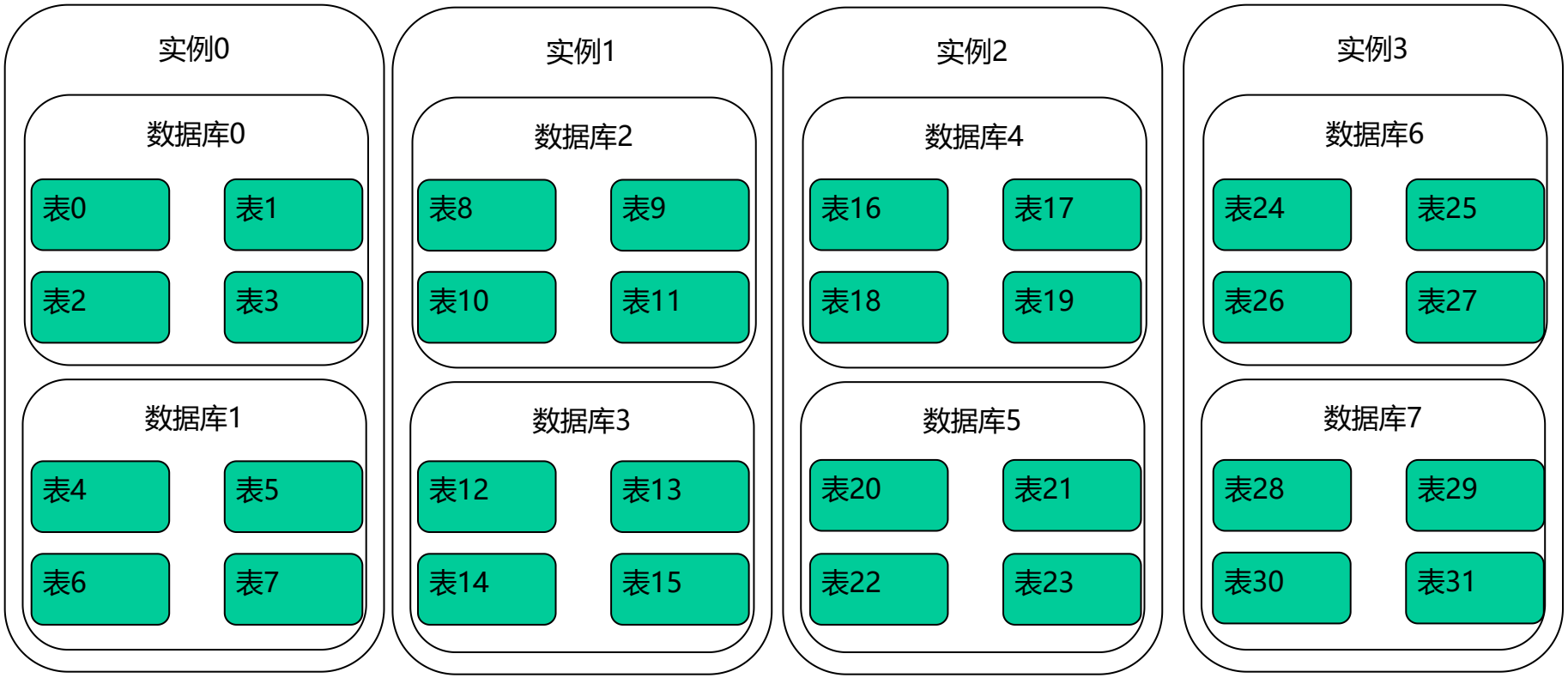
# 案例说明

- 某互联网应用当前拥有的16亿条用户消费数据记录需要存放在MySQL中
- 一般而言，MySQL单表存储5000万条数据记录就已经达到极限了

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 切片0 | 切片1 | 切片2 | 切片3 | 切片4 | 切片5 | 切片6 | 切片7 |
| 切片8 | 切片9 | 切片10 | 切片11 | 切片12 | 切片13 | 切片14 | 切片15 |
| 切片16 | 切片17 | 切片18 | 切片19 | 切片20 | 切片21 | 切片22 | 切片23 |
| 切片24 | 切片25 | 切片26 | 切片27 | 切片28 | 切片29 | 切片30 | 切片31 |

