阿里云

藏经阁

阿里云开发者电子书系列

深入MySQL实战

快速理解MySQL核心技术

业内资深大咖联合出品，详细解读AliSQL
在双11等高并发场景下的应用与实践

深入MySQL实战

MySQL高可用 MGR8.0 最佳实践

作者：张彦东

MGR特性

（一）MGR是什么

1. MGR的定义

MGR是具备强大的分布式协调能力，可用于创建弹性、高可用性、高容错的复制拓扑的一个MySQL插件。

2. 通讯协议

基于Paxos算法的GCS原子广播协议，保证了一条事务在集群内要么在全部节点上提交，要么全部回滚。

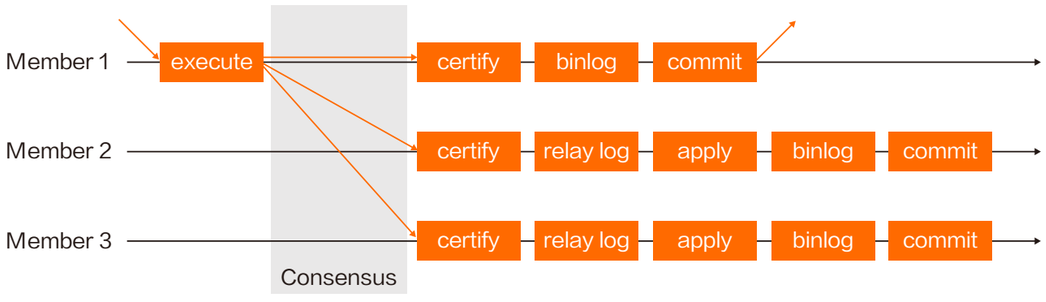
3. 组成员资格

MGR内部提供一个视图服务，集群节点之间相互交换各自的视图信息， 从而且实现集群整体的稳态。

4. 数据一致性

MGR内部实现了一套不同事务之间修改数据的冲突认证检测机制。在集群的所有节点当中进行一个冲突认证检测，反之，通过冲突认证检测的事务即可提交成功。

（二）示例



上图是一个三节点的MGR实例集群，Member1代表Primary节点，Member2、Member3代表Secondary节点。

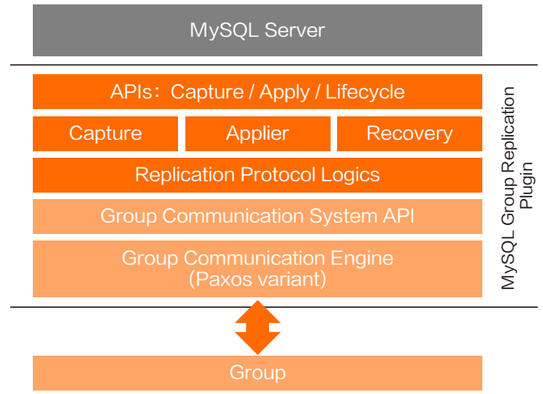
当一个事务发起提交后，它会通过原子广播协议分发到集群其他Secondary节点。集群的Secondary节点通过冲突检测之后，事务提交成功。在大多数的Secondary节点提交成功之后，会在Primary节点进行提交。

反之，如果在冲突认证检测失败，Secondary节点会丢弃这段事务对应的Binlog，Primary节点回滚该事务。

集群架构

MGR架构分为单主模式和多主模式。

（一）MGR插件组成



如上图所示，MGR插件使用MySQL Server与API接口层以及若干组件，最后由GCS（Group Communication System）协议封装而成。

MySQL Server调用MGR插件是基于MySQL现有的主从复制，利用Row格式的Binlog和Gtid等功能实现的集群架构。

API接口层复制基于MySQL Server交互的接口集，在逻辑上将MySQL内核与MGR插件隔绝开来。其他组件例如Capture组件，它是负责事务状态在集群内提交或是回滚，以及通过Binlog event广播到其他节点上进行的冲突认证检测进行到哪个阶段。Apply组件代表MGR集群Secondary节点Binlog回放，Recovery组件代表进行崩溃恢复或集群扩容时增量数据的应用。

