Go进阶训练营-第11课
DNS & CDN & 多活架构答疑

邓明

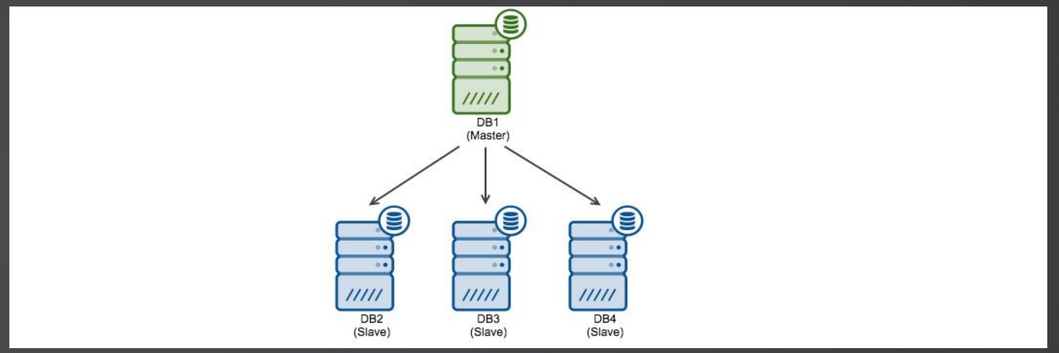
目录

目录

分库分表基本概念——为什么

为什么要分库分表？

- 并发高
- 数据量过大



可选择的技术：

- 主从分离：主从分离既不能解决写，也不能解决读。对于写来说，只能走写库，所以撑不住高并发；对于读来说，虽然可以走读库，但是如果单表数据过多，查询时间依旧很长。更加可怕的是，可能连索引都无法装进内存
- 分区：类似于主从分离，但是比主从分离更弱

单一数据库实例的处理能力是有限的！！

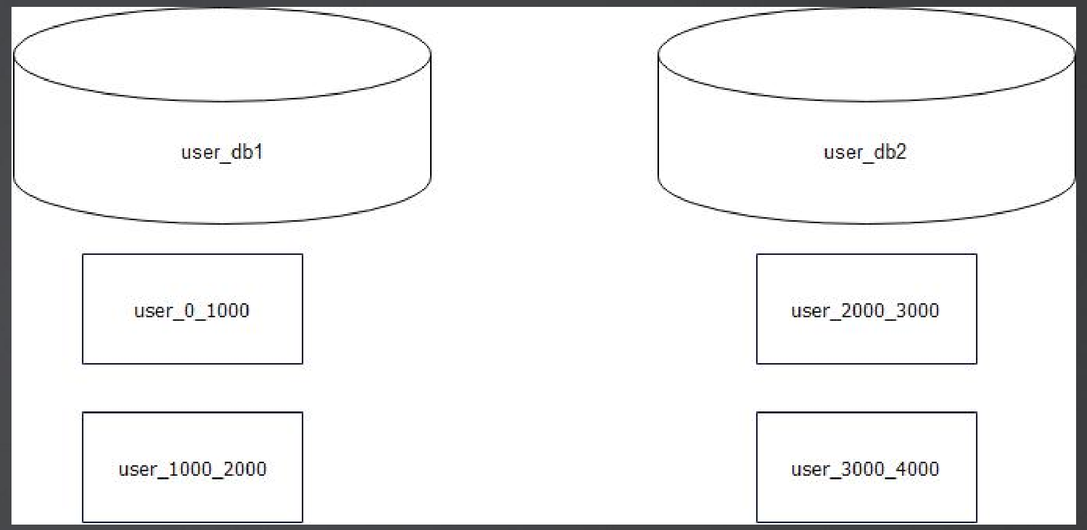
分库分表基本概念——分库和分表

分库分表说的是两件事：

- 分库：将数据分布在不同的数据库里面
- 分表：将数据分布在不同的数据表里面

可以只分库，也可以只分表，也可以都一起分。

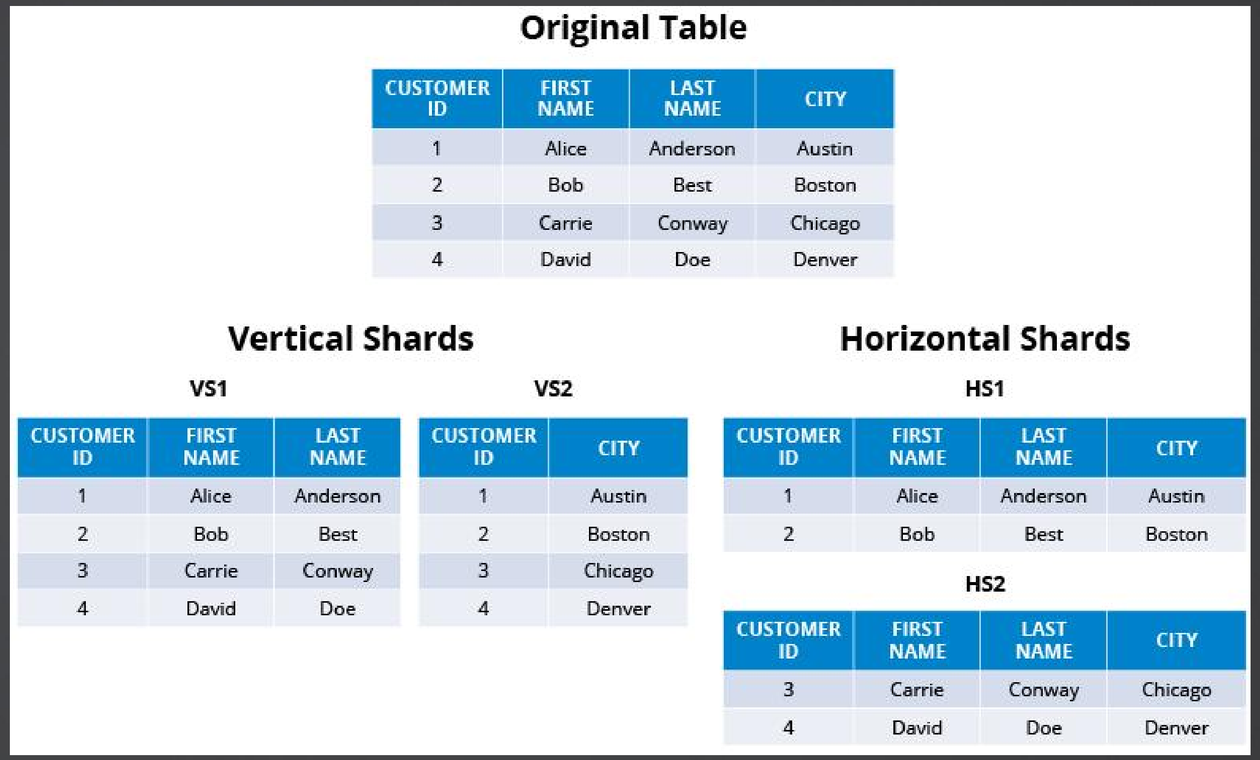
注意：单纯的分表，还是会面临单一数据库节点写瓶颈的问题。



分库分表基本概念——拆分方式

分库分表的两种拆分方式：

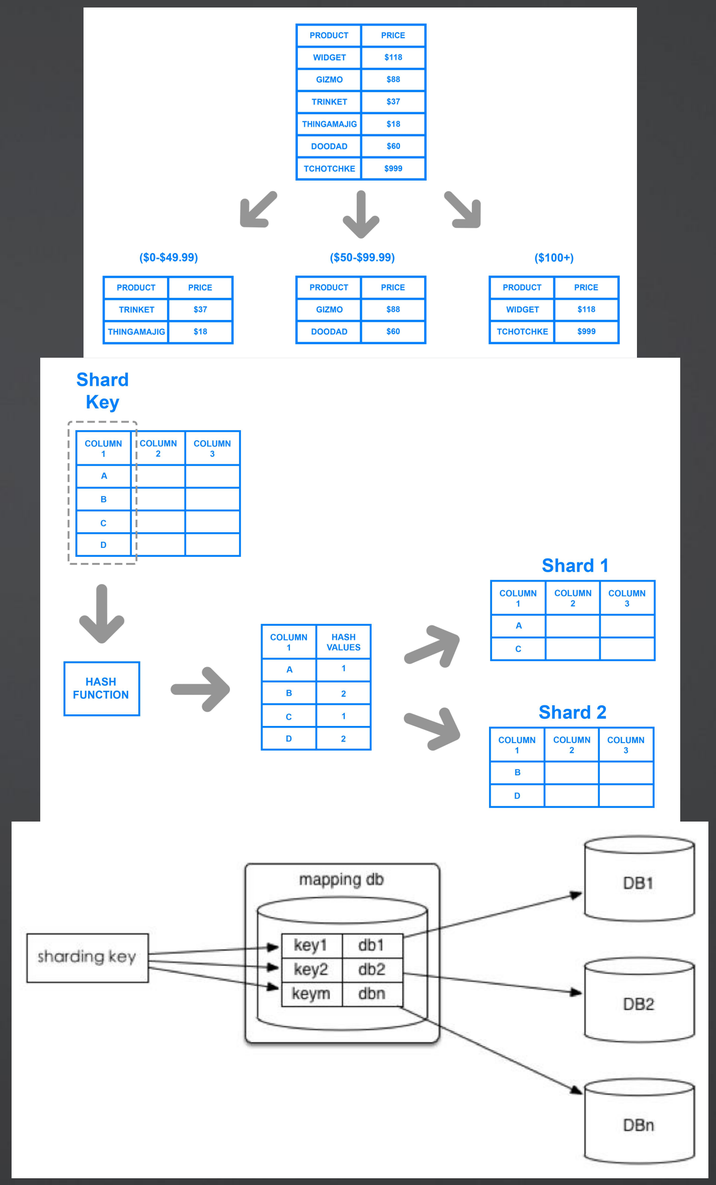
- 垂直拆分：一般是指按列分，比如说常用的列放在一起，不常用的列放在一起。典型的例子是订单基本信息表，订单详情表，订单扩展表；或者用户基本信息表，用户扩展信息表；
- 水平拆分：按行分，比如说按照范围来分，(0, 10000), (10000, 20000)