# 案例说明

- 如果将这些数据分解到4个数据库实例里，每个数据库实例包含2个数据库，每个数据库里有4个表。



# 4.3.2 分库分表的解决方案

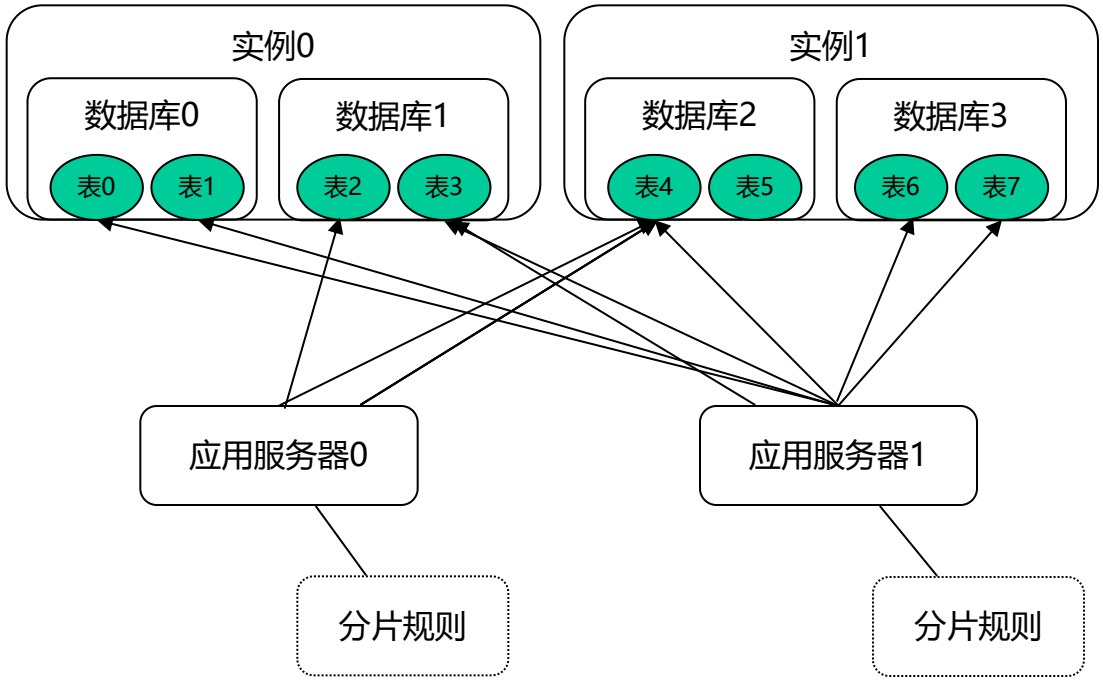
- 客户端分片
- 代理分片
- 支持事务的分布式数据库

# 客户端分片

# 基本概念

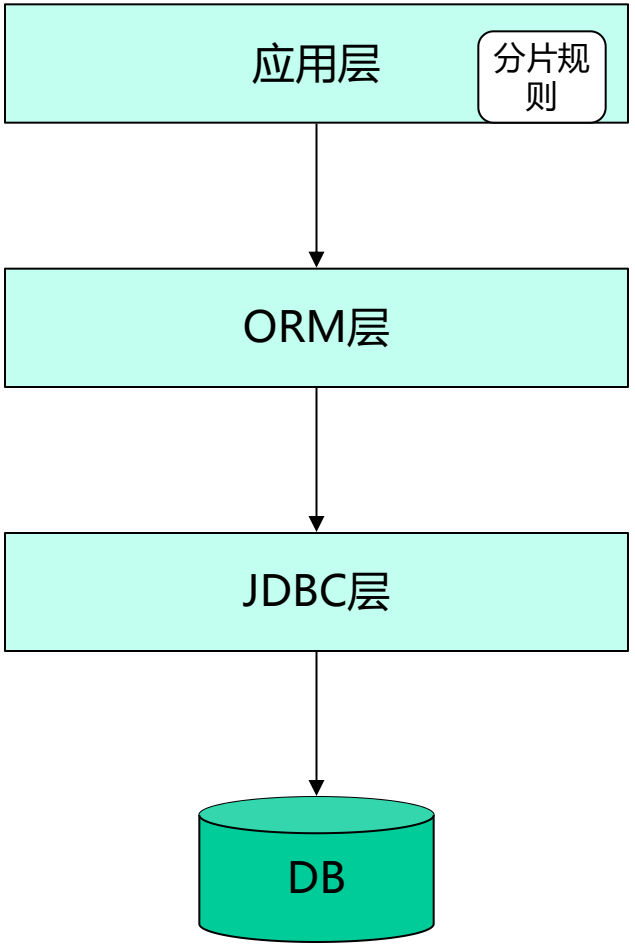
- 客户端分片就是使用分库分表的数据库的应用层直接操作分片逻辑，分片规则需要在同一个应用的多个节点间进行同步，每个应用层都嵌入一个操作切片的逻辑实现，一般通过依赖Jar包来实现。

- 具体实现方式分为三种：
  - 在应用层直接实现
  - 通过定制JDBC协议实现
  - 通过定制ORM框架实现



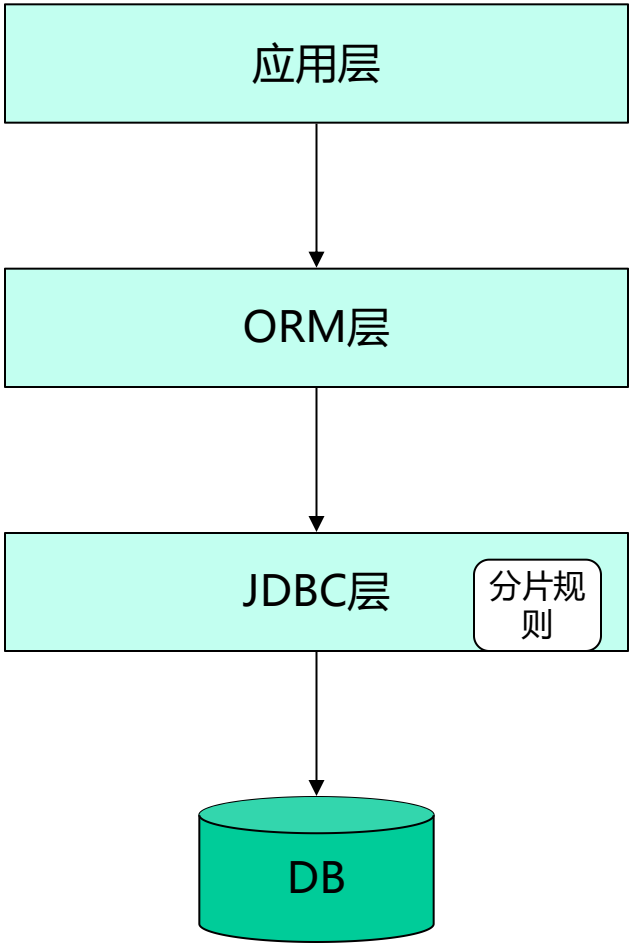
# 在应用层直接实现

- 直接在应用层读取分片规则，然后解析分片规则，据此实现切分的路由逻辑，从应用层直接决定每次操作应该使用哪个数据库实例、库、表等。
- 需要侵入业务，但实现简单，适合快速上线，切分逻辑由开发者自行定义，容易调试维护。但要求开发者既要实现业务逻辑，还需要实现框架需求。
- 该实现方式会让数据库保持的连接比较多，对整体应用服务器池的维护将造成压力。