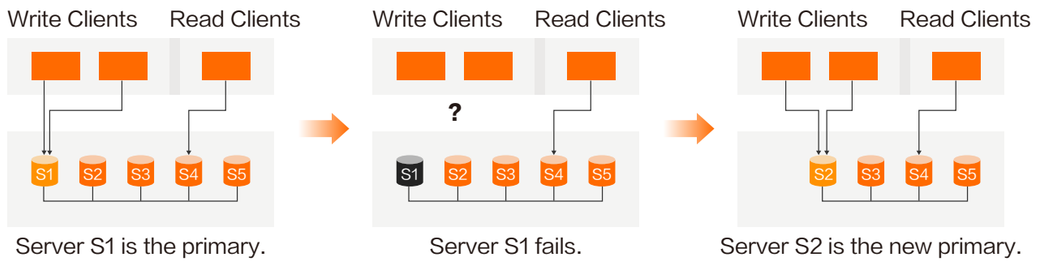
（二）单主模式

ON表示单主模式，也是默认模式，OFF表示多主模式。

如下图所示，在单主模式下（group_replication_single_primary_mode = ON）：

1. 该变量在所有组成员中必须设置为相同的值，同一个组中，不能将成员部署在不同模式中。例如，一个成员配置为单主模式，另一个成员配置为多主模式。

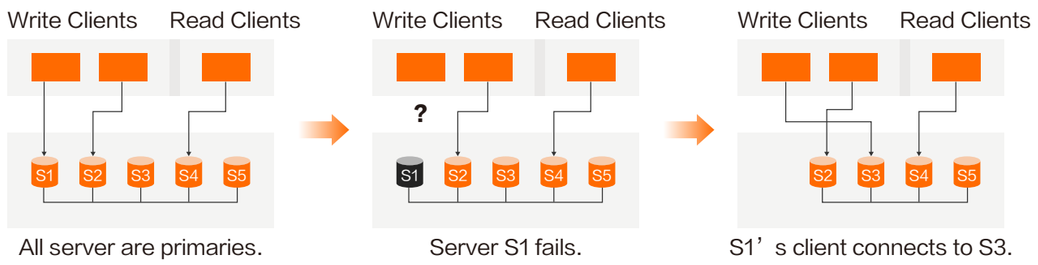
2. 该集群具有一个设置为读写模式的主节点，组中的所有其他成员都设置为只读模式（superread-only = ON）；



单主模式集群原理流程图

3.读写节点通常是引导该组的第一个节点，加入该集群的所有其他只读节点均需要从读写节点同步数据，并自动设置为只读模式。

（三）多主模式



多主模式集群原理流程图

在多主模式下（group_replication_single_primary_mode = OFF）：

1.所有节点不会区分Primary和Standby角色；

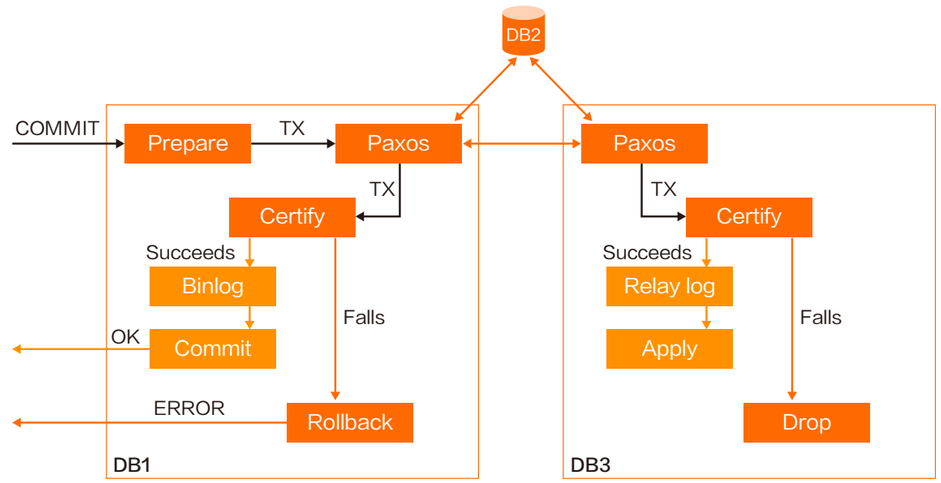
2.加入该集群时，与其他组成员兼容的任何节点都被设置为读写模式，并且可以处理写请求，即使它们在集群内是并发执行的；