分库分表——水平拆分的常见方式

垂直拆分比较少讨论，都是业务相关，难免要改代码，水平拆分讨论比较多。水平拆分细分又可以分成：

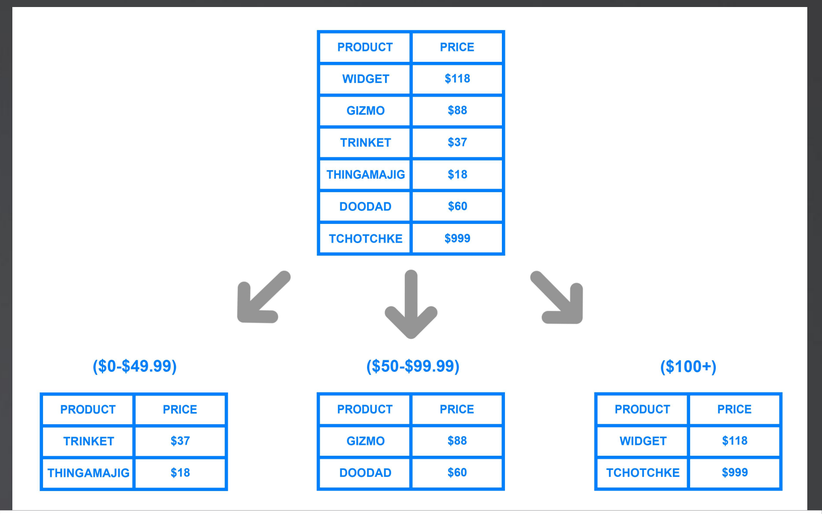
- 范围拆分：比如说按照主键范围拆分，时间范围拆分。这种拆分方式扩容比较容易
- 取余拆分：一般数据库和数据表按照 shard key 拆分，也叫做哈希拆分
- 映射关系拆分：单独建立一个主键到数据库或者数据表的映射。这个常结合前面两种一起用。比如说典型的可以按照用户 ID 来拆分，而后建立一个 email 到数据库数据表的中间表