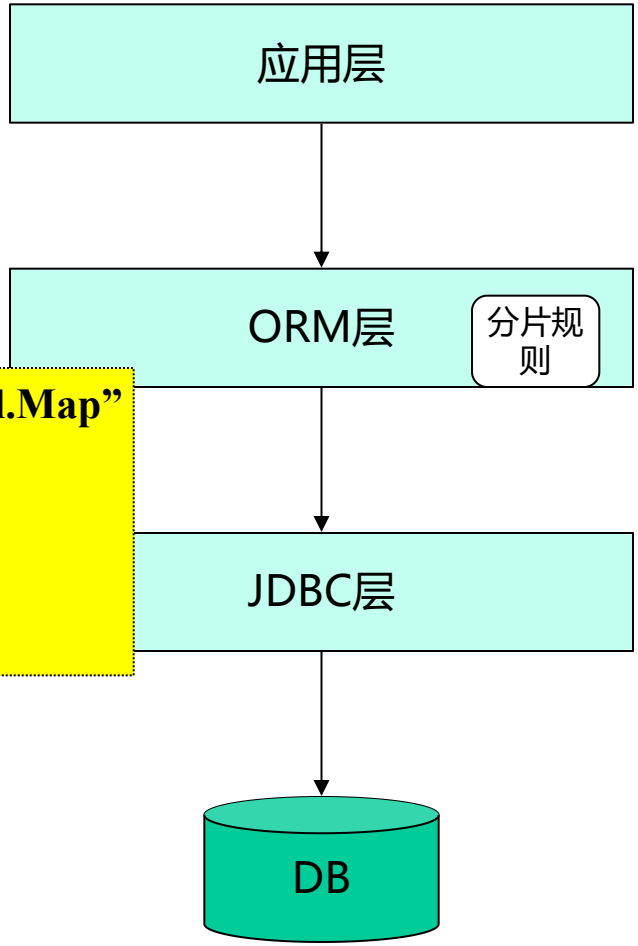
# 通过定制JDBC协议实现

- 可让开发者集中精力实现业务逻辑，无须关心分库分表的实现。
- 通过定制JDBC协议来实现，也就是针对业务逻辑层提供与JDBC一致的接口，分库分表在JDBC的内部实现。
- 开发者需要理解JDBC协议
- 流行的框架由Sharding JDBC



# 通过定制ORM框架实现

- 分片规则实现到ORM框架中或者通过ORM框架支持的扩展机制来完成分库分表的逻辑。
- 以Mybatis为例：




```
<select id="getEmpAsMapById" parameterType="java.util.Map"
            resultType="User">
    SELECT userId, username
    FROM t_employee
    WHERE userId = #{userId}
</select>
```

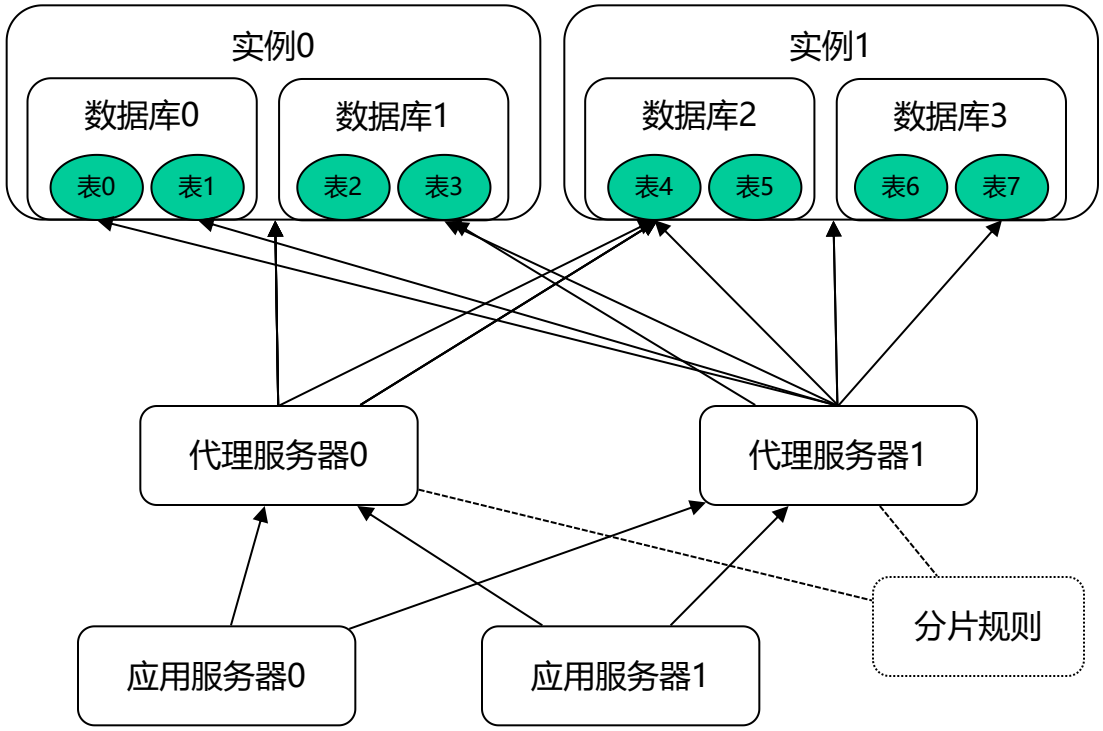
```
// 根据 id 查询信息，并把结果信息封装成 Map
Map<String, Object> getEmpAsMapById(Integer userId);
```

# 代理分片

# 代理分片

- 代理分片就是在应用层和数据库层之间增加一个代理层，把分片的路由规则配置在代理层，代理层对外提供与JDBC兼容的接口给应用层。
- 应用层的开发人员不用关心分片规则，只需关心业务逻辑的实现，待业务逻辑实现之后，在代理层配置路由规则即可。

