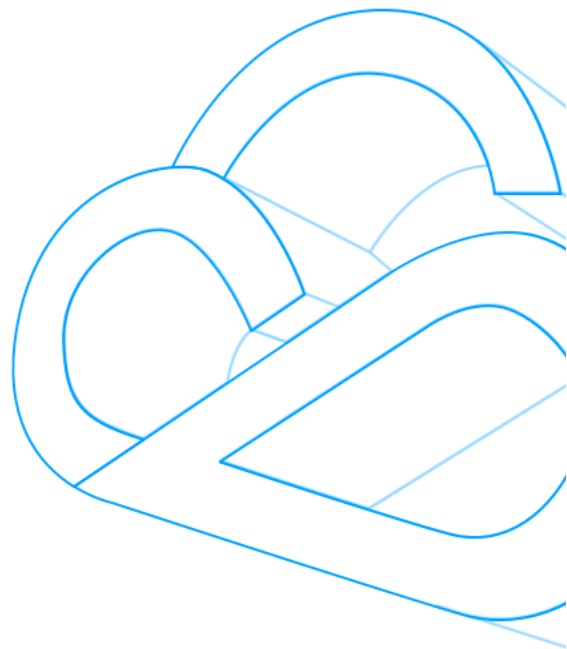




腾讯云数据库 TDSQL MySQL 版 分布式 V10.3.22.8/集中式 V8.0.22.8 调优



文档版本：

发布日期：

腾讯云计算（北京）有限责任公司

版权声明

本文档著作权归腾讯云计算（北京）有限责任公司（以下简称“腾讯云”）单独所有，未经腾讯云事先书面许可，任何主体不得以任何方式或理由使用本文档，包括但不限于复制、修改、传播、公开、剽窃全部或部分本文档内容。

本文档及其所含内容均属腾讯云内部资料，并且仅供腾讯云指定的主体查看。如果您非经腾讯云授权而获得本文档的全部或部分内容，敬请予以删除，切勿以复制、披露、传播等任何方式使用本文档或其任何内容，亦请切勿依本文档或其任何内容而采取任何行动。

商标声明



“腾讯”、“腾讯云”及其它腾讯云服务相关的商标、标识等均为腾讯云及其关联公司各自所有。若本文档涉及第三方主体的商标，则应依法由其权利人所有。

免责声明

本文档旨在向客户介绍本文档撰写时，腾讯云相关产品、服务的当时的整体概况，部分产品或服务在后续可能因技术调整或项目设计等任何原因，导致其服务内容、标准等有所调整。因此，本文档仅供参考，腾讯云不对其准确性、适用性或完整性等做任何保证。您所购买、使用的腾讯云产品、服务的种类、内容、服务标准等，应以您和腾讯云之间签署的合同约定为准，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

修订记录

文档版本	发布日期	修订人	修订内容
------	------	-----	------

目录

修订记录.....	ii
目录.....	iii
前言.....	iv
1 调优概述.....	1
2 性能调优.....	3
2.1 服务器调优.....	3
2.2 数据库配置参数优化.....	7
3 SQL 调优.....	9
3.1 了解执行计划.....	9
3.1.1 底层索引结构.....	9
3.1.2 执行计划解读.....	10
3.2 利用执行计划等优化 SQL	15
3.3 表结构、表空间、DDL 优化	22
3.4 合理规划 SQL 大小与事务大小，优化应用与数据库的交互.....	23
3.5 减小对数据库的总读写量、计算/存储压力	24
3.6 优化业务逻辑，提高应用并行能力.....	25
3.7 架构与读写分离.....	25

前言

文档目的

本文档用于帮助用户掌握云产品的操作方法与注意事项。





目标读者

本文档主要适用于如下对象群体：

- 客户
- 交付 PM
- 交付技术架构师
- 交付工程师
- 产品交付架构师
- 研发工程师
- 运维工程师

符号约定

本文档中可能采用的符号约定如下：

符号	说明
 说明：	表示是正文的附加信息，是对正文的强调和补充。
 注意：	表示有低度的潜在风险，主要是用户必读或较关键信息，若用户忽略注意消息，可能会因误操作而带来一定的不良后果或者无法成功操作。
 警告：	表示有中度的潜在风险，例如用户应注意的高危操作，如果忽视这些文本，可能导致设备损坏、数据丢失、设备性能降低或不可预知的结果。
 禁止：	表示有高度潜在危险，例如用户应注意的禁用操作，如果不能避免，会导致系统崩溃、数据丢失且无法修复等严重问题。

1 调优概述

说明:

由于优化方式共通性，性能优化章节部分内容引用于互联网公开文档。

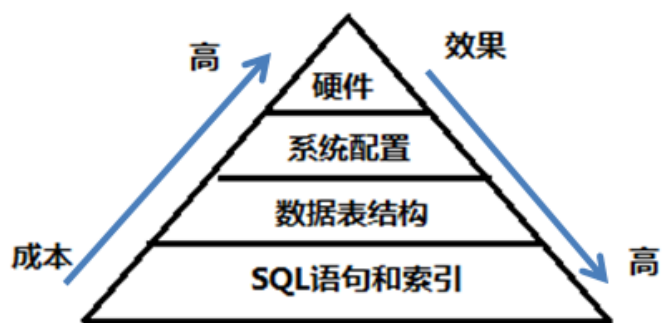
数据库的优化，是需要结合整个业务系统去考虑的。若数据库存在瓶颈，可以考虑优化业务端逻辑，减小对数据库的总读写量、计算/存储压力。如读写总量和压力已不便优化，那么就需要结合业务操作数据库的逻辑，进行数据库端优化。

数据库优化在资源的视角上，可以认为是在充分利用内存的情况下，减少 CPU 占用、减少磁盘 IO(尤其是随机 IO)、减少网络 IO、减少争用等待和死锁。

数据库优化在优化对象上，则通常又可以分为库表及 SQL 优化、数据库自身参数等方面优化、架构优化、硬件与系统优化、业务逻辑优化等内容。

- 对于架构优化，应当在系统设计时充分结合业务的实际情况，一方面结合读写并发模型及量级，选择集中式实例或分布式实例，考虑使用读写分离策略；另一方面考虑把不适合数据库做的事情放到数据仓库、搜索引擎或者缓存中去做。
- 对于库表及 SQL 优化，其需要和业务逻辑的充分优化紧密配合，确认业务表结构设计合理，SQL 语句优化足够，索引有较高的使用率等。
- 对于数据库自身参数等方面优化，通常与硬件资源有关，也会考虑数据安全等方面的内容。通常在满足数据安全前提下，开放了足够的资源使用权限后，没有太多的参数可以再进行调整。
- 对于系统优化、硬件方面的优化，通常考量整个系统的瓶颈在哪里，选择场景匹配的硬件资源、系统环境，适当调整硬件、系统参数使其适合数据库发挥性能。