3.如果组复制中的某个节点停止接受写事务，例如，在某个节点意外宕机的情况下，可以将与其连接的客户端重定向或故障转移到处于读写模式的任何其他健康的节点；

4.组复制本身不处理客户端故障转移，因此需要使用中间件框架（例如MySQL Router 8.0代理，连接器或应用程序本身）来实现。

数据同步原理

（一）同步原理示例

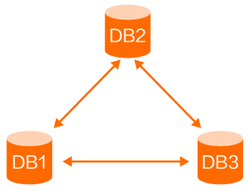


同步原理示例

如上图所示，以一个三节点的MGR集群为例。在单主模式下，当一个事务发起提交，它会通过原子广播协议将事务伴随着Binlog Event广播到其他Secondary节点上。在获得集群大多数节点同意之后，它会进行一个提交。如果通过冲突认证检测，那么该事务最终会在集群当中提交。

如果在Secondary节点上面没有通过冲突认证检测，那么Secondary节点丢弃该事务对应的Binlog，Primary节点回滚该事务。

（二）冲突检测原理



PK HASH	GTID SET
db1.tb1.pk=1	group_name:1-10
db1.tb1.pk=2	group_name:1-15
.....	.....

冲突检测原理

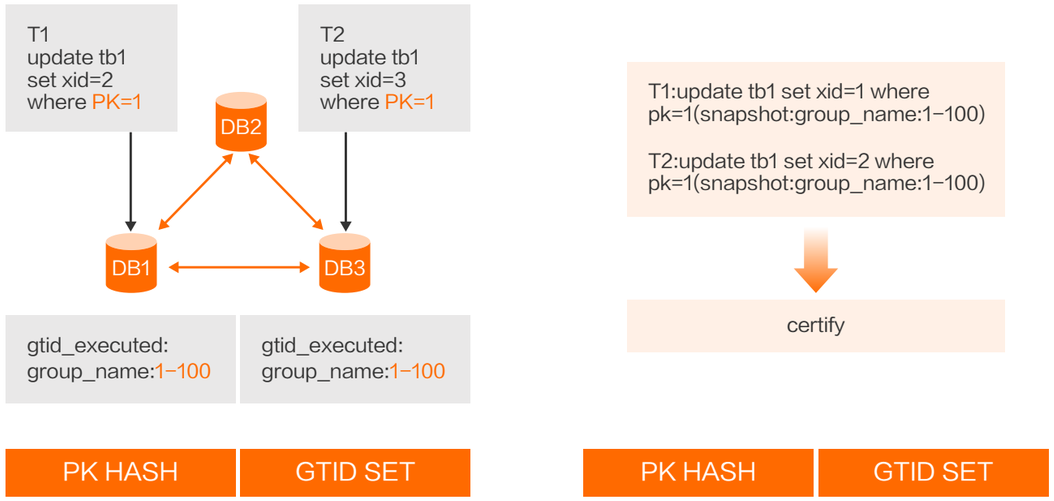
如上图所示，在冲突检测时：

1.每个事务的Gtid Set和对应的主键Hash值组成事务认证列表，在每个节点的内存当中都维护这样一个冲突检测库。

2.Gtid set：标记数据库的快照版本，事务提交前从gtid_execute变量中获取该值；

3.事务提交前，数据库中执行了的Gtid集合，随着Binlog中的Event通过原子广播的方式分发到集群的所有节点上进行事务冲突检测。

（三）冲突检测示例



如上图所示，以T1与T2这两条Update语句为例。

若T2修改的数据在冲突检测数据库中无匹配记录，则判定为通过冲突检测认证；

若T2中的GTID SET包含了冲突检测数据库中相同主键值的GTID SET，则冲突认证检测通过；