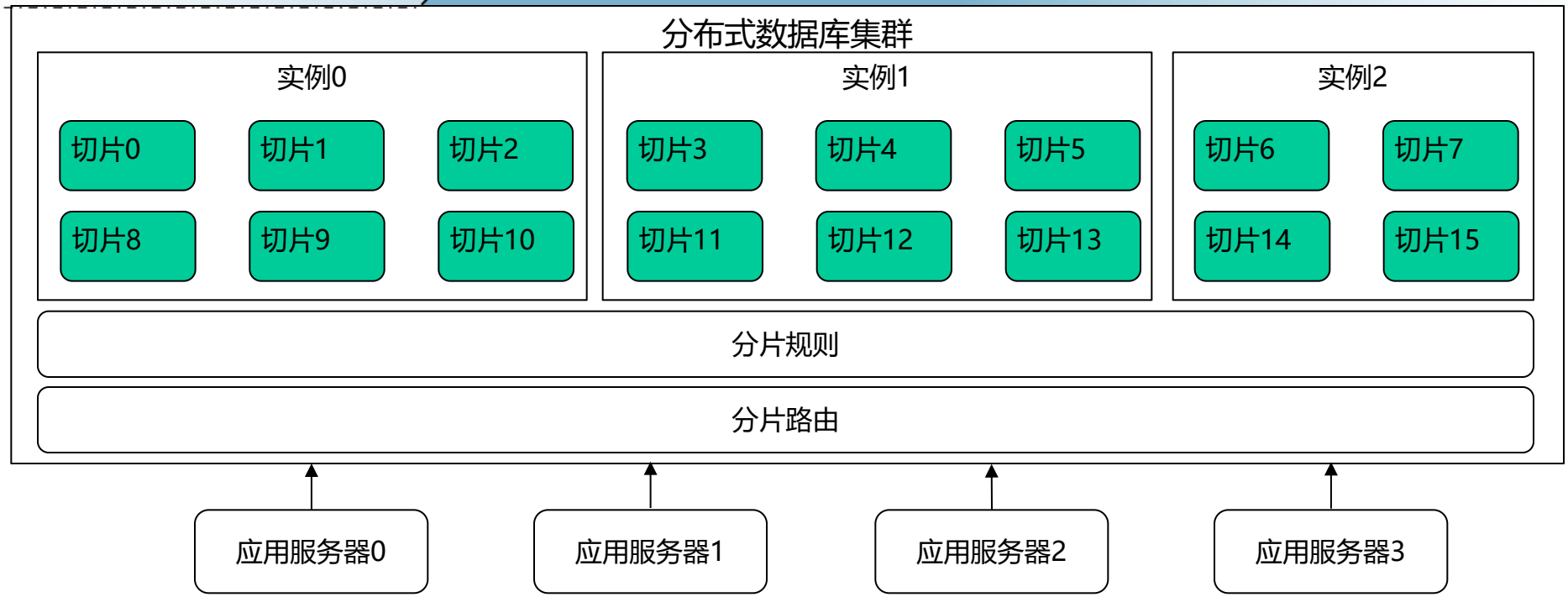
- 代理层的引入增加了一层网络传输，对性能会造成影响。
- 需要维护代理层，增加了人员和硬件的成本。
- 可用的框架：Cobar和Mycat等。



# 支持事务的分布式数据库

分库分表解决方案 → 支持事务DDB

# 支持事务的分布式数据库




- 现在有很多产品如OceanBase、TiDB等对外提供可伸缩的体系架构，并提供一定的分布式事务支持，将可伸缩的特点和分布式事务的实现包装到分布式数据库内部，对使用者透明，使用者不需要直接控制这些特性。
- TiDB对外提供JDBC的接口，让应用层像使用MySQL等传统数据库一样，无需关注伸缩、分片、事务管理等任务。
- 目前不太适用于交易系统，较多用于大数据日志系统、统计系统、查询系统、社交网络等。

# 4.3.3 分库分表的架构设计

- 切分方法
- 水平切分方式的路由过程和分片维度
- 分库分表引起的问题

# 切分方法

# 垂直切分

- 垂直切分是指按照业务将表进行分类或分拆，将其分布到不同数据库上
  - 按业务进行分库
  - 按业务进行分表
- 不同业务模块的数据可以分散到不同数据库服务器
  - 例如，User数据、Pay数据、Commodity数据
- 也可以冷热分离，根据数据的活跃度将数据进行拆分。
  - 冷数据：变化更新频率低，查询次数多的数据。
  - 热数据：变化更新频率高，活跃的数据。
- 也可以人为将一个表中的内容划分为多个表，例如将查询较多，变化不多的字段拆分成一张表放在查询性能高的服务器，而将频繁更新的字段拆分并部署到更新性能高的服务器。

# 垂直切分例子

- 在微博系统的设计中，一个微博对象包括文章标题、作者、分类、创建时间等属性字段，这些字段属于变化频率低的冷数据，而每篇微博的浏览数、回复数、点赞数等类似的统计信息属于变化频率高的热数据。
- 因此，一篇博客的数据可以按照冷热差异，拆分成两张表。冷数据存放的数据库可以使用MyISAM引擎，能更好地进行数据查询；热数据存放的数据库可以使用InnoDB存储引擎，更新性能好。
- 读多写少的冷数据库可以部署到缓存数据库上。

# 垂直切分的优缺点

- 优点：
  - 拆分后业务清晰，拆分规则明确
  - 系统之间进行整合或扩展很容易
  - 按照成本、应用的等级或类型等将表放到不同的机器上，便于管理
  - 便于实现动静分离、冷热分离的数据库表的设计模式
  - 数据维护简单
- 缺点：
  - 部分业务表无法关联（Join），只能通过接口方式解决，提高了系统的复杂度
  - 受每种业务的不同限制，存在单库性能瓶颈，不易进行数据扩展和提升性能
  - 事务处理复杂

# 水平切分

- 水平切分不是将表进行分类，而是将其按照某个字段的某种规则分散到多个库中，在每个表中包含一部分数据，所有表加起来是全量数据。
- 简言之，将数据按一定规律，按行切分，并分配到不同的库表里，表结构完全一样。
- 例如：在博客系统中，当同时有100万个用户在浏览时，如果是单表，则单表会进行100万次请求；假如将其分为100个表，并且分布在10个数据库中，每个表进行1万次请求，则每个数据库会承受10万次请求。当然还可以分配到不同服务器的服务实例中，分的表越多，每个单表的压力越小。