上述几个方面，架构设计、硬件选择是基础，而在给定的架构、硬件条件下，提高业务逻辑和数据库 SQL 优化的契合程度，是主要的优化方向。如果业务性能的瓶颈是由于索引等数据库层优化不够导致，那么再好的架构和硬件也无法支撑高性能需求的应用。因此，我们给出如下图示表示几种优化的成本、效果关系。本文后续章节主要描述库表及 SQL 优化与业务逻辑优化这两个方面，但也会涉及部分硬件、系统、架构、数据库自身参数等方面的内容。



2 性能调优

2.1 服务器调优

硬件优化

CPU 类型选择

根据不同业务选择不同的 CPU 类型：

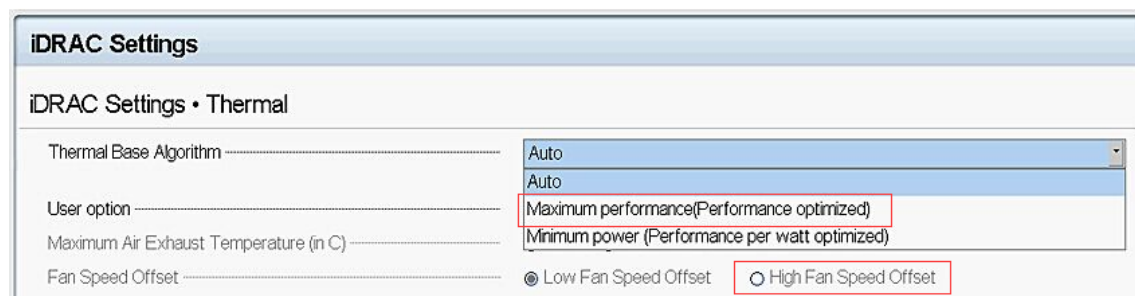
- 复杂计算或并发数较多的，可以选择主频和核数较多的 CPU。
- IO 密集型可以选择中等配置 CPU，但需要选择更好配置的磁盘，例如 Nvme-SSD 磁盘等。

BIOS 设定：建议优先选择性能模式而非节能模式

1. 主板等选择性能模式

如某服务器的 iDRAC（Integrated Dell Remote Access Controller）中 Thermal Base Algorithm 选 maximum performance：最大性能选项可以提供更好的性能，对处理器和内存的散热响应更加积极，以增加风扇功率这一点点代价换来稳定的性能。Auto 设置时，这个选项映射至 System BIOS > System BIOS Settings. System Profile Settings。

Fan Speed Offset 选 High Fan speed offset：这个选项的设置是使风扇速度接近全速运转（大约 90%-100%）。如果服务器上安装了高功率的 PCIE 卡或其他设备时，这个设置比较有用。



i 说明：

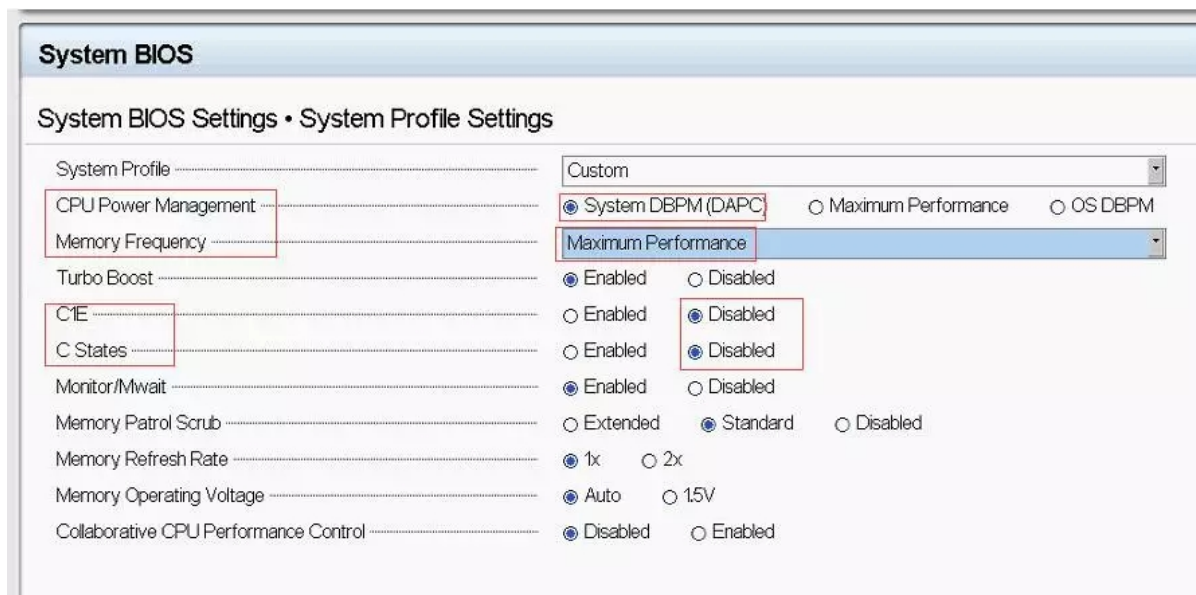
截图来自于某服务器厂商网站。

2. CPU 选择最大性能模式

CPU 选择 Maximum Performance，发挥最大功耗性能。

C1E，处理器处于闲置状态时启用或禁用处理器切换至最低功耗状态，建议关闭（默认启用）。

C States（C 状态），启用或禁用处理器在所有可用电源节能状态下运行，建议关闭（默认启用）。



说明:

截图来自于某服务器厂商网站。

3. 内存频率选择最大性能模式

Memory Frequency（内存频率）选择 Maximum Performance（最佳性能）。

4. 关闭 NUMA

视情况关闭服务器 NUMA。

网卡

- 建议万兆及以上网卡。
- 使用 bond 或者 team 技术的链路聚合模式或主备模式。

硬盘

- 文件系统用 XFS，服务器的 SAS 盘建议用 raid5 作为系统盘，SSD 盘建议用 raid0 作为数据库用盘，并且使用常用优化参数挂载，如下命令：

```
/bin/mount -t xfs /dev/md0 /data1 -o  
noatime,nodiratime,nobarrier,largeio,inode64,swalloc,sunit=1024,swidth=4096
```