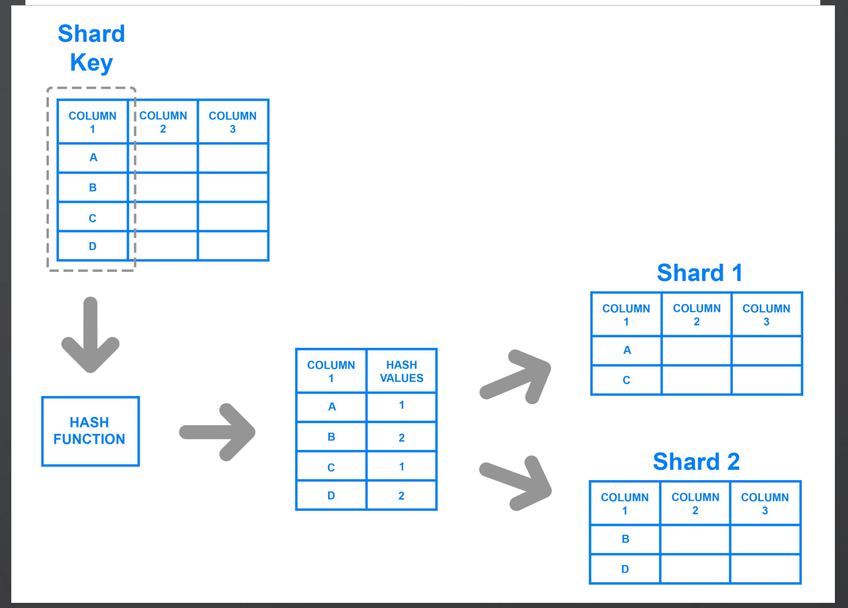


分库分表基本概念——拆分维度

从什么维度来拆分，可选择的并不多，常见的有：

- 业务 ID：常见的就是用户 ID 或者订单 ID；
- 日期：一般用在范围拆分上
- 地区：在国际业务里面比较常见

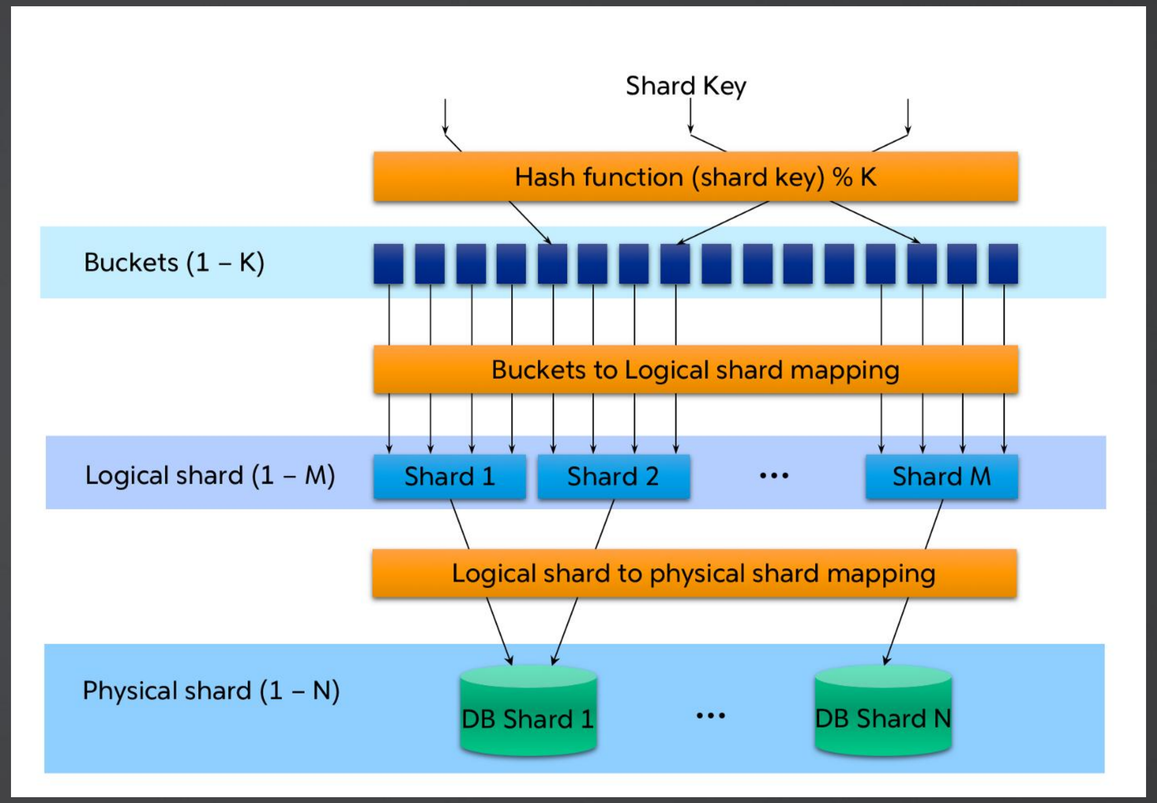




分库分表基本概念——物理表

分库分表不一定直接对应到物理表，可能在物理表中间有一个逻辑表，由另外一个组件来完成逻辑表到物理表的映射。

这个技术就和 Redis 的槽，一致性哈希虚拟节点一样，都是遇事不决加中间层的典型。



目录

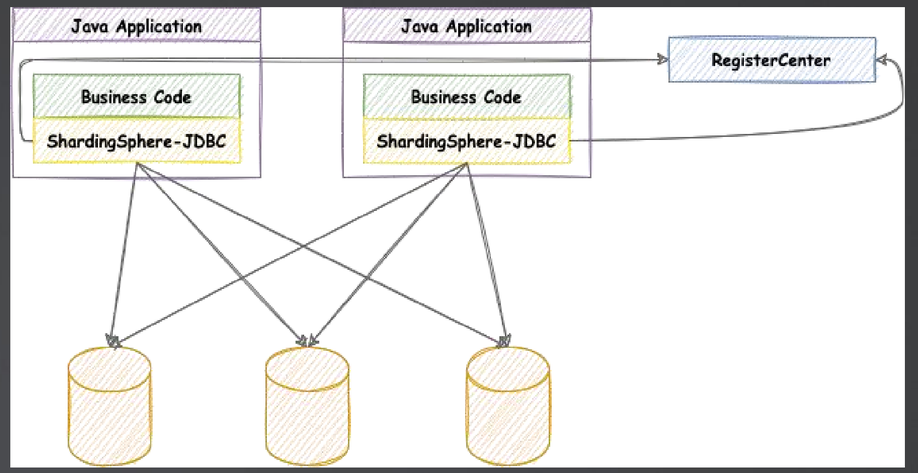
分库分表中间件

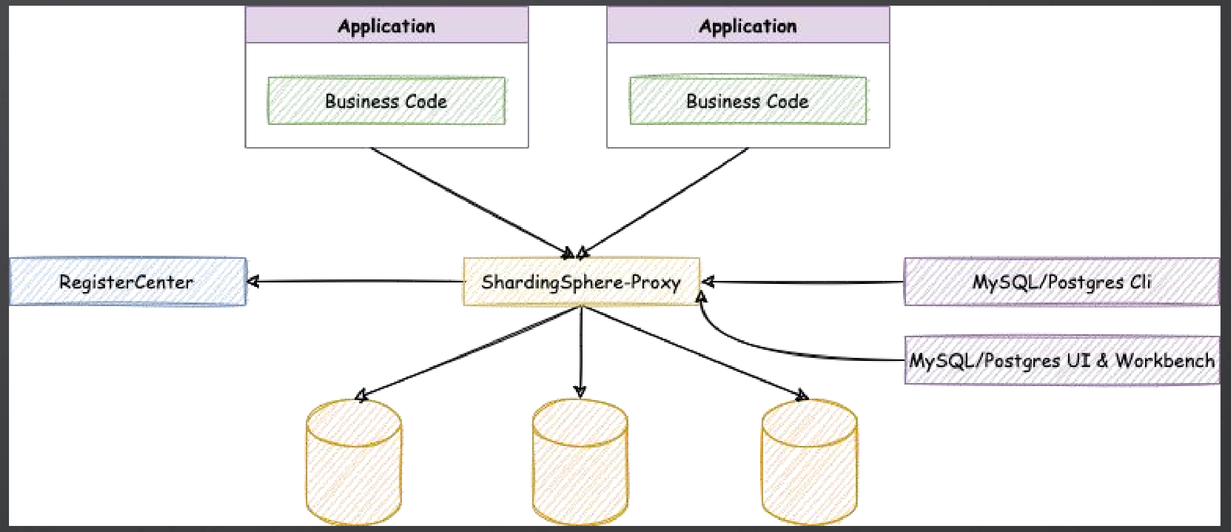
分库分表中间件大体上可以说三种：

1. SDK：也就是和语言强相关

2. Proxy：所有的数据库查询请求发到一个中间代理，代理来处理分库分表

3. mesh：云原生时代出现的，目前没有成熟的框架





分库分表中间件

分库分表中间件大体上可以说三种：

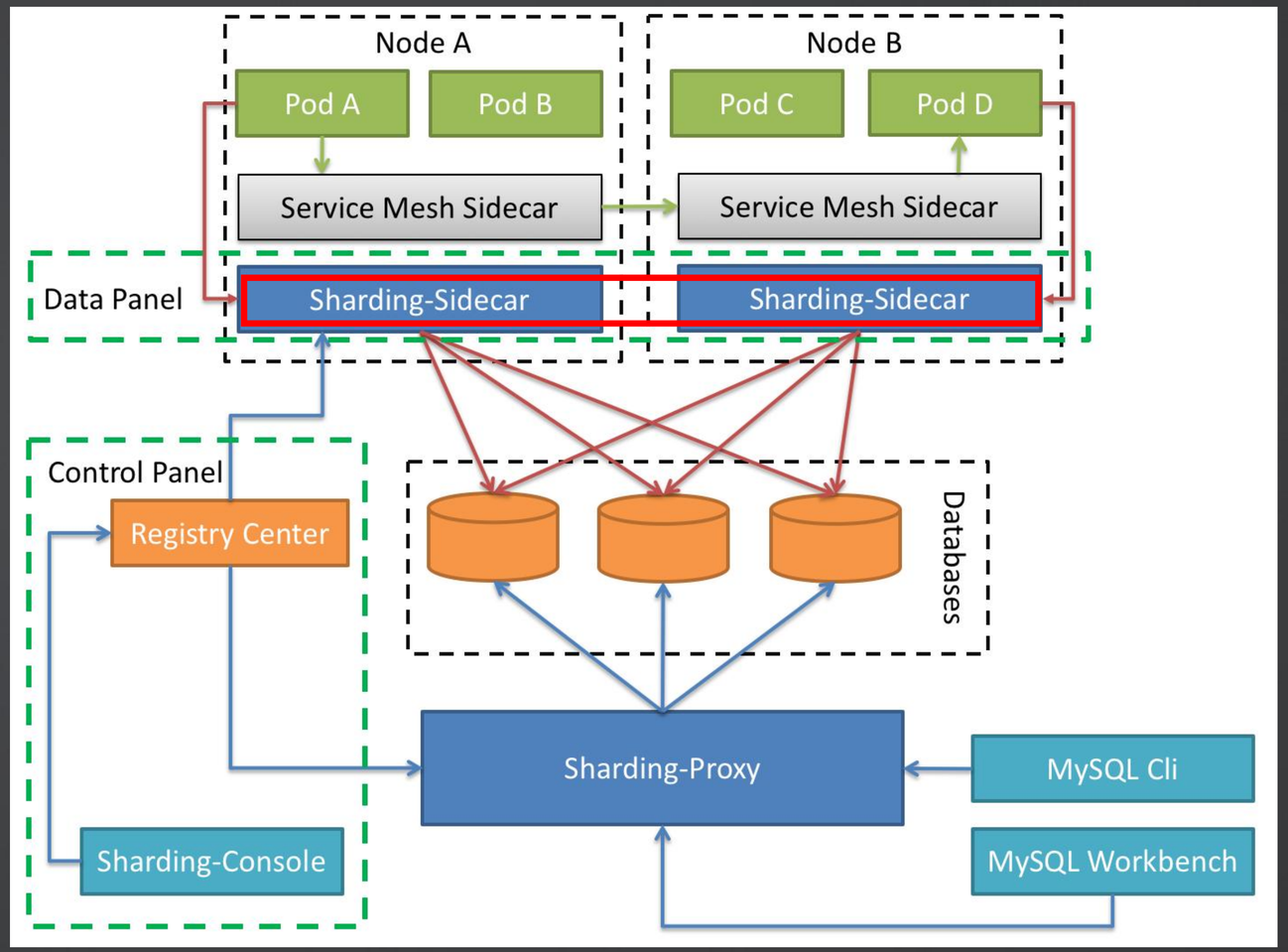
1. SDK：也就是和语言强相关

2. Proxy：所有的数据库查询请求发到一个中间代理，代理来处理分库分表

3. mesh：云原生时代出现的，目前没有成熟的框架

目前我个人评价 shardingsphere 是最为成熟的分库分表中间件。它这几年间影响了很多公司的分库分表的中间件设计。

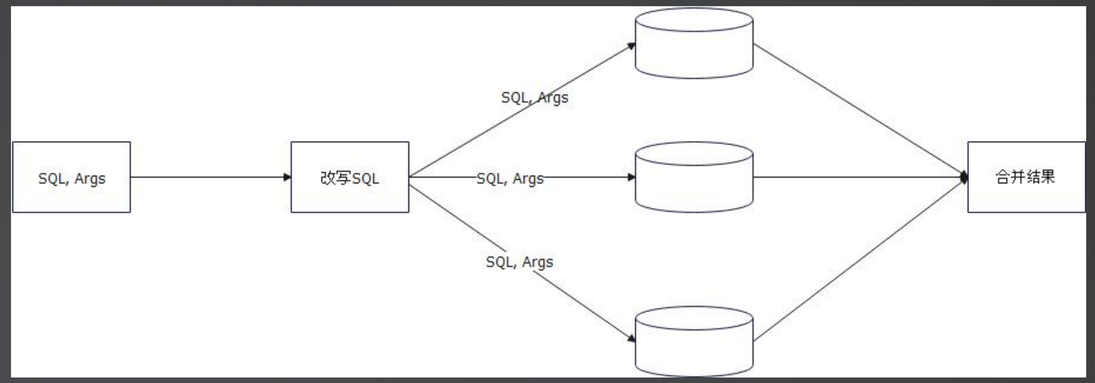
遇事不决选 shardingsphere



分库分表中间件——设计要点

分库分表中间件的核心步骤：

1. 改写 SQL
2. 执行 SQL
3. 合并结果

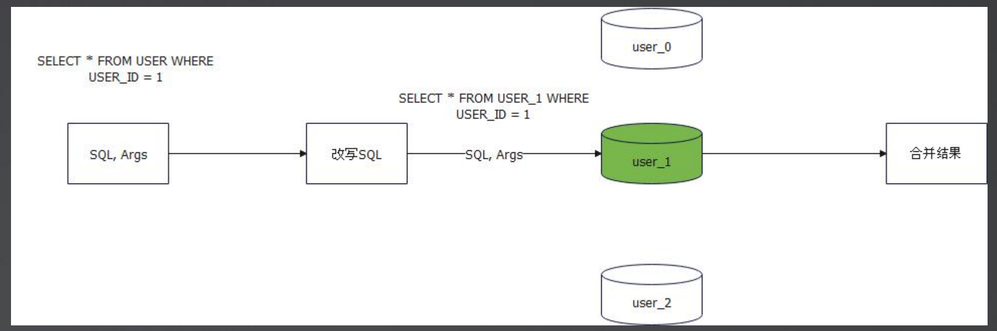


后面沿着这种思路去，大概是能够做出来一个玩具的分库分表的东西来

分库分表中间件——改写 SQL

改写 SQL 会有几种情况：

1. 只需要改个表名，数据库名。一般不需要全局状态的查询就是这样，例如单一数据查询。

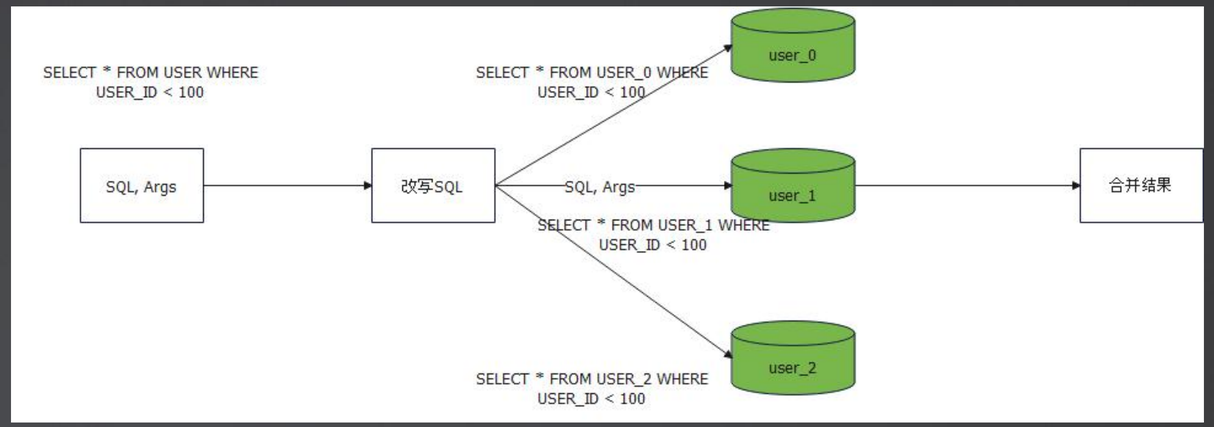


