# 概述

# 原理

## 数据库锁

## 写时复制

## MVCC

## 读写分离

## 负载均衡

## 池化

### 线程池

### 连接池

## 热点数据

# 优秀方案

## TDSQL

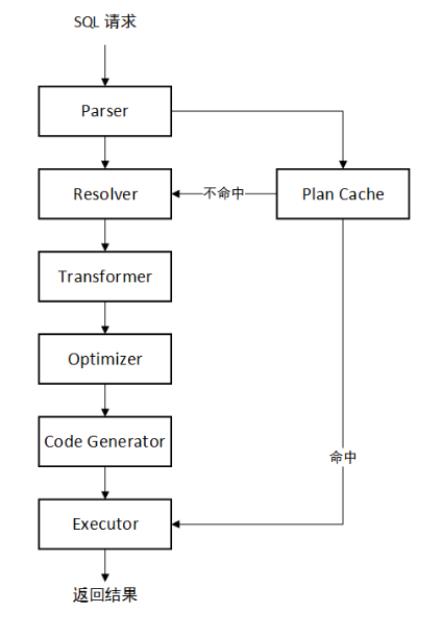
## OceanBase

### OBProxy

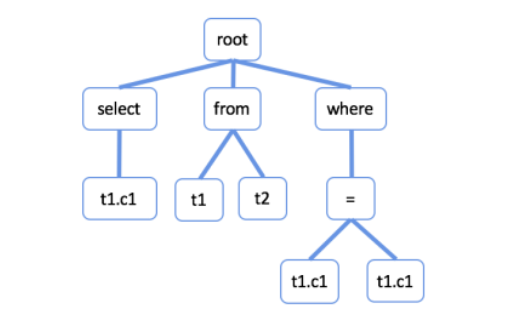
#### 读写分离

### SQL引擎

#### 快速参数化



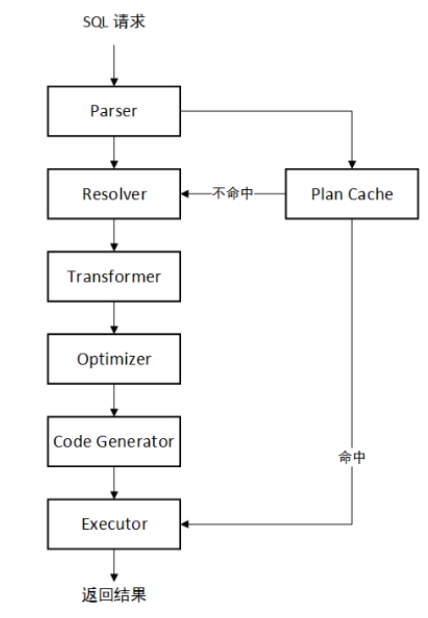
Parser（词法/语法解析模块）：  
 在收到用户发送的SQL请求串后， Parser会将字符串分成一个个的“单词”，并根据预先设定好的语法规则解析整个请求，将SQL请求字符串转换成带有语法结构信息的内存数据结构，我们称为“语法树”（Syntax Tree）。  
 为了加速SQL请求的处理速度，OceanBase对SQL请求采用了特有的**“快速参数化”**，以加速查找plan cache的速度。



#### 缓存执行计划树

#### 查询重写

#### 并行优化



Optimizer（优化器）：  
 优化器是整个SQL请求优化的核心，其作用是为SQL请求生成最佳的执行计划。  
 在优化过程中，优化器需要综合考虑SQL请求的语义、对象数据特征、对象物理分布等多方面因素，解决访问路径选择、连接顺序选择、连接算法选择、  
分布式计划生成等多个核心问题，最终选择一个对应该SQL的最佳执行计划。  
 为了充分利用OceanBase的分布式架构和多核计算资源的优势，OceanBase的查询优化器会对执行计划做并行优化：**根据计划树上各个节点的数据分布，对串行执行计划进行自底向上的分析，把串行的逻辑执行计划改造成一个可以并行执行的逻辑计划**。

#### 执行器

## TiDB

## GoldenDB

### proxy

#### LVS

#### 线程池

#### 连接池

#### 读写分离

#### 黑名单

对于某些耗时比较久的SQL，会加入黑名单中。

### SQL引擎

#### 缓存执行计划树

#### 查询重写

#### 并行优化

对于UNION可以采用并行的方式，分发到不同group。

#### 元数据缓存

Proxy本地会缓存元数据，当MDS发生元数据变更的时候会推送到proxy。

#### force index

#### MPP

### GTM

#### 批量申请GTID

问题：每个事务都要申请一次GTID，与GTM交互频繁。

性能提升机制：计算节点批量申请GTID，GTM一次执行多个计算节点的批量申请。

#### 批量释放GTID

问题：每个事务都要释放GTID，与GTM交互频繁。

性能提升机制：计算节点批量释放GTID，GTM一次执行多个计算节点的批量释放。

#### 批量查询GTID

问题：每个事务都要查询活跃事务列表，与GTM交互频繁。

性能提升机制：计算节点汇总多次查询，GTM一次执行多个计算节点的汇总查询。

#### GTM横向扩展

问题：多集群共用GTM，导致GTM压力大。

性能提升机制：支持多GTM部署，最多一个集群独占一套GTM。

效果：减少计算节点与GTM交互次数，减少GTM日志落盘次数、主从复制次数。

### DB

#### 写时复制

#### MVCC

#### 快同步

#### 备机IO线程和SQL线程读写锁拆分

**优化原理：**

借鉴binlog的写入和dump过程的读写锁拆分逻辑，优化MTS并行复制读写锁拆分，复用MYSQL\_BIN\_LOG对象中LOCK\_binlog\_end\_pos锁及binlog\_end\_pos变量，binlog\_end\_pos变量在此处记录了IO线程写入relay\_log文件的最大位置。而SQL线程读取relay\_log时，判断不可以超过该binlog\_end\_pos位置即可。