- 磁盘 IO 调度策略，机械磁盘用 mq-deadline【单队列为 deadline】，SSD 用 none【单多列为 noop】，可以将 echo "deadline" > /sys/block/sda/queue/scheduler 加入/etc/rc.local 以开机自动设定调度策略。

#查看调度算法:

```
cat /sys/block/sda/queue/scheduler
```

#临时更改调度算法:

```
echo deadline >/sys/block/sda/queue/scheduler
```

#永久更改调度算法:

修改/etc/grub.conf

kernel 的最后加上 elevator=deadline

- 考虑使用带电池的硬件 raid 卡。阵列写策略为 WB，关闭预读，勾选 Force WB with no battery，关闭磁盘本身 Cache。



① 说明:

截图来自于某服务器厂商网站。

如果 BIOS 里面没设置，可以使用如下命令设置（以 DELL 服务器为例）：

关闭预读命令:

MegaCli64 -LDSetProp NORA -LALL -aALL

设置 cache 在电池充放电时有效:

MegaCli64 -LDSetProp CachedBadBBU -lall -a0

关闭磁盘本身 Cache:

MegaCli64 -LDSetProp -DisDskCache -Lall -aALL

中断

- 在常见中断中，大量的网络收发可能需要消耗较多 CPU，为了避免网卡中断占满单个 CPU 核心，通常使用多队列网卡，每个队列绑定一个 CPU。

网络环境

- 单中心内部 ping ip 最大延时要求小于 0.5ms，同城跨中心强同步 ping ip 最大延时要求小于 5ms。

- 交换机端口速率大于主机网卡速率或万兆，总带宽不小于所接服务器网卡带宽和的一半。

其他

- 注意跨机架和跨机房分配服务器，至少 DB 和 PROXY 机器要跨机架。
- 有条件可以使用 UPS 等尽可能保证机房供电正常。

操作系统配置优化

系统选择

建议最小化安装，X86 推荐使用 CentOS 分支（redhat）7.8 及其以上版本，ARM 要求银河麒麟 V10。

软件源

所有服务器需配置好对应系统的完整 yum/apt 源，必要时可以方便地安装工具。可以使用 `yum -y install iotop` 等命令检查 yum 的有效性。

时间与同步

部署 NTP 服务器，所有服务器连接 NTP 服务器，保证服务器间的时间误差不超过 3 秒。建议使用北京时间（UTC+08:00）。

资源限制类

tdsql 标准化安装已设定，无需手动设定，如下配置：

```
vi /etc/profile
# BEGIN TDSQL SET
ulimit -HSn 600000
export HISTSIZE=5000
export HISTTIMEFORMAT="%F %T \`who am i\` \"
umask 0022
# END TDSQL SET
vi /etc/security/limits.conf
# BEGIN TDSQL SET
* - nofile 1000000
# END TDSQL SET
#优化每个用户创建最大进程数
echo "* soft nproc 60000" > /etc/security/limits.d/20-nproc.conf
echo "root soft nproc unlimited" >> /etc/security/limits.d/20-nproc.conf
```

常用内核参数

tdsql 标准化安装已设定，无需手动设定，如下参数：

```
fs.file-max=6553500
vm.max_map_count=655360
net.ipv4.ip_local_port_range=32768 61000
```

```
kernel.pid_max=98304
kernel.threads-max=8241675
net.ipv4.tcp_tw_reuse=1
net.ipv4.tcp_window_scaling=1
net.ipv4.tcp_max_syn_backlog=4096
net.core.somaxconn=4096
net.core.netdev_max_backlog=2000
vm.swappiness=0
net.ipv4.tcp_keepalive_time=10
net.ipv4.tcp_keepalive_intvl=10
net.ipv4.tcp_keepalive_probes=9
net.ipv4.tcp_retries2=7
kernel.core_pattern=/data/coredump/core-%e-%p-%t
net.ipv4.conf.tunl0.arp_ignore=1
net.ipv4.conf.tunl0.arp_announce=2
net.ipv4.conf.all.arp_announce=2
net.ipv4.conf.tunl0.rp_filter=0
net.ipv4.conf.all.rp_filter=0
net.ipv4.ip_forward=0
fs.aio-max-nr = 1048576
```

2.2 数据库配置参数优化

限制类参数：**tdsql** 已默认选择通用优化值,通常不需要修改,如:

- DB

```
max_connections      = 10000
innodb_open_files    = 10240
open_files_limit     = 100000
max_prepared_stmt_count = 200000
reject_table_no_pk    = ON
```

- Proxy

```
<!--conn: 配置每个 IP 最多的连接数 0 不限制-->
<conn conn_limit="0"/>
<!--table: Groupshard 下最大表的个数限制, 默认为 1000-->
<table max="1000"/>
```

并发/内存/日志等参数：**tdsql** 已默认选择通用优化值,通常不需要修改,如:

```
lower_case_table_names    = 1
table_open_cache_instances = 32
table_open_cache          = 20480
binlog_write_threshold    = 1610612736
lock_wait_timeout         = 5
long_query_time           = 1
slow_query_log            = ON
max_allowed_packet        = 1073741824
innodb_flush_method       = O_DIRECT
innodb_thread_concurrency = 64
#部分参数根据环境已智能设置
thread_cache_size         = 8
innodb_io_capacity        = 10000
innodb_io_capacity_max    = 20000
innodb_flush_neighbors    = 0
innodb_buffer_pool_chunk_size = 32M
innodb_buffer_pool_size   = 2000M
innodb_buffer_pool_instances = 3
innodb_log_buffer_size    = 268435456
```