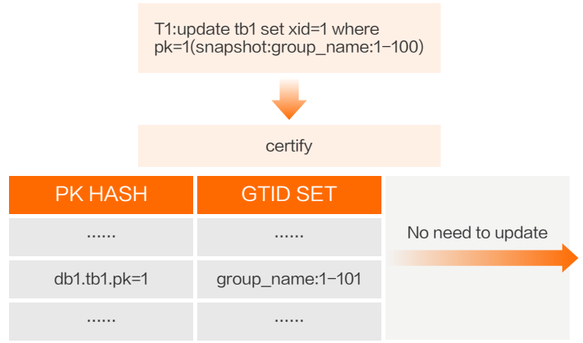
冲突认证检测通过后，每个节点的冲突检测数据库按照如下规则变更：

1.若在冲突检测数据库中无匹配记录，则向其中插入一条新的记录。

2.如果有记录，则更新冲突检测数据库中的GTID SET值。

若T1修改的数据在冲突检测数据库中有匹配到记录，且T1的GTID SET不包括冲突检测数据库中的GTID SET，则判定为冲突检测不通过。

注意：冲突检测不通过时，不会更新认证数据库里的GTID SET。



性能分析

（一）存在的问题

目前业内存在以下问题：

1.MGR可确保仅在集群中的大多数节点都已收到事务，并就并发发送的所有事务之间的相对顺序达成一致后才提交事务。相对顺序意味着，在分发到Secondary节点之后，可以不按照Primary节点提交的顺序进行提交，只需保证和集群的一致性即可。

2.在流量小的时候不存在任何的性能问题。当流量突增时，如果集群中某些节点的写入吞吐量比其他节点少，尤其是小于Primary节点，则这些节点的数据和Primary节点的数据存在偏差。例如说在集群当中，一个3节点的集群，如果节点之间服务性能存在差异的话，则会存在性能问题。

3.在单主模式的集群中，如果发生故障转移，在新的Primary节点可以立刻接受写入请求的情况下，则存在集群内事务一致性的问题。

举例说明，当集群扩容时，例如由3节点集群扩容到5节点集群：

1.无论采用Clone的方式还是用Xtrabackup做全量数据恢复后，都避免不了在集群扩容期间产生的增量数据以二进制日志的方式来追平；

2.若新扩容的节点配置较低，写入吞吐差，则新加入的节点很有可能一直处于Recover的状态，该节点很难达到Online的状态。

（二）事务一致性保证

在进行高可用切换之后，存在事务一致性保证，这是由于Secondary节点和Primary节点存在追数据的过程，如果数据没有追平，那么业务数据可能会读到旧的数据，用户可以根据group_replication_consistency参数对应的可选值进行调整，总共有5个值如下：