# 水平切分的优缺点

- 优点：
  - 单库单表的数据保持在一定的量级，有助于性能的提高
  - 切分的表的结构相同，应用层改造较少，只需要增加路由规则即可
  - 提高了系统的稳定性和负载能力
- 缺点：
  - 切分后，数据是分散的，很难利用数据库的Join操作，跨库Join性能差
  - 拆分规则难以抽象
  - 分片事务的一致性难以解决
  - 数据扩容的难度和维护量极大

# 垂直、水平切分的共同点

- 存在分布式事务的问题
- 存在跨节点Join的问题
- 存在跨节点合并排序、分页的问题
- 存在多数据源管理的问题

- 垂直切分更偏向于业务拆分的过程
- 水平切分更偏向于技术性能指标

# 水平切分方式的路由过程和分片维度

# 水平切分的路由过程

- 分库分表后，数据将分布到不同的分片表中，通过分库分表规则查找到对应的表和库的过程叫做路由。
- 我们在设计表时需要确定对表按照什么样的规则进行分库分表。例如，当生成新用户时，程序得确定将此用户的信息添加到哪个表中。
- 同样，在登录时我们需要通过用户的账号找到数据库中对应的记录。

# 水平切分的分片维度

- 按哈希切片
  - 对数据的某个字段求哈希，再除以分片总数后取模，取模后相同的数据为一个分片，这样将数据分成多个分片
  - 好处：数据切片比较均匀，对数据压力分散的效果较好
  - 缺点：数据分散后，对于查询需求需要进行聚合处理
- 按照时间切片
  - 按照时间的范围将数据分布到不同的分片

# 分库分表引起的问题

# 扩容与迁移

- 通用的处理方法：
  - 1、按照新旧分片规则，对新旧数据库进行双写
  - 2、将双写前按照旧分片规则写入的历史数据，根据新分片规则迁移写入新的数据库
  - 3、将按照旧的分片规则查询改为按照新的分片规则查询
  - 4、将双写数据库逻辑从代码中下线，只按照新的分片规则写入数据
  - 5、删除按照旧分片规则写入的历史数据

# 扩容与迁移的问题

- 数据一致性问题：
  - 由于数据量大，通常会造成不一致问题，因此，通常先清理旧数据，洗完后再迁移到新规则的新数据库下，再做全量对比。还需要对比评估迁移过程中是否有数据更新，如果有需要迭代清洗，直至一致。
  - 如果数据量巨大，无法全量对比，需要抽样对比，抽样特征需要根据业务特点进行选取。
  - 注意：线上记录迁移过程中的更新操作日志，迁移后根据更新日志与历史数据共同决定数据的最新状态，以达到迁移数据的最终一致性。
- 动静数据分离问题：
  - 对于一些动静敏感的数据，如交易数据，最好将动静数据分离。选取时间点对静历史数据进行迁移。

# 查询问题

- 查询问题解释：
  - 在分库分表以后，如果查询的标准是分片的主键，则可以通过分片规则再次路由并查询，但是对于其他主键的查询、范围查询、关联查询、查询结果排序等，并不是按照分库分表维度来查询的。
- 查询问题的解决方案：
  - 1、在多个分片表查询后合并数据集（效率很低）
  - 2、按查询需求定义多分片维度，形成多张分片表（空间换时间）
  - 3、通过搜索引擎解决，如果有实时要求，还需要实时搜索。（难度大）

# 分布式事务问题

- 多库多表分布式所引发的一致性问题。

# 同组数据跨库问题

- 要尽量把同一组数据放到同一台数据库服务器上，不但在某些场景下可以利用本地事务的强一致性，还可以使这组数据实现自治。

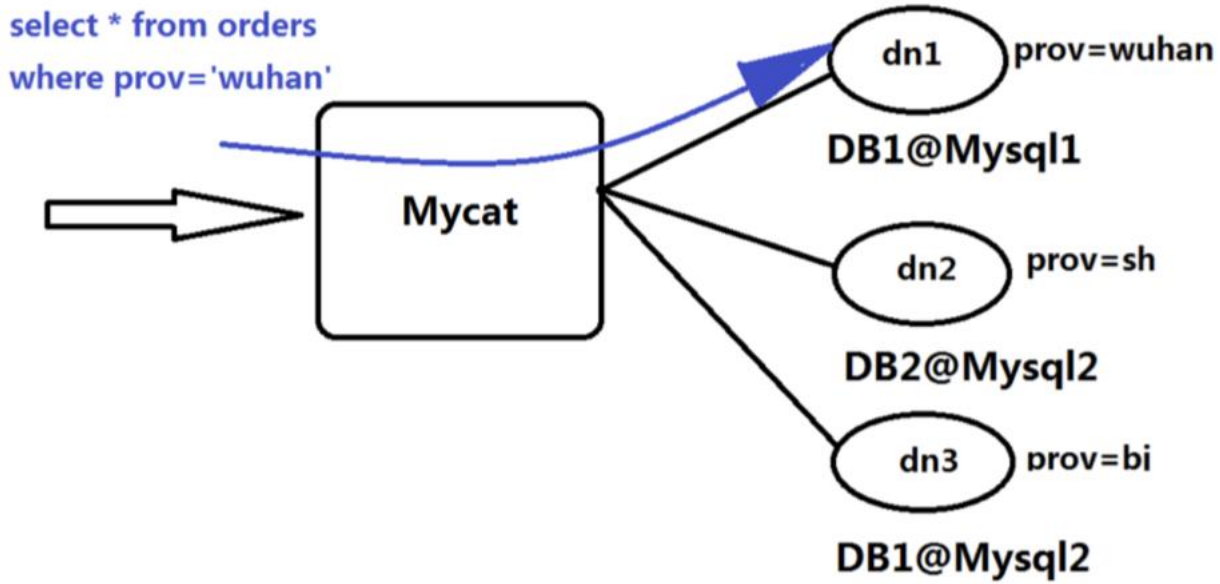
# 4.3.4 分库分表的中间件简介

- Mycat
- Sharding JDBC

# Mycat

- Mycat是一个强大的数据库中间件，不仅仅可以用作读写分离、以及分表分库、容灾备份，而且可以用于多租户应用开发、云平台基础设施。
- Mycat后面连接的Mycat Server，就好象是MySQL的存储引擎，如InnoDB，MyISAM 等，因此，Mycat 本身并不存储数据，数据是在后端的MySQL上存储的，因此数据可靠性以及事务等都是MySQL保证的。