3 SQL 调优

3.1 了解执行计划

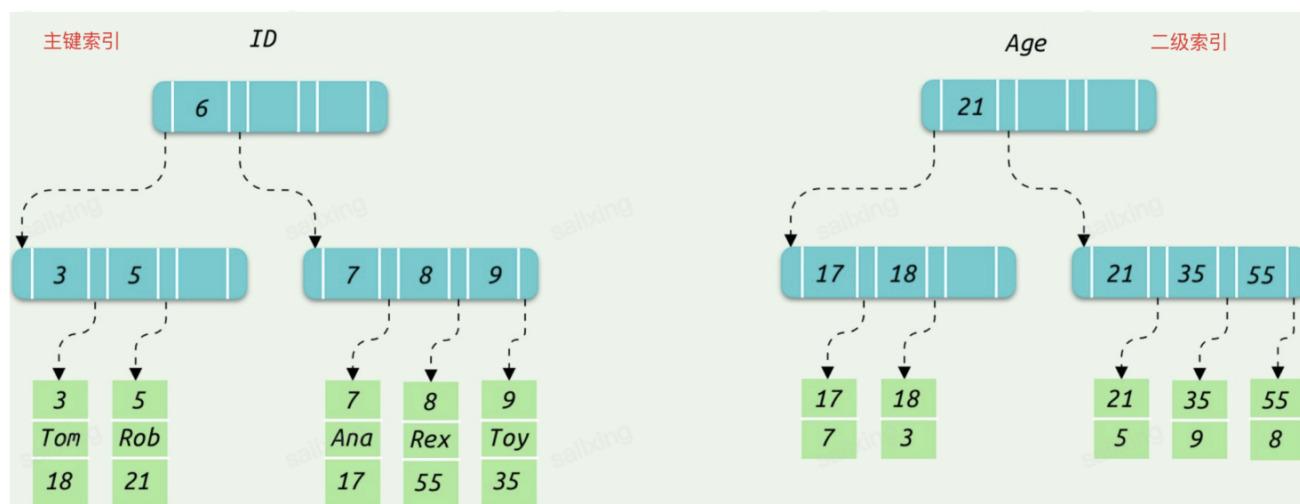
3.1.1 底层索引结构

逻辑结构示例

```
# 创建表
CREATE TABLE users(
id INT NOT NULL,
name VARCHAR(20) NOT NULL,
age INT NOT NULL,
PRIMARY KEY(id)
);
# 插入数据
INSERT INTO users(id,name,age) values(3,'Tom',18);
INSERT INTO users(id,name,age) values(5,'Rob',21);
INSERT INTO users(id,name,age) values(7,'Ana',17);
INSERT INTO users(id,name,age) values(8,'Rex',55);
INSERT INTO users(id,name,age) values(9,'Toy',35);
# 添加二级索引
ALTER TABLE users ADD INDEX index_age(age);
```

在数据库中主键索引的叶子节点存的是整行数据，而二级索引叶子节点内容是主键的值。

上述示例索引结构如下：



索引查询逻辑

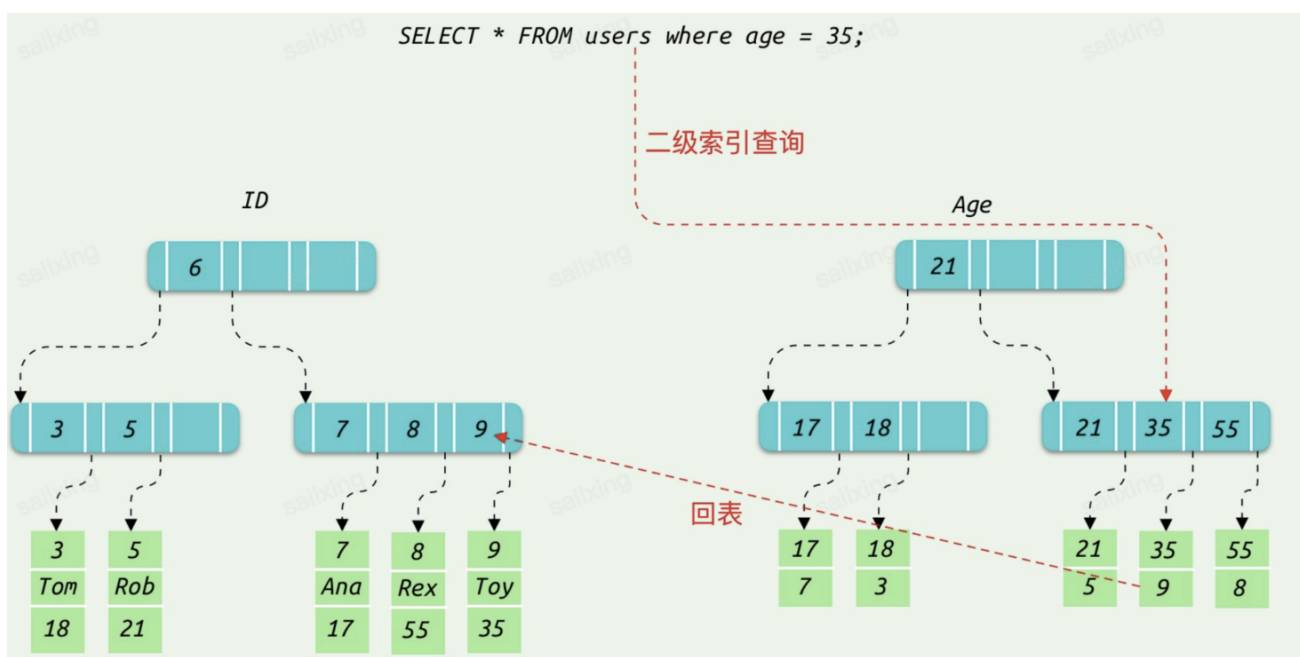
- 使用主键索引查询：

主键唯一，且只需要查找主键索引树。

- 使用二级索引查询：

先遍历二级索引，根据得到的主键值去回表查询；

先遍历二级索引，如果不需要除主键或当前索引列以外的其它列，则不需要回表（即覆盖索引）。



优化思路

基于非主键索引的查询需要多扫描一棵索引树，使用覆盖索引是一个常用的性能优化手段。

3.1.2 执行计划解读

查看 SQL 执行计划

语法: Explain + SQL

注意:

查看执行计划，SQL 不会真正执行。

在只读备机上，无法查看非只读类 SQL 的执行计划。

执行计划各个字段的含义

```
mysql> explain select * from nt3 join nt4 on nt3.a=nt4.b where nt4.b=1\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: nt3
    partitions: NULL
         type: const
possible_keys: PRIMARY
         key: PRIMARY
        key_len: 4
         ref: const
         rows: 1
    filtered: 100.00
      Extra: NULL
***** 2. row *****
      id: 1
    select_type: SIMPLE
        table: nt4
    partitions: NULL
         type: ALL
possible_keys: NULL
         key: NULL
        key_len: NULL
         ref: NULL
         rows: 2
    filtered: 50.00
      Extra: Using where
2 rows in set (0.00 sec)
```

id: 操作表顺序

- id 不同，执行顺序从大到小。
- id 相同，执行顺序从上到下。

select_type: select 类型

- Simple: 简单查询，不包含子查询或 union。
- Primary: 最外层的查询
- Subquery: 子查询
- Union: union 之后的查询
- Dependent subquery: 依赖于外查询的子查询

table: 操作的表名

type: 找到对应行是数据扫描方式

- ALL: 全表扫描
- index: 遍历索引树
- range: 对索引树进行范围扫描
- ref: 使用非唯一索引或唯一索引前缀进行查询
- eq_ref: 多表连接中, 使用主键或唯一索引进行查询。
- const/system: 根据主键或唯一索引进行查询
- NULL: 不需要访问表结构或索引直接得到结果

key: 实际使用到的索引

ref: 使用某些列或常量来查找数据

rows: 扫描行的数量

Extra: 其它关键信息

- Using filesort: 在没有索引的列上进行排序
- Using index: 不需要回表
- Using where: 部分条件不在索引中
- Using temporary: 使用临时表来存储结果集, 常用于分组。