IO线程将event时间写入relay\_log后，将会持有LOCK\_binlog\_end\_pos锁，并且更新binlog\_end\_pos值为当前正在写io\_cache的最大位置。

SQL线程从relay\_log读取evnet事件时，不再需要持有relay\_log::LOCK\_log锁。只需要持有LOCK\_binlog\_end\_pos锁，获取binlog\_end\_pos位置。然后通过该位置和relay\_log文件名判断读取relay\_log到何处停止。

**优化效果：**

对于低并发以及写压力不大的场景（备机同步和回放是同一个relay\_log），性能提升非常明显，可以达到25%左右。在高并发下，可能没有性能提升。

#### 主备收发event合并优化

原来的binlog发送是binlog event一个个发送，备机的IO线程在接收event时，也是一个个event进行处理的。

由于通常一个事务的binlog是由多个event组成的，而且仅在一组binlog event的最后一个event需要给主机回响应。所以，考虑此处将多个binlog event强制合并为一个网络包发送。

主机将多个binlog event合并为一个大的网络包逻辑为：

1. 如果网络包超过16M，则直接发送，不再继续组合剩余的binlog event
2. 如果组合到一个需要给主机回响应的event时，那么本次组包结束。即当一个组提交的所有binlog events小于16M时，则一个组提交作为一个大包发送到备机

备机解析多个binlog event事件逻辑：

原本的备机IO线程从网络包中取出1个event事件，解析后再写入relaylog中。现在从网络包中取出的是多个event的组合，需要循环解析event，直至结束位置。

**优化效果：**

在不同场景下，性能提升可以达到10%~20%。

#### semi-sync插件内置优化

semi-sync主备复制是一个插件，在实际运行过程中，是以插件的形式加载的。

Mysql中所有插件的调用，都需要被一把插件读写锁管理，在一个事务提交过程中，mysql与semi-sync插件有频繁的交互，导致这把插件读写锁成为一个非常热点的资源。

因此，考虑将semi-sync插件内置到mysqld进程中，不再以插件的形式加载。整个内置的过程，利用了大部分semi-sync源码，在mysql层实现了同样的功能，对业务无感。同时，由于不再是插件的模式，因此不再执行INSTALL/UNINSTALL semi-sync plugin命令的执行。

**优化效果：**

在实际测试中，优化前的版本，从1主备增加到1主3备，下降不大，但是再次增加到1主5备，甚至1主7备时，性能急剧下降，只有1主1备的一半性能。

优化后的版本，从1主1备到1主7备，性能全都一致。相当于优化后的版本，在1主多备的场景下，性能能够提升50%~100%左右。

#### 备机回放位置信息存储修改为TABLE模式

优化前配置项：

master-info-repository=FILE

rela-log-info-repository=FILE

优化后配置项：

master-info-repository=TABLE

rela-log-info-repository=TABLE

配置项为FILE时，表示的master\_info和relay\_log\_info均使用文件形式存储主备复制相关的位置信息等。默认在data/data目录下。实际运行过程中，由于文件中存储的同步位置和回放位置，需要不断的更新，这就造成了2个文件需要不断的flush和sync操作。默认的sync\_master\_info=10000，sync\_relay\_log=10000，sync\_relay\_log\_info=10000，表示10000个event做一次sync操作。这种模式不但对性能稍微有一点影响，主要还会造成异常断电等场景下，位置记录丢失等问题。

配置项为FALSE时，表示master\_info和relay\_log\_info使用系统表存储，默认使用的是InnoDB表，其记录的更新和持久化将与实际业务一起，不会产生单独的IO操作，不但对性能没有影响，也可以利用innodb的crash-recovery来解决异常恢复的问题。

#### binlog大小修改为100M

优化前配置（10M）：

max\_binlog\_size=10485760

优化后配置（100M）：

max\_binlog\_size=104857600

该配置项表示binlog的大小达到多大切换到下一个binlog文件。

由于binlog过小导致了发生频繁的切换，而binlog切换时是阻塞当前所有的写事务的，影响非常大。因此将binlog大小调大，减少切换次数。

优化效果：

实际测试从10M调整到1G时，性能可以提升5%~10%左右。

### Loadserver

#### 多线程

将文件拆分为多个，分别用不同线程处理。

### 存在问题

1. 没有处理group的热点数据

有对应的监控信息，但是这不能在线解决热点数据，只能作为事后处理。

1. 由于元数据推送问题导致读写分离错误

有这样的一个情况，CM会在主备关系发生变化的时候才会推送给MDS最新的元数据（主备关系），此时如果dbproxy重启，则会自动重置主备关系，这个时候可能就会发送到旧的备机（select操作），这个应该让CM定期发送（而不是变化的时候才发送），proxy可以获取最新的主备关系。