# Mycat基本原理




- Mycat拦截了用户发送过来的SQL语句，首先对SQL语句做一些特定的分析：如分片分析、路由分析、读写分离分析、缓存分析等，然后将此 SQL 发往后端的真实数据库， 并将返回的结果做适当的处理，最终再返回给用户。
- 当 Mycat收到一个SQL时，会先解析这个SQL，查找涉及到的表，然后看此表的定义，如果有分片规则， 则获取到SQL里分片字段的值，并匹配分片函数，得到该SQL对应的分片列表，然后将SQL发往这些分片去执行，最后收集和处理所有分片返回的结果数据，并输出到客户端。

# Mycat应用场景

- 单纯的读写分离，此时配置最为简单，支持读写分离，主从切换
- 分表分库，对于超过1000万的表进行分片，最大支持1000亿的单表分片
- 多租户应用，每个应用一个库，但应用程序只连接Mycat，从而不改造程序本身，实现多租户化
- 报表系统，借助于Mycat 的分表能力，处理大规模报表的统计
- 替代Hbase，分析大数据
- 作为海量数据实时查询的一种简单有效方案，比如100亿条频繁查询的记录需要在 3 秒内查询出来结果，除了基于主键的查询，还可能存在范围查询或其他属性查询，此时 Mycat 可能是最简单有效的选择

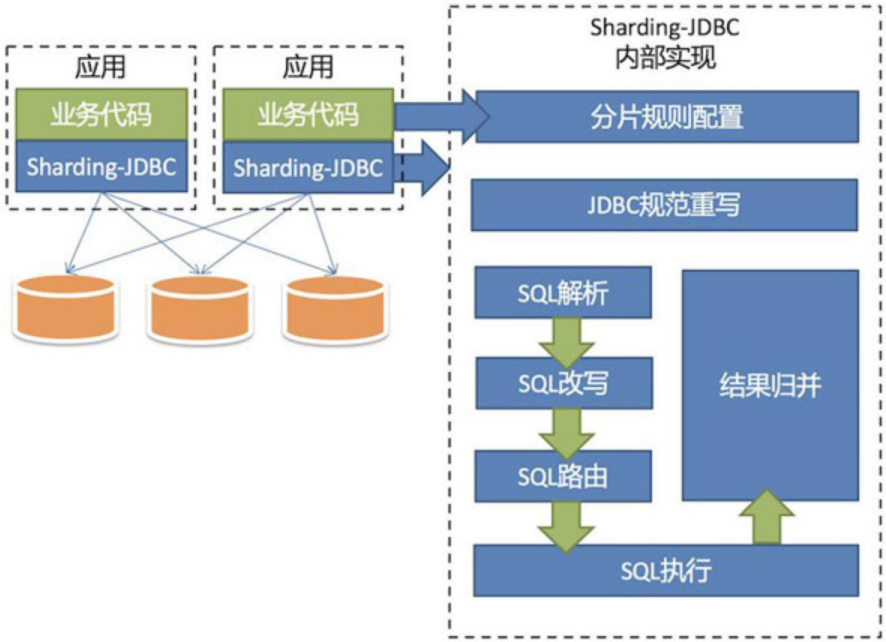
# Sharding JDBC

- Sharding-JDBC是当当应用框架ddframe中，从关系型数据库模块dd-rdb中分离出来的数据库水平分片框架，实现透明化数据库分库分表访问。Sharding-JDBC直接封装JDBC API，可以理解为增强版的JDBC驱动，旧代码迁移成本几乎为零：
  - 可适用于任何基于Java的ORM框架，如JPA、Hibernate、Mybatis、Spring JDBC Template或直接使用JDBC。
  - 可基于任何第三方的数据库连接池，如DBCP、C3P0、BoneCP、Druid等。
  - 理论上可支持任意实现JDBC规范的数据库。虽然目前仅支持MySQL，但已有支持Oracle、SQLServer等数据库的计划。
- Sharding-JDBC定位为轻量Java框架，使用客户端直连数据库，以jar包形式提供服务，无proxy代理层，无需额外部署，无其他依赖，DBA也无需改变原有的运维方式。
- Sharding-JDBC分片策略灵活，可支持等号、between、in等多维度分片，也可支持多分片键。
- SQL解析功能完善，支持聚合、分组、排序、limit、or等查询，并支持Binding Table以及笛卡尔积表查询。

# Sharding JDBC原理

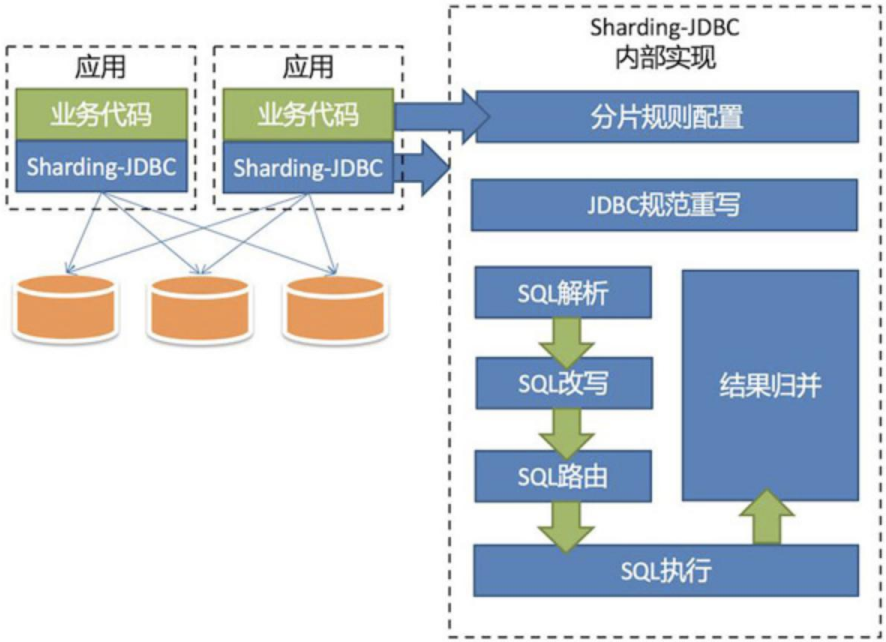
- 分片规则配置
  - Sharding-JDBC的分片逻辑非常灵活，支持分片策略自定义、复数分片键、多运算符分片等功能。
    - 如：根据用户ID分库，根据订单ID分表这种分库分表结合的分片策略；或根据年分库，月份+用户区域ID分表这样的多片键分片。