注意:

在分布式场景下的网关执行计划, 会多一列 Info, 记录了实际发往的 Set 名称和 SQL 信息, 如下图。

```
mysql> explain select * from t1 \G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: t1
    partitions: p4,p5,p6,p7
       type: ALL
possible_keys: NULL
        key: NULL
     key_len: NULL
         ref: NULL
        rows: 3
   filtered: 100.00
      Extra: NULL
info: set_1576218740_3,explain select * from `asia`.`t1`
***** 2. row *****
      id: 1
    select_type: SIMPLE
      table: t1
    partitions: p0,p1,p2,p3
       type: ALL
possible_keys: NULL
        key: NULL
     key_len: NULL
         ref: NULL
        rows: 1
   filtered: 100.00
      Extra: NULL
info: set_1576218438_1,explain select * from `asia`.`t1`
2 rows in set (0.00 sec)
```

TDSQL 专有的 Proxy 下推

网关下推是 TDSQL 在分布式场景下对 SQL 处理的优化，是将 SQL 进行拆分下发到不同 DB 的操作；

通过下推 SQL，网关只需要将各个 Set 的返回结果进行聚合。

为什么要进行下推

1. 在分布式场景下，shard 表的数据是分布在不同 DB 上的，用户的一条请求在网关这里可能是对多个 DB 的操作；
2. 减少访问不必要的 Set；
3. 复杂操作下推到 DB，网关只需要负责收发聚合操作；
4. 减少不必要的数据库拉取。

网关常见的下推场景

1. 单表查询

对于指定 shardkey 的单表查询，直接下推到目标 Set。

```
mysql> explain select * From t1 where a=1 \G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: NULL
  partitions: NULL
        type: NULL
possible_keys: NULL
          key: NULL
        key_len: NULL
          ref: NULL
         rows: NULL
       filtered: NULL
      Extra: no matching row in const table
      info: set_1576218438_1,explain select * from `asia`.`t1` where (a = 1)
1 row in set (0.00 sec)
```

对于未指定 shardkey 的单表查询，广播到所有 Set。

```
mysql> explain select * From t1 where b=1 \G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t1
  partitions: p4,p5,p6,p7
        type: ALL
possible_keys: NULL
          key: NULL
        key_len: NULL
          ref: NULL
         rows: 1
       filtered: 100.00
      Extra: Using where
      info: set_1576218740_3,explain select * from `asia`.`t1` where (b = 1)
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: t1
  partitions: p0,p1,p2,p3
        type: ALL
possible_keys: NULL
          key: NULL
        key_len: NULL
          ref: NULL
         rows: 1
       filtered: 100.00
      Extra: Using where
      info: set_1576218438_1,explain select * from `asia`.`t1` where (b = 1)
2 rows in set (0.00 sec)
```

2. shardkey 相等的表连接

- 对于 shardkey 相等的表连接操作，下推到所有 Set 运行：explain select * From t1, t2 where t1.a=t2.a;
- 对于 shardkey 相等且 shardkey 指定明确值的条件下，下推到指定 Set 运行：explain select * from t1, t2 where t1.a=t2.a and t1.a=1;
- 对于 shardkey 相等且 shardkey 指定明确值，但是使用了 or 条件的情况下，下推到所有 set 运行：explain select * from t1, t2 where t1.a=t2.a and (t1.a=1 or t1.a=2);
- 对于没有 shardkey 相等的多表连接操作，如果没有其它条件，则无法下推，走嵌入式。explain select * From t1 where t1.a > (select min(a) from t2)----- 拉取 t1 的数据，计算子查询得到 t2 的一条数据，之后在本地做 join；explain select * From t1 join t2 on t1.a=t2.b---- -- 拉取 t1 的数据，然后根据 t1 的数据作为条件去拉取 t2 的数据。

3. 子查询

子查询如可以通过等值传递，判断出父子查询的 shardkey 相等时，则下推到所有 Set:

explain select * from t1 where a in (select a from t2)

```
mysql> explain select * from t1 where a in (select a from t2) \G
***** 1. row *****
trace: [
  {
    "DBQuery": "Select `t1`.`a`, `t1`.`b` from `asia`.`t1` where (`t1`.`a`) in (select `t2`.`a` from `t2`)",
    "ProxyDeduplicate": "false"
  }
]
1 row in set (0.00 sec)

mysql> explain select * from t1 where t1.a in (select b from t2 where t2.a=t2.b) \G
***** 1. row *****
trace: [
  {
    "DBQuery": "Select `t1`.`a`, `t1`.`b` from `asia`.`t1` where (`t1`.`a`) in (select `t2`.`b` from `t2` where (`t2`.`a` = `t2`.`b`))",
    "ProxyDeduplicate": "false"
  }
]
1 row in set (0.00 sec)
```

父子查询的 shardkey 不相等时，则无法下推:

explain select * from t1 where a in (select b from t2)

```
mysql> explain select * from t1 where a in (select b from t2) \G
***** 1. row *****
trace: [
  {
    "optype": "TableRename",
    "t1": "T1(a,b)",
    "t2": "T4(b)",
    "timecost": "0.022000"
  },
  {
    "0.OpType": "Load table",
    "1.TableName": "T1",
    "2.PushedDownCond": "(/*filter*/1)",
    "3.NumOfRows": ">=2",
    "timecost": "0.787000"
  },
  {
    "0.OpType": "Load table",
    "1.TableName": "T4",
    "2.PushedDownCond": "(1 and /*estimated filter: */(`b` in (2,4,3,1) or `b` is null))",
    "3.NumOfRows": ">=2",
    "timecost": "0.473000"
  },
  {
    "Query": "select `asia`.`t1`.`a`,`asia`.`t1`.`b` from `asia`.`T1` `t1` where (`asia`.`t1`.`a`) in (select `asia`.`t2`.`b` from `asia`.`T4` `t2` where 1)",
    "timecost": "0.001000"
  }
]
1 row in set (0.01 sec)
```