分库分表中间件——改写 SQL

改写 SQL 会有几种情况：

1. 只需要改个表名，数据库名。一般不需要全局状态的查询就是这样，例如单一数据查询。

2. 重写为多个 SQL：范围查询，或者需要全局状态的查询。

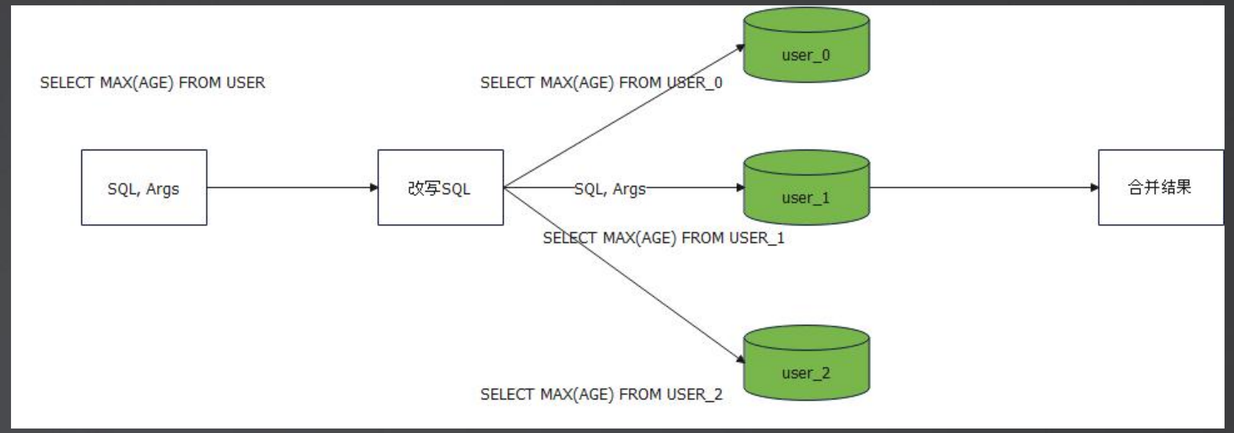


分库分表中间件——改写 SQL

改写 SQL 会有几种情况：

1. 只需要改个表名，数据库名。一般不需要全局状态的查询就是这样，例如单一数据查询。

2. 重写为多个 SQL：范围查询，或者需要全局状态的查询。



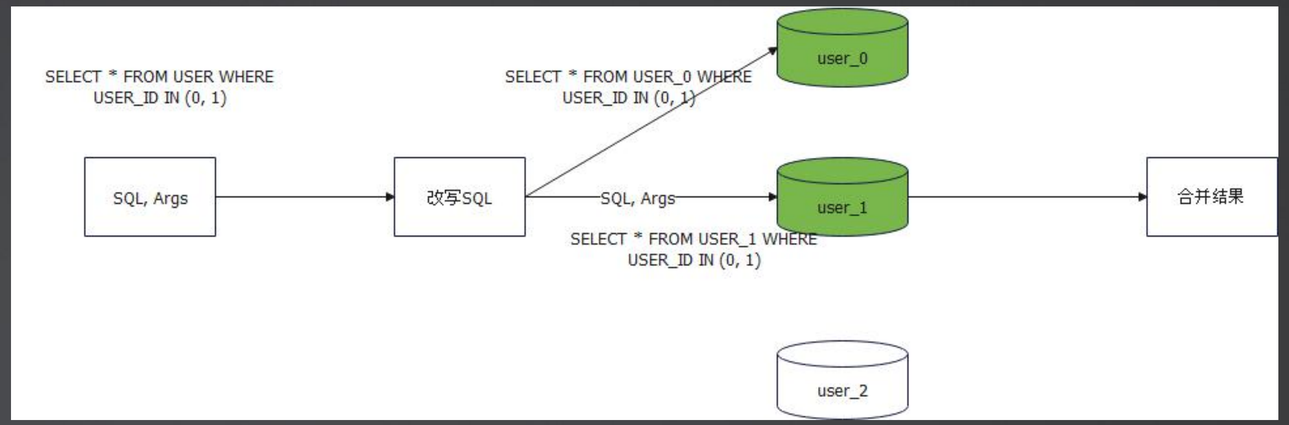
那么，求平均值改怎么改写

分库分表中间件——改写 SQL

改写 SQL 会有几种情况：

1. 只需要改个表名，数据库名。一般不需要全局状态的查询就是这样，例如单一数据查询。

2. 重写为多个 SQL：范围查询，或者需要全局状态的查询。



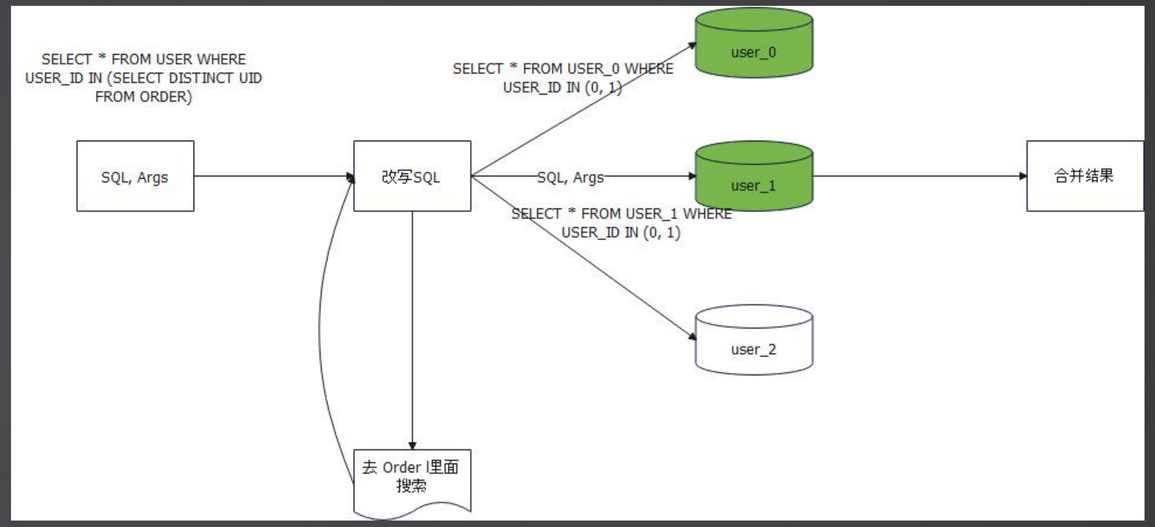
分库分表中间件——改写 SQL

改写 SQL 会有几种情况：

1. 只需要改个表名，数据库名。一般不需要全局状态的查询就是这样，例如单一数据查询。

2. 重写为多个 SQL：范围查询，或者需要全局状态的查询。

3. 发起多次查询：每次查询都可能出现情况 1 或者情况 2，典型的就是子查询和关联查询（难难难，非常难！）



问题：如果一个语句,SELECT * FROM ORDER, ORDER_ITEM，这两个表的分库分表规则一样，能不能只发起一次查询？

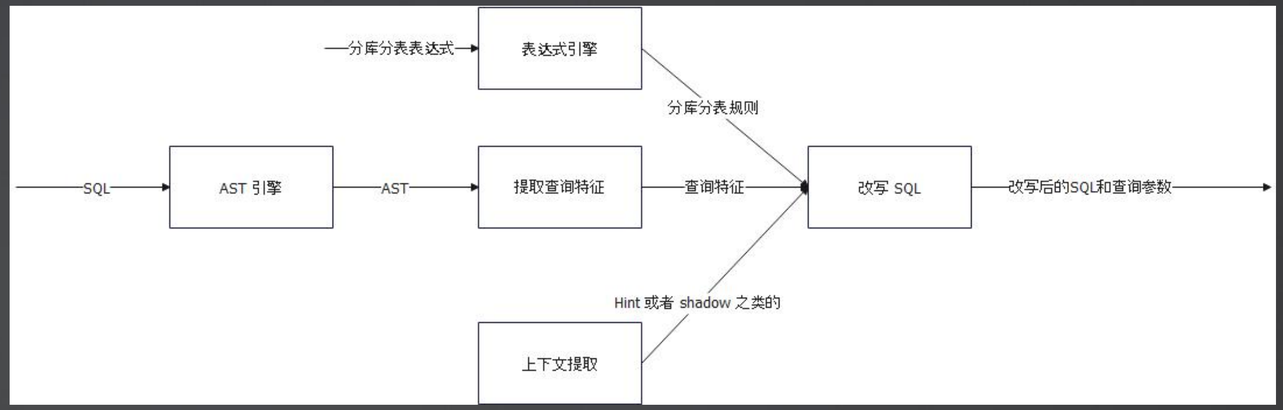
分库分表中间件——改写 SQL

改写SQL的关键点：

1. 分库分表规则：分库，决定了如何连上数据库，这个在后面执行 SQL 的时候需要使用；

2. 查询特征：查询条件，查询列，查询类型，是否子查询，是否关联；

3. 上下文：通常是指，用户针对某些请求手动指定的信息，比如说强制走某个库，或者说打到 shadow 或者压测库。



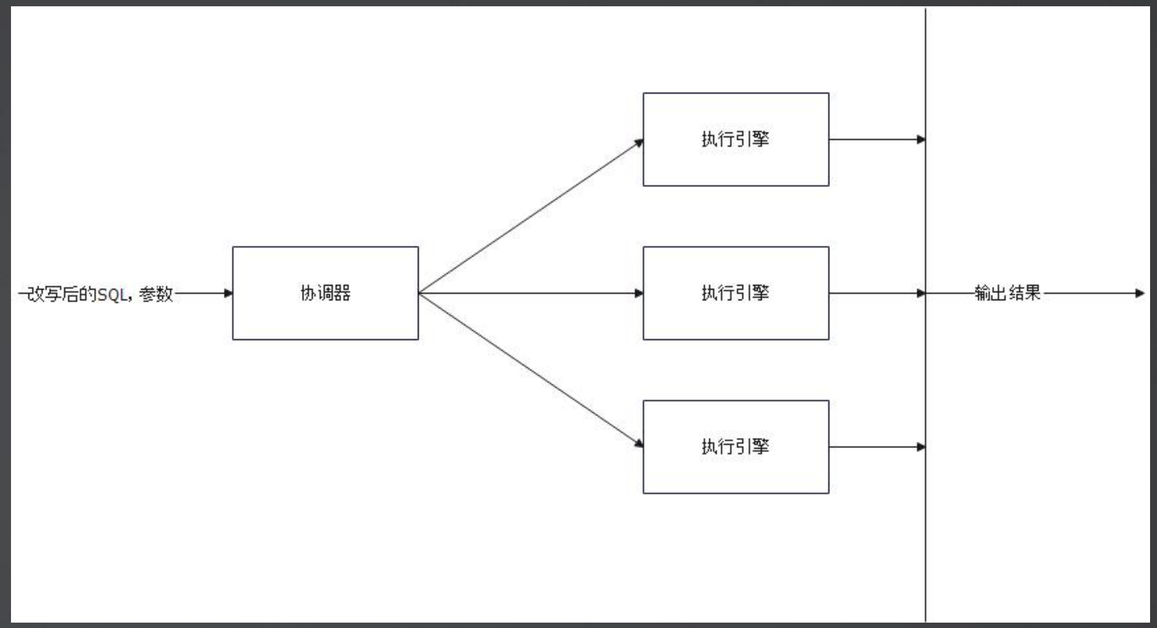
自增主键只是作为改写 SQL 的一个步骤，可以理解为额外多插入一个主键列，并且用生成的 ID 作为值。