  - Sharding-JDBC除了支持等号运算符进行分片，还支持in/between运算符分片，提供了更加强大的分片功能。
  - Sharding-JDBC提供了spring命名空间用于简化配置，以及规则引擎用于简化策略编写。由于目前刚开源分片核心逻辑，这两个模块暂未开源，待核心稳定后将会开源其他模块。



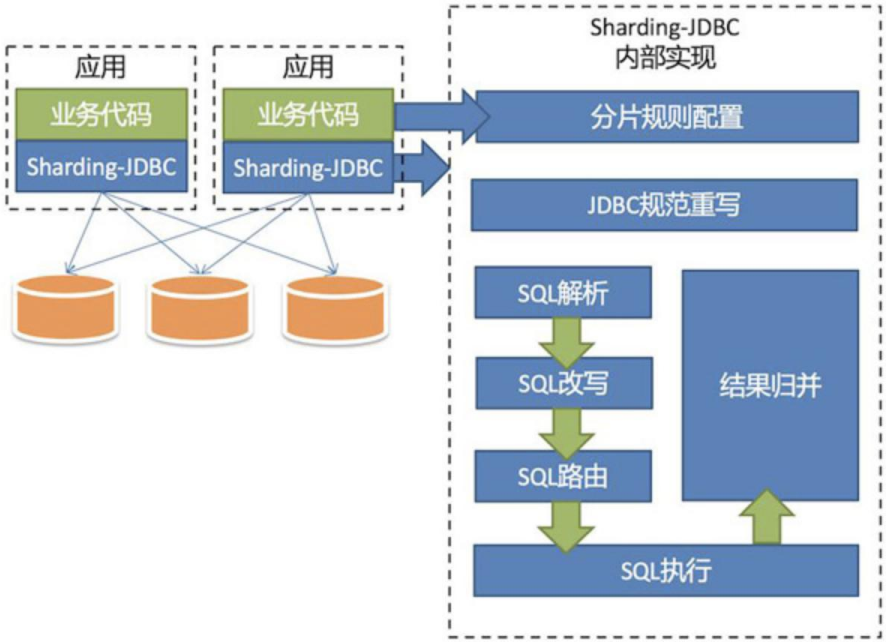
# Sharding JDBC原理

- JDBC规范重写
  - Sharding-JDBC对JDBC规范的重写思路是针对DataSource、Connection、Statement、PreparedStatement和ResultSet五个核心接口封装，将多个真实JDBC实现类集合（如：MySQL JDBC实现/DBCP JDBC实现等）纳入Sharding-JDBC实现类管理。
  - Sharding-JDBC尽量最大化实现JDBC协议，包括addBatch这种在JPA中会使用的批量更新功能。
  - 但分片JDBC毕竟与原生JDBC不同，所以目前仍有未实现的接口，包括Connection游标，存储过程和savePoint相关、ResultSet向前遍历和修改等不太常用的功能。此外，为了保证兼容性，并未实现JDBC 4.1及其后发布的接口（如：DBCP 1.x版本不支持JDBC 4.1）。



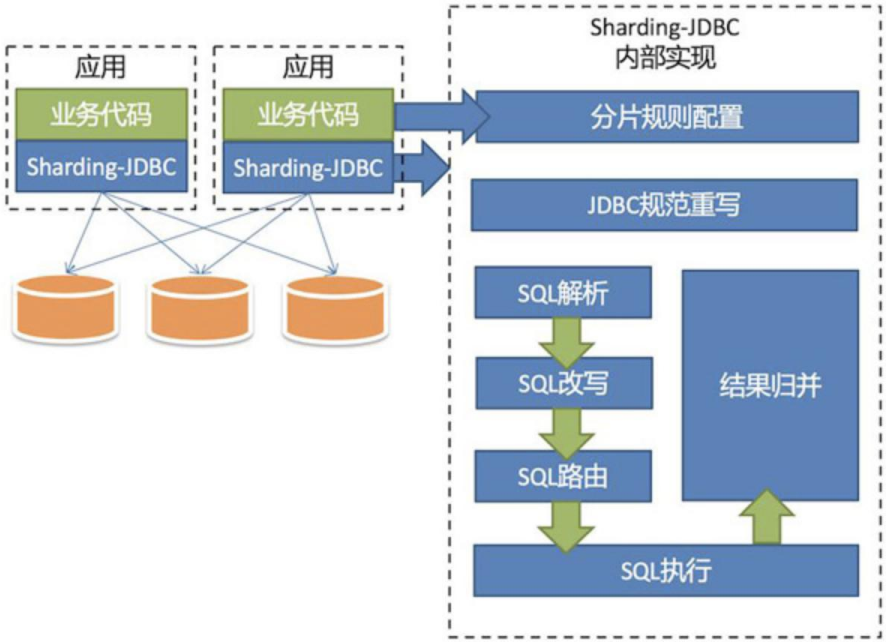
# Sharding JDBC原理

- SQL解析
  - SQL解析作为分库分表类产品的核心，性能和兼容性是最重要的衡量指标。目前常见的SQL解析器主要有fdb/jsqlparser和Druid。Sharding-JDBC使用Druid作为SQL解析器，经实际测试，Druid解析速度是另外两个解析器的几十倍。
  - 目前Sharding-JDBC支持join、aggregation（包括avg）、order by、 group by、limit、甚至or查询等复杂SQL的解析。目前不支持union、部分子查询、函数内分片等不太应在分片场景中出现的SQL解析。