优化思路:

总而言之，网关下推是为了让查询的效率更高，为了将复杂查询下推，需要有 shardkey 相等的条件。

3.2 利用执行计划等优化 SQL

充分利用分片字段、二级分区字段

- 当 where/on/group by 等条件带上具体的分片字段、二级分区字段，SQL 能仅访问对应分片、二级分区的数据，最小化对资源的利用，同时也很有可能减小分布式事务的比例。
- 如对于 TDSQL 默认的 hash 分片而言，尽量使用确切的 where 分片字段条件，如部分范围查询 shardkey between 1 and 10 实际上可以转换成 IN 条件。

减少跨数据节点的 JOIN，尽量使 SQL 能下推到存储层

- 如果涉及到多表频繁联合查询，这些表最好使用相同的 `shardkey`，避免需要将数据加载到 `proxy` 层进行计算，减少数据访问的同时也可以提高效率。

示例

```
create table t1(a int key,b int) shardkey=a;
create table t2(a int key,b int) shardkey=a;
create table t3(a int key,b int) shardkey=a;
insert into t1(a,b) values(1,1),(2,2),(3,3),(4,4),(5,5);
insert into t2(a,b) values(1,1),(2,2),(3,3),(4,4),(5,5);
insert into t3(a,b) values(1,1),(2,2),(3,3),(4,4),(5,5);
```

对于 `shardkey` 相等的多表联合查询操作，下推到所有 `set`（存储层）运行

```
txsql> explain select * from t1,t2 where t1.a=t2.a;
```

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+
| id | select_type | table | partitions          | type | possible_keys | key  | key_len | ref  | rows
| filtered | Extra | info                  |
+---+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+
| 1 | SIMPLE      | t1    | p0,p1,p2,p3,p4,p5,p6,p7 | ALL  | PRIMARY      | NULL | NULL    | NULL | 100.00 |
| NULL   | 2 | 100.00 | NULL | set_1686300115_1, explain select * from `mydb`.`t1` join
`mydb`.`t2` where (t1.a = t2.a) |
| 1 | SIMPLE      | t1    | p8,p9,p10,p11,p12,p13,p14,p15 | ALL  | PRIMARY      | NULL | NULL    | NULL | 100.00 |
| NULL   | 3 | 100.00 | NULL | set_1686300187_3, explain select * from `mydb`.`t1` join
`mydb`.`t2` where (t1.a = t2.a) |
| 1 | SIMPLE      | t2    | p8,p9,p10,p11,p12,p13,p14,p15 | eq_ref | PRIMARY      | PRIMARY | 4
| mydb.t1.a | 1 | 100.00 | NULL | set_1686300187_3, explain select * from `mydb`.`t1` join
`mydb`.`t2` where (t1.a = t2.a) |
| 1 | SIMPLE      | t2    | p0,p1,p2,p3,p4,p5,p6,p7 | eq_ref | PRIMARY      | PRIMARY | 4
| mydb.t1.a | 1 | 100.00 | NULL | set_1686300115_1, explain select * from `mydb`.`t1` join
`mydb`.`t2` where (t1.a = t2.a) |
+---+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+
4 rows in set (0.01 sec)
```

对于 `shardkey` 相等且 `shardkey` 指定明确值的条件下，下推到指定 `set`（存储节点）运行

```
txsql> explain select * from t1,t2 where t1.a=t2.a and t1.a=1;
```

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+
| id | select_type | table | partitions          | type | possible_keys | key  | key_len | ref  | rows
| filtered | Extra | info                  |
+---+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+
4 rows in set (0.01 sec)
```

```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered |
Extra | info
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | p1 | const | PRIMARY | PRIMARY | 4 | const | 1 | 100.00 |
NULL | set_1686300115_1, explain select * from `mydb`.`t1` join `mydb`.`t2` where ((t1.a = t2.a)
and (t1.a = 1)) |
| 1 | SIMPLE | t2 | p1 | const | PRIMARY | PRIMARY | 4 | const | 1 | 100.00 |
NULL | set_1686300115_1, explain select * from `mydb`.`t1` join `mydb`.`t2` where ((t1.a = t2.a)
and (t1.a = 1)) |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

# 对于 shardkey 相等且 shardkey 指定明确值，但是使用了 or 条件的情况下，下推到所有 set
（存储节点）运行
txsql> explain select * from t1, t2 where t1.a=t2.a and (t1.a=1 or t1.a=2);
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered |
Extra | info
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | p1 | range | PRIMARY | PRIMARY | 4 | NULL | 2 |
100.00 | Using where | set_1686300115_1, explain select * from `mydb`.`t1` join `mydb`.`t2` where
((t1.a = t2.a) and ((t1.a = 1) or (t1.a = 2))) |
| 1 | SIMPLE | t1 | p12 | range | PRIMARY | PRIMARY | 4 | NULL | 2 |
100.00 | Using where | set_1686300187_3, explain select * from `mydb`.`t1` join `mydb`.`t2` where
((t1.a = t2.a) and ((t1.a = 1) or (t1.a = 2))) |
| 1 | SIMPLE | t2 | p12 | eq_ref | PRIMARY | PRIMARY | 4 | mydb.t1.a | 1 |
100.00 | NULL | set_1686300187_3, explain select * from `mydb`.`t1` join `mydb`.`t2` where
((t1.a = t2.a) and ((t1.a = 1) or (t1.a = 2))) |
| 1 | SIMPLE | t2 | p1 | eq_ref | PRIMARY | PRIMARY | 4 | mydb.t1.a | 1 |
100.00 | NULL | set_1686300115_1, explain select * from `mydb`.`t1` join `mydb`.`t2` where
((t1.a = t2.a) and ((t1.a = 1) or (t1.a = 2))) |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

对于没有 shardkey 相等的多表连接查询操作，如果没有其它条件，则无法下推，走嵌入式查询