分库分表中间件——执行 SQL

执行 SQL：

1. 根据找到目标库的连接信息，执行查询；

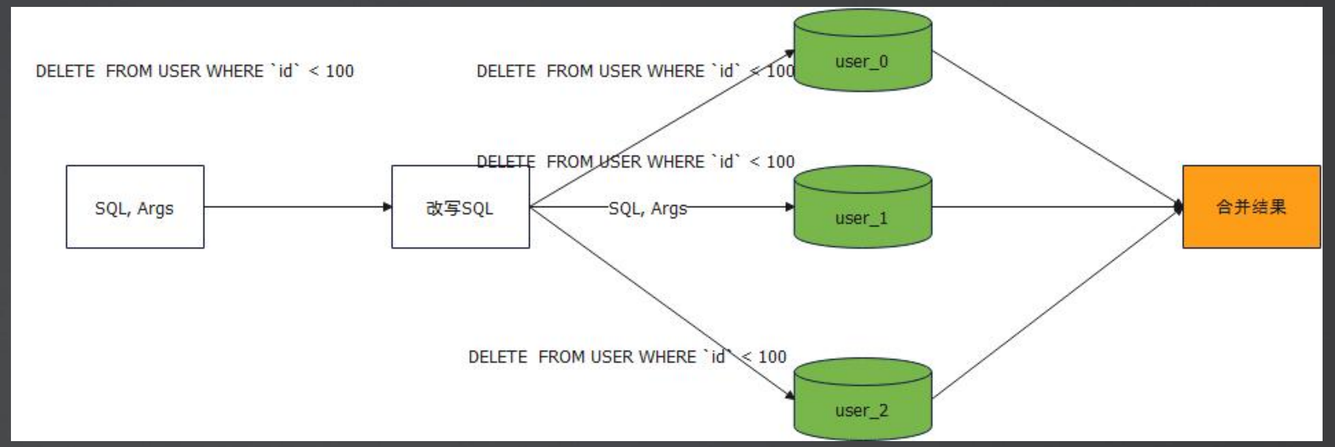
2. 协调所有的执行的 SQL，执行完毕之后步入到下一个阶段



分库分表中间件——执行 SQL

执行 SQL 的难点：

如果我们改写 SQL 的结果是生成了多个 SQL，那么怎么保证它们都成功或者都不成功？



右图中删除语句，要怎么执行？

它本质上就是一个分布式事务的问题。

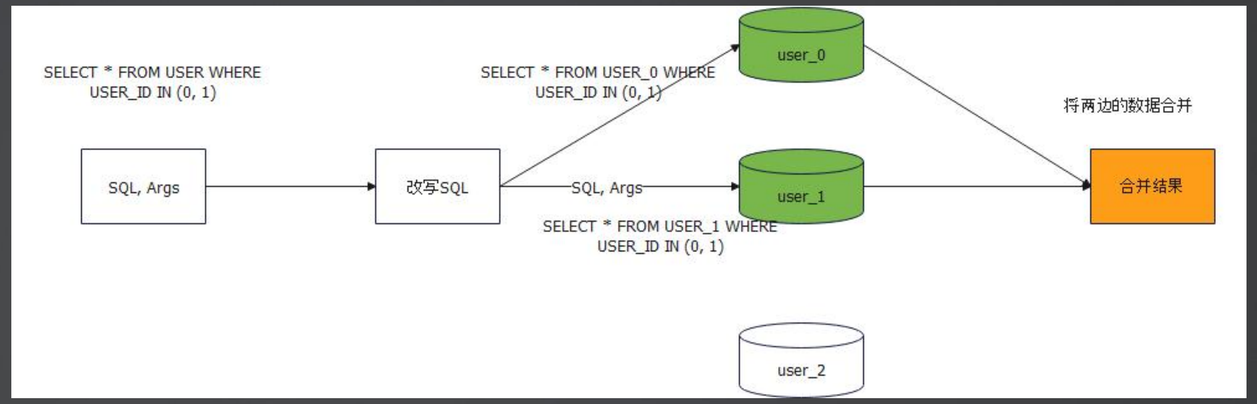
解决思路也就是分布式事务的解决思路。

主要实现思路是两阶段提交-XA

分库分表中间件——合并结果

合并结果也有几种情况：

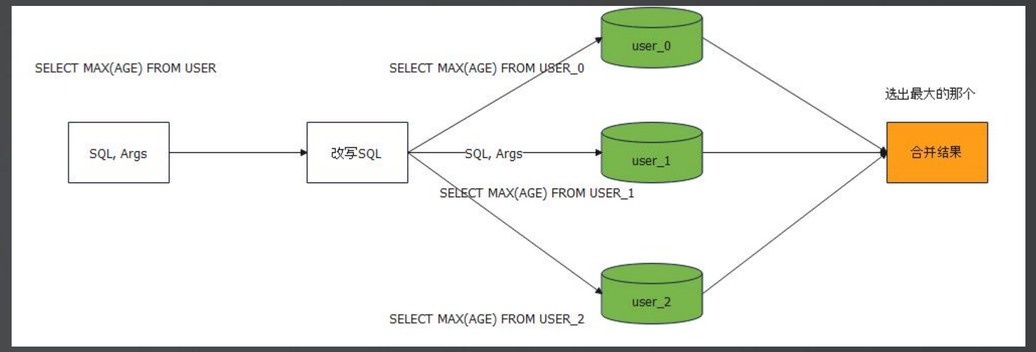
1. 合并为一个结果



分库分表中间件——合并结果

合并结果也有几种情况：

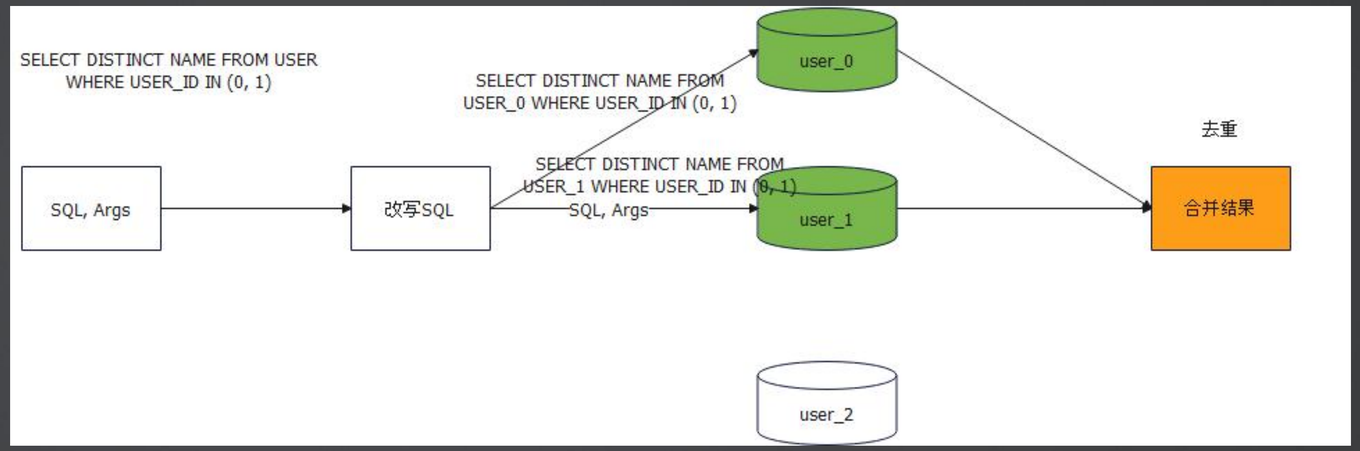
1. 合并为一个结果
2. 聚合：这一类是要全局状态



分库分表中间件——合并结果

合并结果也有几种情况：

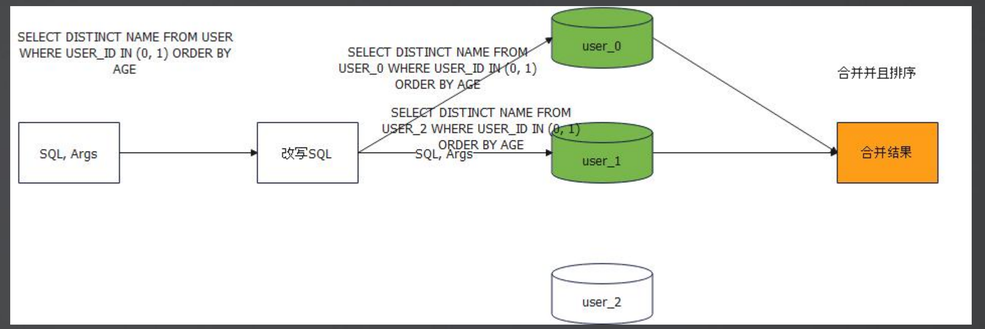
1. 合并为一个结果
2. 聚合
3. 去重



分库分表中间件——合并结果

合并结果也有几种情况：

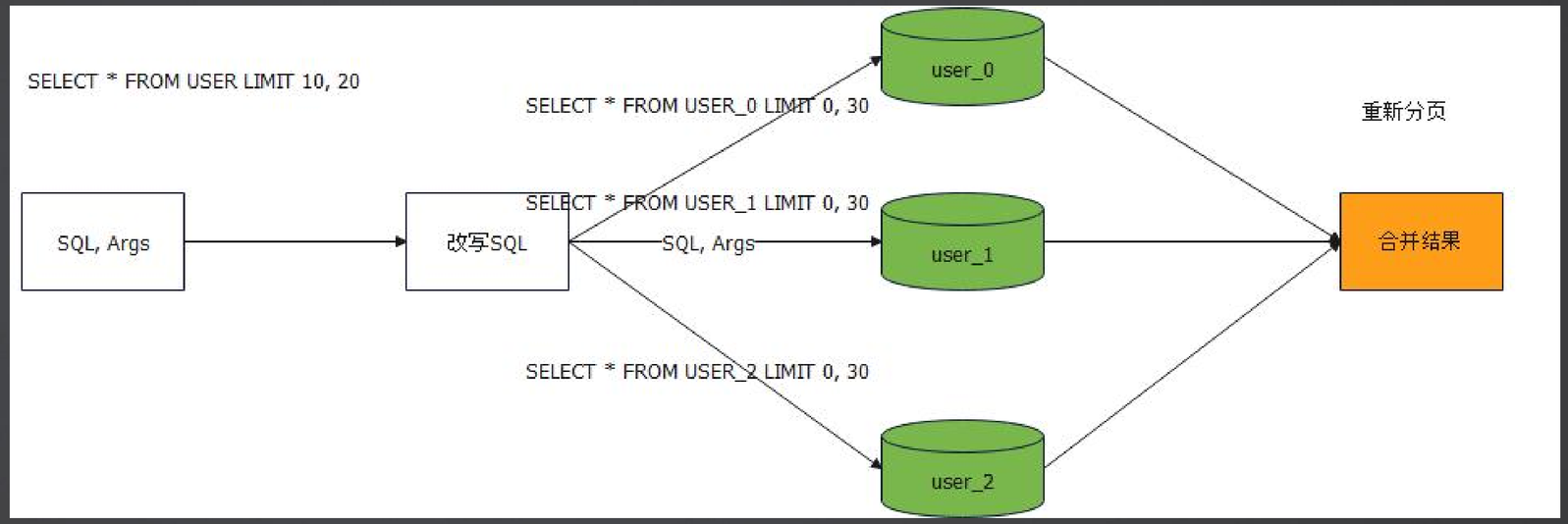
1. 合并为一个结果
2. 聚合
3. 去重
4. 排序



分库分表中间件——合并结果

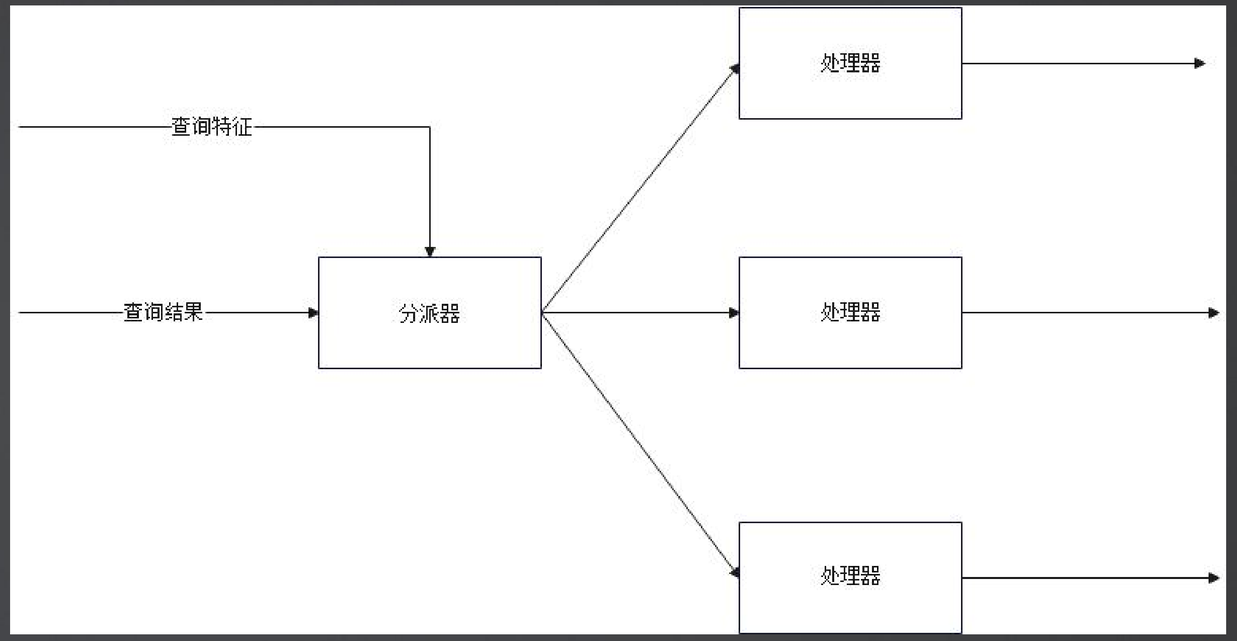
合并结果也有几种情况：

1. 合并为一个结果
2. 聚合
3. 去重
4. 排序
5. 分页：最复杂的部分



分库分表中间件——合并结果

关键在于，根据查询特征来分派处理器。



分库分表中间件——玩具版接口设计大略

```go
// Sharding 整个 sharding 的过程
// 其实就是接收一个 sql 和 参数，然后我们执行它，返回结果
// 我们也可以不用自己的接口，而是实现一个 sharding 的 driver
// 类似于 MySQL 的 driver
// 出于教学的目的，这里我们尝试自己定义接口，会更加清晰
type Sharding interface {
    Exec(ctx context.Context, sql string, args... interface{}) (sql.Result, error)
    Query(ctx context.Context, sql string, args... interface{}) (sql.Rows, error)
}
```

```go
type mockShard struct {
    rewriter Rewriter
    executor Executor
    merger Merger
}

func (m *mockShard) Query(ctx context.Context, sql string, args ...interface{}) (sql.Rows, error) {
    panic( v: "implement me")
}

// Exec 因为我们说 sharding 核心就是三个步骤
// 重写、执行和合并结果
// 于是我们引入三个接口来代表这三个过程
func (m *mockShard) Exec(ctx context.Context, sql string, args ...interface{}) (sql.Result, error) {
```

分库分表中间件——玩具版接口设计大略

```go
// Rewriter 代表重写 SQL
type Rewriter interface {
    Rewrite(ctx context.Context, rwCtx *RewriteContext) (*RewriteResult, error)
}

// RewriteContext 代表一个重写上下文，
// 里面放着你需要的各种数据。
type RewriteContext struct {
    sql string
    args []interface{}
}

// RewriteResult 代表重写后的结果
type RewriteResult struct {
    // 重写后的一堆查询
    queries []*RewriteQuery
}
```

```go
// RewriteQuery 重写后的 SQL
type RewriteQuery struct {
    sql string
    args []interface{}

    // 你可能需要一些查询特征字段，用于执行和合并结果阶段使用
    // 比如说要根据 dbname 去找到链接信息，
    // 特别是要考虑到后面合并结果的方式五花八门，
    // 这里的字段可能会非常丰富
    tableName string
    dbName string
}
```

分库分表中间件——玩具版接口设计大略

```go
// Executor 代表执行 SQL
type Executor interface {
    // Execute 这里额外返回一个错误，是我们自身的错误，而不是执行查询引起的错误
    Execute(ctx context.Context, exeCtx *ExecuteContext) (*ExecuteResult, error)
}

// ExecuteContext 代表一个执行上下文
type ExecuteContext struct {
    queries []*RewriteQuery
}

// ExecuteResult 执行结果
type ExecuteResult struct {