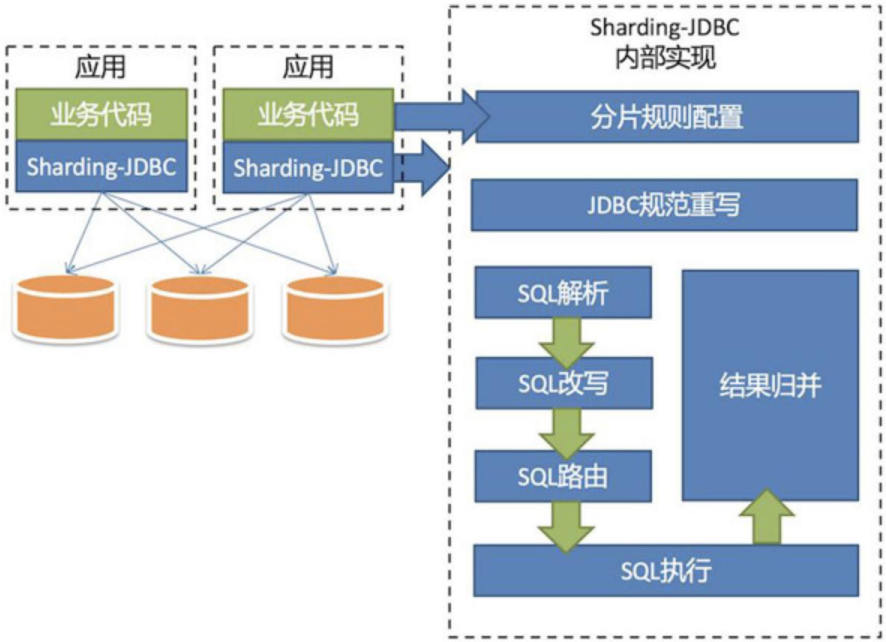
# Sharding JDBC原理

- SQL改写
  - SQL改写分为两部分，一部分是将分表的逻辑表名称替换为真实表名称。另一部分是根据SQL解析结果替换一些在分片环境中不正确的功能。这里具两个例子：
  - 如avg计算。在分片的环境中，以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，然后再结果归并时重新计算平均值。



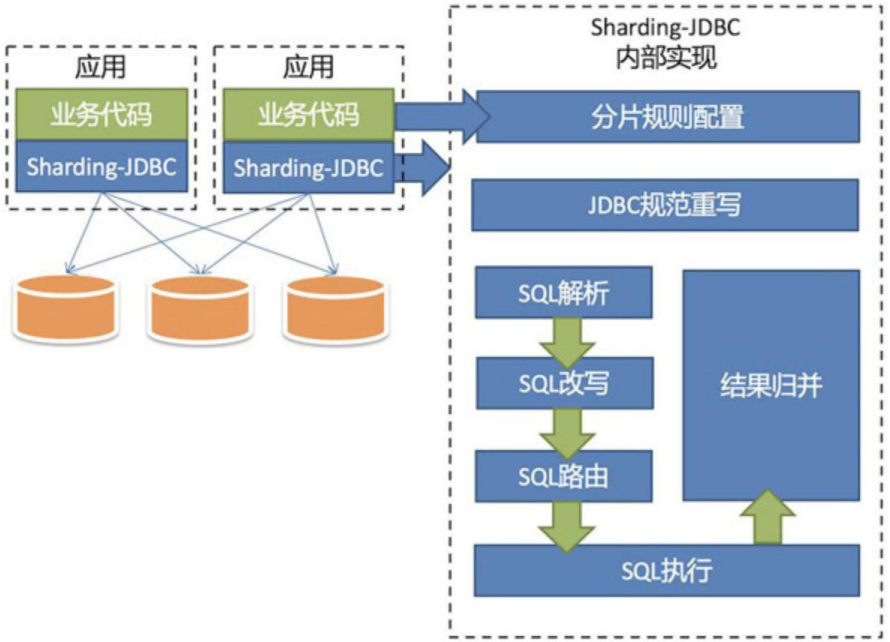
# Sharding JDBC原理

- SQL路由
  - SQL路由是根据分片规则配置，将SQL定位至真正的数据源。主要分为单表路由、Binding表路由和笛卡尔积路由。
  - 单表路由最为简单，但路由结果不一定落入唯一库（表），因为支持根据between和in这样的操作符进行分片，所以最终结果仍然可能落入多个库（表）。
  - Binding表可理解为分库分表规则完全一致的主从表。
  - 笛卡尔积查询最为复杂，因为无法根据Binding关系定位分片规则的一致性，所以非Binding表的关联查询需要拆解为笛卡尔积组合执行。查询性能较低，而且数据库连接数较高，需谨慎使用。



# Sharding JDBC原理

- SQL执行
  - 路由至真实数据源后，Sharding-JDBC将采用多线程并发执行SQL，并完成对addBatch等批量方法的处理。



# Sharding JDBC原理

- 结果归并
  - 结果归并包括4类：普通遍历类、排序类、聚合类和分组类。每种类型都会先根据分页结果跳过不需要的数据。
  - 普通遍历类最为简单，只需按顺序遍历ResultSet的集合即可。
  - 排序类结果将结果先排序再输出，因为各分片结果均按照各自条件完成排序，所以采用归并排序算法整合最终结果。
  - 聚合类分为3种类型，比较型、累加型和平均值型。比较型包括max和min，只返回最大（小）结果。累加型包括sum和count，需要将结果累加后返回。平均值则是通过SQL改写的sum和count计算，相关内容已在SQL改写涵盖，不再赘述。
  - 分组类最为复杂，需要将所有的ResultSet结果放入内存，使用map-reduce算法分组，最后根据排序和聚合条件做相关处理。最消耗内存，最损失性能的部分即是此，可以考虑使用limit合理的限制分组数据大小。
- 结果归并部分目前并未采用管道解析的方式，之后会针对这里做更多改进。

# 第4章

# 数据层的软件架构技术

# Thanks for listening

涂志莹
哈尔滨工业大学计算机学院
企业与服务计算研究中心