```
txsql> explain select * from t1, t2 where t1.b=t2.b\G
```

```
***** 1. row *****
```

```
trace: [
```

```
{
  "optype" : "TableRename",
  "t1" : "T2(a,b)",
  "t2" : "T3(a,b)",
  "timecost" : "0.025000"
},
{
  "0.OpType" : "Load table",
  "1.TableName" : "T2",
  "2.PushedDownCond" : "(/*filter*/1)",
  "3.NumOfRows" : ">=2",
  "Query" : "AllSets , select `a`,`b` from `mydb`.`t1` `t1` where (/*filter*/1) limit 1000",
  "QueryMode" : "All",
  "timecost" : "0.829000"
},
{
  "0.OpType" : "Load table",
  "1.TableName" : "T3",
  "2.PushedDownCond" : "(/*filter*/1 and /*estimated filter: */(`b` in (1,5,2,4,3)))",
  "3.NumOfRows" : ">=2",
  "Query" : "AllSets , select `a`,`b` from `mydb`.`t2` `t2` where (/*filter*/1 and /*estimated filter: */(`b` in (1,5,2,4,3))) limit 1000",
  "QueryMode" : "All",
  "timecost" : "0.684000"
},
{
  "Query" : "select `mydb`.`t1`.`a`,`mydb`.`t1`.`b`,`mydb`.`t2`.`a`,`mydb`.`t2`.`b` from `mydb`.`T2` `t1` join `mydb`.`T3` `t2` where `mydb`.`t1`.`b`=`mydb`.`t2`.`b`",
  "timecost" : "0.002000"
}
]
```

```
1 row in set (0.00 sec)
```

避免全表扫描

最少使用一个左定索引访问数据

- 进行模糊匹配或者使用联合索引时，需要利用到最左前缀。

- 如 a 索引字段为字符串，以 a like '%sfs%' 为查询条件无法使用该索引，应当使用类似 a like 'sfsfs%' 等条件；再如 a,b 为联合索引，以 b='xxx' 为查询条件无法使用该索引，应当使用 a='xxx' and b like 'sfsfs%' 等条件。

示例

```
txsql> explain select * from t1 where b like '%a%';
```

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | NULL | ALL | NULL | NULL | NULL | NULL | 9 | 11.11 | Using where |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set, 1 warning (0.00 sec)
```

```
txsql> explain select * from t1 where b like 'a%';
```

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | NULL | range | idx_b | idx_b | 123 | NULL | 1 | 100.00 | Using index condition |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set, 1 warning (0.01 sec)
```

防止隐式转换

- 当操作符与不同类型的操作数一起使用时，会发生类型转换以使操作数兼容。
- 如 b 索引字段为字符类型，以 b=1 为查询条件时，无法利用索引；应当使用 b='1' 为条件；

示例

```
txsql> create table t1(id int primary key,a int,b varchar(30));
```

```
txsql> insert into t1(id,a,b)
```

```
values(1,1,'1'),(2,2,'2'),(3,3,'3'),(4,4,'4'),(5,5,'5'),(6,6,'6'),(7,7,'7'),(8,8,'8'),(9,9,'9');
```

```
txsql> alter table t1 add index idx_b(b);
```

字段是字符串类型，查询使用 int 型，无法正常使用索引

```
txsql> explain select a from t1 where b=1\G
```



```
***** 1. row *****
```

```
id: 1
```

```
select_type: SIMPLE
```

```
table: t1
```

```
partitions: NULL
```

```
type: ALL
```

```
possible_keys: idx_b
```

```
key: NULL
```

```
key_len: NULL
```

```
ref: NULL
```

```
rows: 9
```

```
filtered: 11.11
```

```
Extra: Using where
```

```
1 row in set, 3 warnings (0.00 sec)
```

字段是字符串类型，查询使用字符串类型，可以正常使用索引

```
txsql> explain select a from t1 where b='1\G
```

```
***** 1. row *****
```

```
id: 1
```

```
select_type: SIMPLE
```

```
table: t1
```

```
partitions: NULL
```

```
type: ref
```

```
possible_keys: idx_b
```

```
key: idx_b
```

```
key_len: 123
```

```
ref: const
```

```
rows: 1
```

```
filtered: 100.00
```

```
Extra: NULL
```

```
1 row in set, 1 warning (0.00 sec)
```

不要在索引列上加函数，应当把索引列单独放在条件的一边

- 等号左边包含表达式不能使用索引，索引列上有函数无法使用索引。
- 如 a 索引字段为 datetime，以 '2021-01-01' <= date_format(a,'%Y-%m-%d') and date_format(a,'%Y-%m-%d') < '2022-01-01' 为查询条件无法使用该索引；应当使用 '2021-01-01 00:00:00' <= a and a < '2022-01-01 00:00:00'；

示例

等号左边包含表达式不能使用索引

```
txsql> explain select * from t1 where a+1=2;
```

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
```

```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
|
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| 1 | SIMPLE | t1 | NULL | ALL | NULL | NULL | NULL | NULL | 9 | 100.00 |
Using where |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set, 1 warning (0.00 sec)

txsql> explain select * from t1 where a=2-1;
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
--+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
|
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
--+
| 1 | SIMPLE | t1 | NULL | ref | idx_a | idx_a | 5 | const | 1 | 100.00 | NULL |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
--+
1 row in set, 1 warning (0.00 sec)
# 索引列上有函数无法使用索引
txsql> explain select * from t1 where power(b,2)=4;
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
|
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| 1 | SIMPLE | t1 | NULL | ALL | NULL | NULL | NULL | NULL | 9 | 100.00 |
Using where |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set, 1 warning (0.00 sec)

```

in 条件的值不应该太多，否则会使用全表扫描的方式

- 如避免 where a in (大量值>1000 个)等用法；

分页查询优化

- 对于翻页类需求，尽量使数据库只返回需要返回的结果集，一般不建议数据全返回后再在应用层排序筛选；

- 可以将多次翻页合并成一次翻页查询缓存在应用中，尤其是深度翻页时，尽可能每次缓存更多结果集；
- 对于数据库的分页，注意使用常用的分页 SQL 改写，如记录每次 limit 的最大最小唯一值，下次查询从该行标识作为与 limit 配合的 where 条件，避免深度分页性能差；再如先用覆盖索引等方式取出需要的行标识，然后再用 IN 条件查询需要的页。

3.3 表结构、表空间、DDL 优化

选择最优的表类型和分片方式

表类型选择

- 数据量大或读写压力大的表，应该选用分片表。
- 数据量不大的表，如果读多写少，应该使用广播表。
- 如果写也不算少，但需要和分片表频繁多行 join 的，也可以考虑使用广播表。
- 不需要和分片表进行 join 的中小表，写也比较多，应该选用 noshard 表。

分片字段选择：

- 分片字段必须包含在所有唯一约束内。
- 分片字段应优先选择不同值超过分片数 2 倍的长度较小的整数字段或 char/varchar 字符字段。
- 分片字段应优先选择只含有字母和数字的字段。
- 分片字段应优先选择经常出现在 where、on 条件中的字段。

分片函数选择：

- 通常使用 hash 列散函数作为分片函数，这种情况下负载均衡较为优良，也适合于绝大多数场景。
- range/list 分区在部分场景有其优势，也可以考虑合理应用。