    results []*QueryResult
}

type QueryResult struct {
    // 合并结果的时候，Merger 的实现自己知道该用什么字段不该用什么字段
    queryRows *sql.Rows
    err error
    execRes sql.Result

    // 可以考虑改进接口，也可以直接在这里保留
    query *RewriteQuery
}
```

分库分表中间件——玩具版接口设计大略

```go
// Merger 代表合并结果
// 这个接口会有很多很多的实现，
type Merger interface {
    Merge(ctx context.Context, mergeCtx *MergeContext) (*MergeResult, error)
}

type MergeContext struct {
    result *ExecuteResult
}

type MergeResult struct {
    rows *sql.Rows
    result sql.Result
    error error
}
```

分库分表中间件——玩具版接口设计大略

```go
// Exec 因为我们说 sharding 核心就是三个步骤
// 重写、执行和合并结果
// 于是我们引入三个接口来代表这三个过程
func (m *mockShard) Exec(ctx context.Context, sql string, args ...interface{}) (sql.Result, error) {
    rwRes, err := m.rewriter.Rewrite(ctx, &RewriteContext{
        sql: sql,
        args: args,
    })
    if err != nil {
        return nil, err
    }
    exeRes, err := m.executor.Execute(ctx, &ExecuteContext{queries: rwRes.queries})
    if err != nil {
        return nil, err
    }

    mergeRes, err := m.merger.Merge(ctx, &MergeContext{
        result: exeRes,
    })
    if err != nil {
        return nil, err
    }
    return mergeRes.result, err
}
```

分库分表中间件——玩具版接口设计大略

```go
// astRewriter 比如说利用 AST 来实现重写
type astRewriter struct {
    cfg *ShardingConfig
}

func (a *astRewriter) Rewrite(ctx context.Context, rwCtx *RewriteContext) (*RewriteResult, error) {
    // 在这里，构建起 AST 树
    // 修改 AST 的节点。比如说插入主键节点，然后节点的值就是主键生成策略生成的值
    // 最后将 AST 转为一个 SQL
    panic( v: "implement me")
}

type ShardingConfig struct {
    // 这里就是你的各种 sharding 的配置
    // 比如说你的表怎么sharding
    // db 怎么 sharding
    // 主键生成策略是什么
    // 不同db的连接信息
    // ... 可以参考 shardingsphere 的配置文件，非常丰富
}
```

分库分表中间件——玩具版接口设计大略

```go
// 简单的遍历执行
type simpleExecutor struct {
    // 维持住了所有的物理表创建的 DB
    // 它基本上是在初始化的时候根据配置来创建的
    dbConn map[string]*sql.DB
}

func (p *simpleExecutor) Execute(ctx context.Context, exeCtx *ExecuteContext) (*ExecuteResult, error) {
    queryResult := make([]*QueryResult, 0, len(exeCtx.queries))
    for _, query := range exeCtx.queries {
        db, ok := p.dbConn[query.dbName]
        if !ok {
            // 要么是用户没有配置，要么是 sharding 出错了
            return nil, errors.New( text: "invalid sharding queries")
        }

        // 这里要判断是SELECT 还是 UPDATE 之类的
        res, err := db.ExecContext(ctx, query.sql, query.args...)
        // 或者是
        // rows, err := db.QueryContext(ctx, query.sql, query.args...)
        queryResult = append(queryResult, &QueryResult{ query: query, err: err, execRes: res})
    }
    return &ExecuteResult{results: queryResult}, nil
}
```

分库分表中间件——玩具版接口设计大略

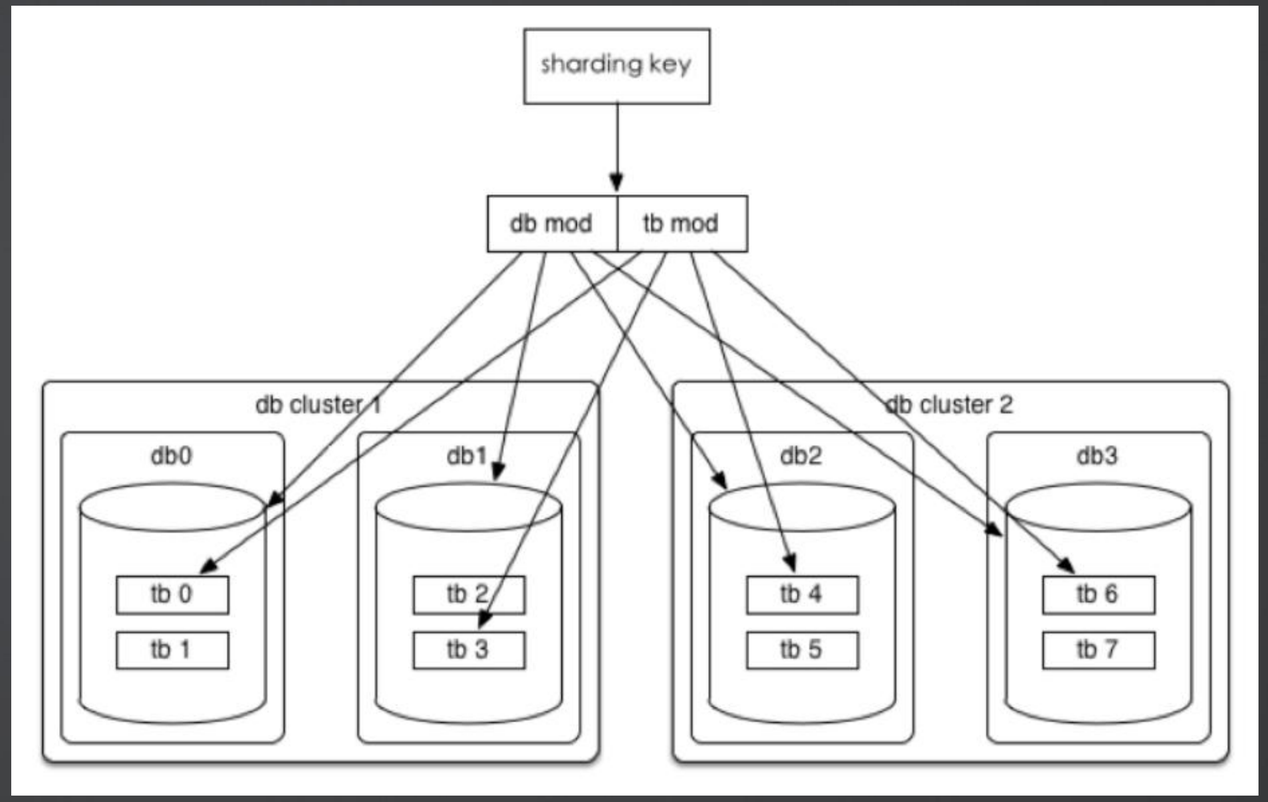
```go
type dispatcherMerger struct {
    // 处理平均值的
    avgMerger Merger

    // 处理计数的
    cntMerger Merger

    // ... 你会有一大堆

}

func (d *dispatcherMerger) Merge(ctx context.Context, mergeCtx *MergeContext) (*MergeResult, error) {
    // 检查 mergeCtx 里面的每一个查询结果和查询特征，然后选择
    // if queryCnt(mergeCtx) {
    //     return cntMerger.Merge(ctx, mergeCtx)
    // }
    panic( v: "implement me")
}
```

目录

分库分表实践案例——美团点评

订单分库分表 32 * 32

- user id 后四位 mod 32 分库：user_id %32
- user id 后四位 div 32 mod 32 分表：(user_id / 32) % 32
- 八个主从集群，每个主从集群四个库
- 主键：时间戳+用户标识码+随机数，因而订单号自带分库分表信息
- 额外再保存一份数据，按照 shop id 进行 8*8 分库分表

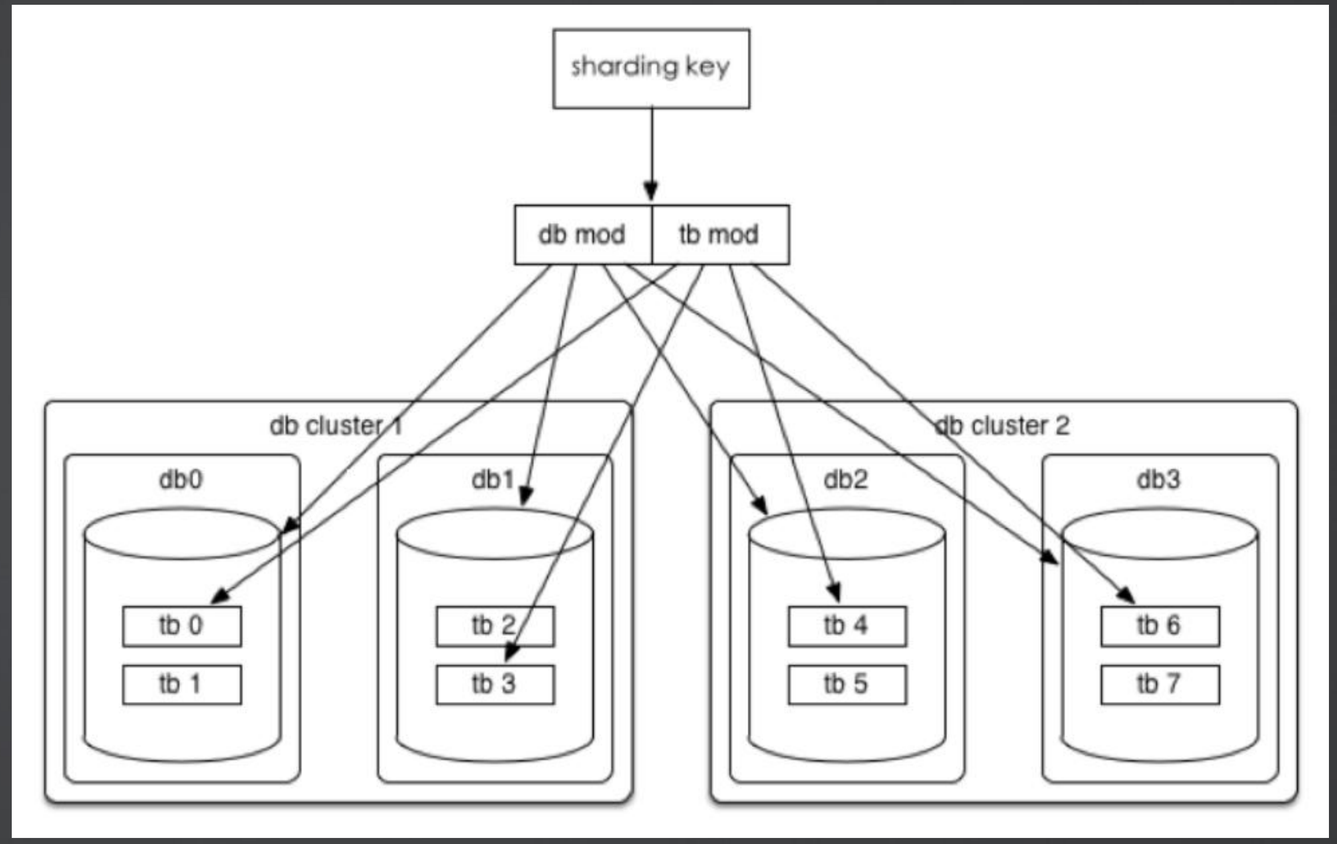


目录

分库分表引入的问题

分库分表引发的问题：

- 主键问题
- 分布式事务问题
- 分页问题
- 不同维度查询



分库分表引入的问题——主键问题

主键生成策略：

- UUID：非自增，插入性能不好
- snowflake：依赖于机器 ID，需要设置集群
- 数据库集群方案：设置步长，利用表的自增来生成

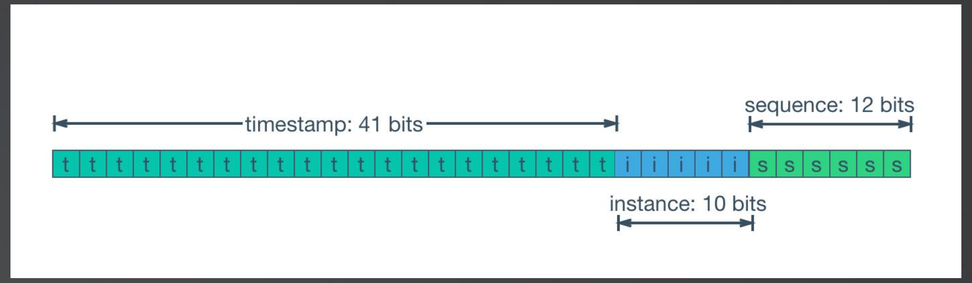
分库分表引入的问题——主键问题

设计自己的主键生成策略：基本上就是参考 snowflake 算法做各种变种，关键在于三个：

1. 时间：要考虑自己设计的主键生成策略能用几年。还要考虑时间的单位。比如说用 38 位 bit 来表达从 2021-01-01开始的毫秒数，大概可以用不到 9 年；