合理规划二级分区字段和类型

二级分区字段选择：

- 分片字段应优先选择经常出现在 where、on 条件中的字段。
- 分片字段应优先选择长度较小的字段。

二级分区函数选择：

- 二级分区函数不能使用 hash。【非分片表的 partition 分区完全兼容 MySQL】
- 二级分区 range 方式常用于时间字段，用于日志相关记录表，将冷热数据隔离，避免大量历史数据影响仅涉及最近数据的事务。
- 二级分区 list 方式常用于不同值较少的类型字段，在合适的 where 条件下如此分区可以避免扫描无关类型的数据。

创建适当的索引

创建利用率高的索引，删除利用率不高的索引，注意联合索引的顺序

- 如果查询条件包含索引的左侧定值，则通常索引可以被用来快速完成数据行的定位。不过以下几种情况基本无法利用充分利用索引，应该规避此类辅助索引的创建：
 - 表行数只有几行。
 - Cardinality 低，即不同值非常少的列，如类型字段不应该创建索引。

删除冗余索引

- 冗余索引有害无利，建议删除。

表存储引擎、字符集等的选择

- 如无特殊需求，统一使用 INNODB 存储引擎。
- 库表字符集建议使用 UTF8MB4，应用不设置字符集或设置为 UTF8MB4。

ER 模型设计相关

- 对于基本不查询的大字段，可以将其单独放在一个另一个表中。

尽量使用业务逻辑作为主键而非自增键

- 尽量减少自增列和序列，避免分布式实现的自增列、序列增加网络交互次数，影响 SQL 响应时间，导致自增或序列成为整个系统资源。

DDL 优化

- 如果要对一个表进行多个 DDL，尽量将其合并成一个 DDL 语句，然后在业务低峰期使用在线 DDL 功能完成。

定期整理表空间碎片

- 在业务低峰期，对 delete 和主键 update 比较多的表进行 optimize 操作，减小表空间大小，提高数据行的扫描效率。

3.4 合理规划 SQL 大小与事务大小，优化应

用与数据库的交互

避免大事务

- 大事务危害：容易导致主从延时，需要较多的磁盘 IO 引起磁盘和网络 IO 峰值，导致其提交时因为刷盘、等待备库返回 BINLOG ACK 耗时较久而阻塞整个实例其他事务提交。
- 对于单个 SQL 可能导致的大事务，一般使用适当的索引条件分多次完成。

合并小事务

- 通常为了数据安全，一次组提交事务提交前，都需要刷盘一次，大量小事务导致较高的磁盘 IOPS。因此，如果可以将小事务合并成稍大的事务，将能提升整体性能。

避免长事务

- 长事务导致数据库快照逻辑存在额外消耗，尤其是 RR 隔离级别下还导致该事务持续期间，其他事务的 UNDO 无法释放。
- 长读写事务持有锁过程，导致锁等待，增加死锁几率。

使用 MULTIQUERY

- 部分场景可以一次性执行多条 SQL，这种情况下可以使用 MULTIQUERY 方式，一次数据库交互就发送多条 SQL 语句，避免会话串行执行 SQL 多次网络交互而响应时间较长。

使用 VALUES 多行插入

- 合理控制批量插入逻辑，寻找一个适合环境的批大小完成批量插入，减小网络交互，提高插入速度：如应用需要批量插入大量数据，应当合理拆分事务，并且使用 JDBC 的 BATCH 提交处理方式。

使用 IN 条件合并多个更新

- 对于多个类似的 UPDATE、DELETE 等，可以使用 IN 条件用一条 SQL 完成多行记录的变更。当然该方式也需要考虑是否会引入锁等待、死锁场景。

3.5 减小对数据库的总读写量、计算/存储压力

减小查询结果集列数

- 如果访问的所有字段刚好在一个索引里面，则数据库会使用纯索引访问提高性能。
- 所以应该严格遵循开发规范，避免使用 SELECT * 等方式取用不需要的数据。

减小查询结果集行数

- 如果可行，限制返回行数。如用户只能查询最近的 100 条交易记录，或者只能查询最近 3 个月的交易记录。
- 使用数据库端分页而不是全返回应用后只保留需要的数据。

减小数据库计算压力

- 如果可以，把排序操作放到应用层完成；如不是查询大量结果却只选择少量结果的场景，可以将排序过程上移到应用层来做。比较常见的如 union 语法应该尽量使用 union all 方式。
- 复杂运算上移至应用：一般认为一秒钟 CPU 只能做 10 万次以内的运算为复杂运算，如含小数的对数及指数运算、三角函数、3DES 及 BASE64 数据加密算法等等。这些复杂函数的大量运算应该上移到客户端，在客户端处理。另外，即便是 now() 这些小函数，也应该尽量在应用端填充完成。

减小数据库存储压力

- 设计表结构时选择够用的最小字段类型和长度：如使用无符号整数等替代字符串存储（类型等使用 TINYINT 替代、IP 使用无符号整形替代等）。
- 在数据库中仅存储应用端压缩过的数据。

3.6 优化业务逻辑，提高应用并行能力

- 分布式数据库的网络交互多于集中式，这可能会使单个事务的响应时间增加，为了提高业务整体吞吐量，应该提高应用并行度，此时通常需要增加连接池上限，并优化业务处理逻辑：如一些跑批任务，可以增加并行度，提高数据库资源利用率尽快完成跑批任务。
- 减少热点行更新场景：热点行更新事务持续时间长，一方面导致锁等待，另一方面导致死锁检测工作量大。热点行更新场景应该尽量使用缓存结合数据库持久化的方式。
- 避免锁等待以及死锁：事务当以相同顺序持有锁；核心交易类事务短小化；给锁定类 SQL 合适的索引查询条件。

3.7 架构与读写分离

尽量不在主库执行统计分析汇总或全表扫描抽取等慢 SQL

- 部分慢 SQL 如果放在主库执行，很可能影响核心事务的性能。此时，应该将合适的只读事务放到备库执行。

只读业务建议使用读写分离账号在从库中执行

- 创建只读账号并配置读写分离策略，通过修改业务读写数据库模型，将读请求打入从库中执行。

合理扩容分片、扩容资源，保持资源利用率在 60% 以下

- 资源利用率高时，故障的风险增加。此外，资源利用率高不可避免导致资源争用的现象，表现为事务响应时间增速相比整体吞吐量增速大得多。故此，在资源利用率高时，应该合理利用分布式数据库的优势，以增加服务器资源等方式减小单个服务器的资源利用率。