2. instance 或者说 workerid：中间位。要考虑自己要用多大的集群来生成主键，和自己的并发量有关系；

3. 序号 ID：和时间相关。一般在时间采用毫秒作为单位的情况下，这里考虑的就是一毫秒内最大的并发可能是多少。



其余考量：主要是考虑在主键要不要带上一些特殊的业务含义，例如美团点评那边带上了用户 ID 作为一部分。

这种设计的方案是大致递增，但是不是严格递增。

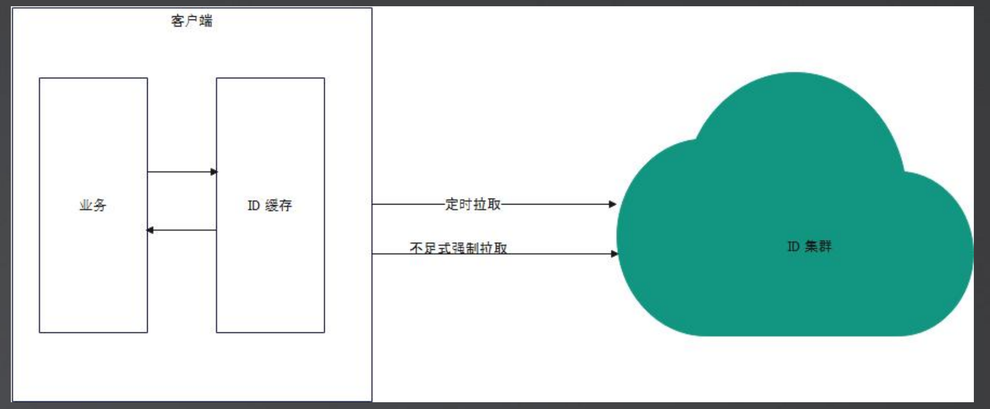
分库分表引入的问题——主键问题

主键生成服务客户端优化：当我们把主键生成服务单独部署成为一个集群的时候，可以尝试提供客户端来优化性能：

1. 客户端缓存：客户端拉取可以拉很大一批 ID，然后缓存在本地，每次从本地直接分配。可以进一步考虑 thread-local 的机制，避免线程的竞争。

2. 提前拉取。客户端可以定时询问集群，每次拉取一大批主键 ID 用于同一个业务。要有兜底方案，在提前拉取的 ID 快被消耗完的时候，拉取一批 ID。

3. singleflight 拉取 ID：即便触发了从集群拉取 ID 的动作，也要控制住，一般来说多个 Goroutine 触发，只需要有一个去拉取 ID 就可以。要有兜底机制，防止在拉取过程中，又来了新的需求，导致这一次拉取的 ID 不够分配。



分库分表引入的问题——分布式事务

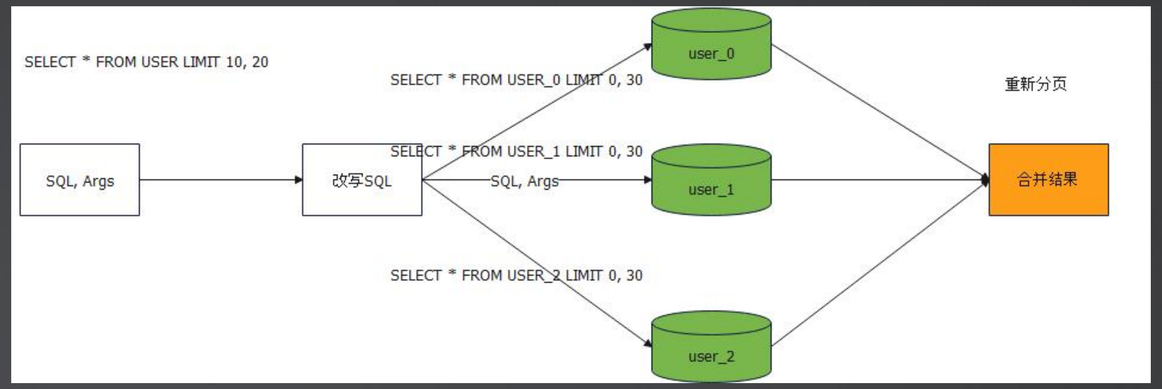
原本的本地事务已经无法使用了：

- 引入分布式事务框架
- 采用柔性事务解决方案，例如 SAGA 和 Event Sourcing 之类的方案

分库分表引入的问题——分页问题

分页的常见方案：

- 全局视野法，即改写 SQL，也就是中间件用的：
- SELECT * FROM XXX LIMIT X, Y
  SELECT * FROM XXX LIMIT 0, X+Y
- 然后拿到所有分库分表的结果，再进行排序
- 要发起 N 次查询，总共拉取 N * (X +Y) 条数据，再内存排序
- 缺点
  - 网络传输量大
  - 应用排序性能差
  - X 和 Y 任何一个大，数据库性能都差

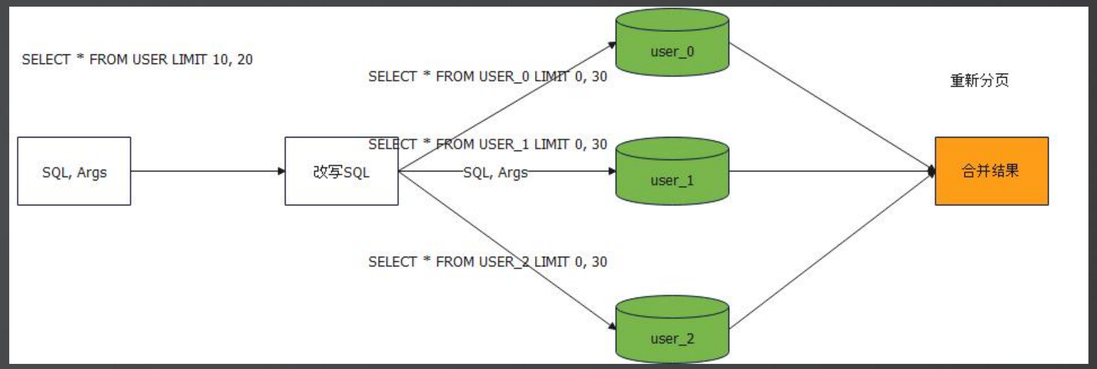


分库分表引入的问题——分页问题

改进：禁止使用偏移量，同时加上一个条件作为偏移量的替代。

- SELECT * FROM XXX LIMIT 0, Y
- 下一次查询，变成 SELECT * FROM XXX WHERE ID> $(max_id) LIMIT 0, Y
- 其中 max_id 是上一次查询返回的最大 ID

- 但是依旧需要应用排序，但是数据量被固定为 N * Y



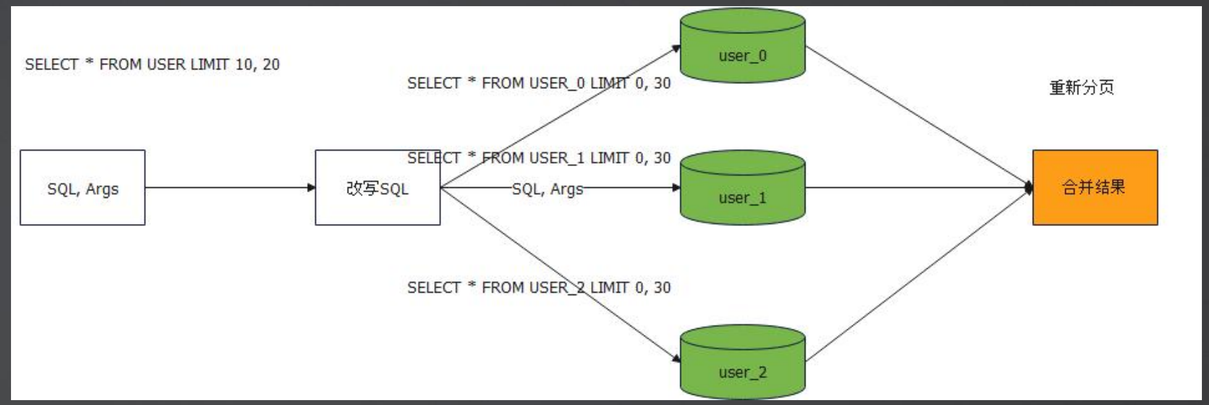
分库分表引入的问题——分页问题

模糊查询法，如果不在乎精度，或者说不在乎排序的准确度，那么可以用这个模糊查询法。

SELECT * FROM XXX LIMIT X, Y

那么N个库就是 SELECT * FROM XXX LIMIT X/N, Y/N

相当于是每个库都搞一点，最后合并在一起，同样也需要应用内再次排序，但是数据量稳定在 Y

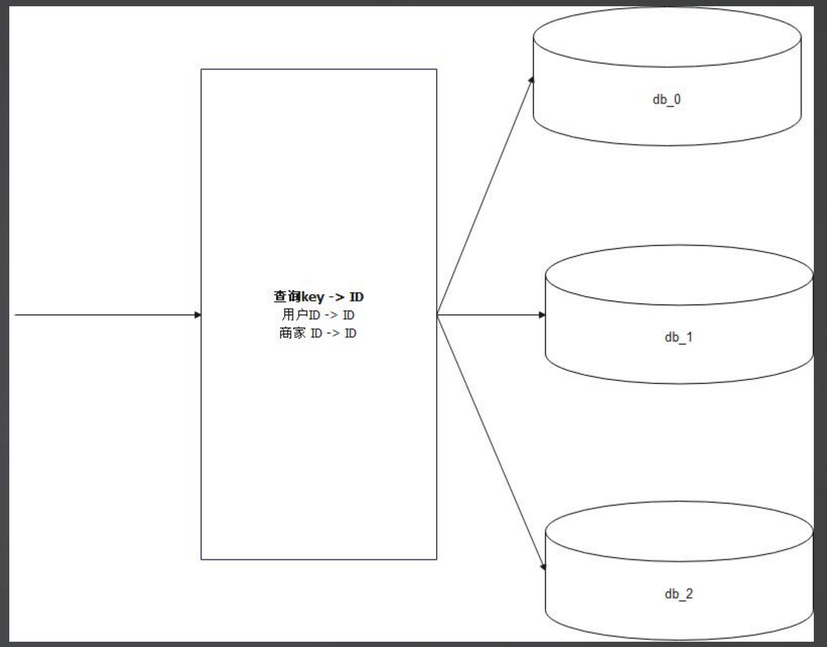


分库分表引入的问题——不同维度查询

问题：比如说订单表，按照用户 ID 来分的话，那么商家在查询的时候，就无法利用分库分表的特性，就需要在全部表里面查询一遍

解决思路：

- 复制：将数据复制一份，按照商家 ID 进行分库分表。一般来说只需要复制主表，其余数据可以用主表的业务 ID 来进行二次查询获得
- 中间 mapping：这种一般是用主键作为分库分表，然后建立查询键到主键的映射



代码地址

https://github.com/flycash/geekbang-go-camp

THANKS