1. EVENTUAL

开启该级别的事务（T2），事务执行前不会等待先序事务（T1）的回放完成，也不会影响后序事务等待该事务回放完成。

2.BEFORE

开启了该级别的事务（T2），在开始前首先要等待先序事务（T1）的回放完成，确保此事务将在最新的数据上执行。

3.AFTER

开启该级别的事务（T1），只有等该事务回放完成。其他后序事务（T2）才开始执行，这样所有后序事务都会读取包含其更改的数据库状态，而不管它们在哪个节点上执行。

4. BEFORE_AND_AFTER

开启该级别等事务（T2），需要等待前序事务的回放完成（T1）；

同时后序事务（T3）等待该事务的回放完成。

5. BEFORE_ON_PRIMARY_FAILOVER

在发生切换时，连到新主的事务会被阻塞，等待先序提交的事务回放完成；

这样确保在故障切换时客户端都能读取到主服务器上的最新数据，保证了一致性。

用户根据不同的级别，根据业务上是读多写少，还是写多读少，或者是读写均衡的请求，不同的场景选择不同的值即可。

（三）流控机制

1.流控机制的功能

性能的问题对应着解决方案，MGR的流控机制试图解决以下问题：

1）集群内节点之间不会相差太多的事务；

2）快速适应集群内不断变化的负载，例如集群内的写压力暴增或集群中增加更多的节点；

3）均分可用写容量到集群内的所有节点上；

4）避免减少吞吐量而造成资源浪费。

2.基本机制

流控机制存在两个基本机制：

1）监控集群内所有节点以收集有关吞吐量和队列大小的一些统计信息，以便对每个节点能承受的最大写入压力进行有根据的评估；

2）对集群中的所有节点的并发写能力时刻保持监控，一旦某节点的并发压力超过了集群中所有节点的平均写能力，就会对其执行流量控制。

3.基本队列

流控机制存在两个基本队列：认证队列和二进制日志应用队列。

当其中一个队列的大小超过用户定义的阈值时，就会触发流量控制机制。

对于流量控制配置：

首先，需要选择对谁配置流量控制，是对认证队列、还是针对应用队列、还是两者都需要配置流量控制。然后，对需要配置流量控制的对象（认证队列和应用队列）设置流量控制阈值。

4.流控过程

流控具体过程如下：

1）将根据上一阶段延迟的事务数量逐步减少10%，让触发限流机制的队列减小到限流机制被触发的阈值之内， 待到恢复后，为避免在队列大小超过阈值时出现吞吐量的陡增，在此之后，每个时间段的吞吐量只允许增长相同的10%；

2）当前的限流机制不会影响到触发限流机制阈值内的事务，但是会延迟应用超过阈值的事务，直到本监控周期结束；

3）如果触发节流机制的阈值设置的非常小，部分事务的延迟可能会接近一个完整的监控周期。

适用场景

在日常业务中，MGR高可用集群存在如下经典适用场景：

1.弹性复制

需要非常灵活的复制基础设施的环境，其中MySQL Server的数量必须动态增加或减少，并且在增加或减少Server的过程中，对业务的副作用尽可能少。例如，云数据库服务。

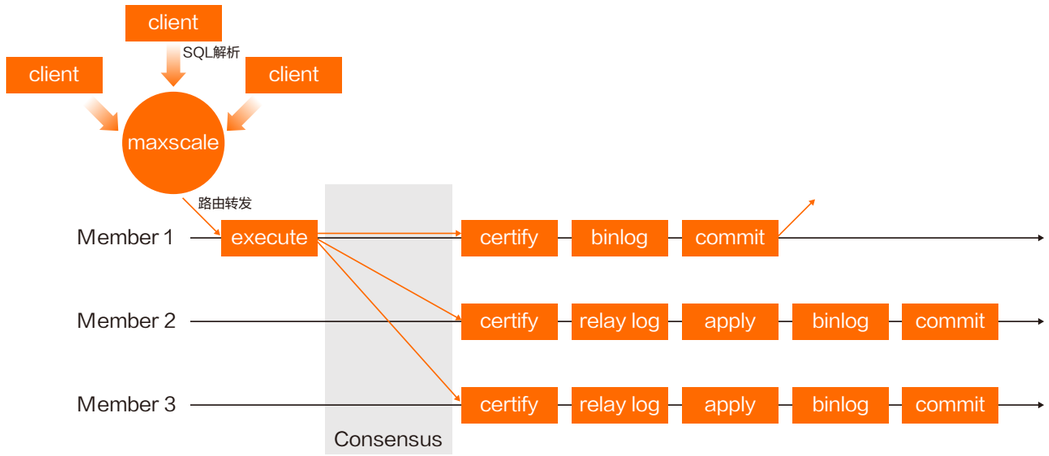
2.高可用分片

分片是实现写扩展的一种流行方法。基于MGR实现的高可用分片，其中每个分片都会映射到一个复制组上（逻辑上需要一一对应，但在物理上一个复制组可以承载多个分片）。

3.替代主从复制

在某些情况下，使用一个主库会造成单点争用。在某些情况下，向整个组内的多个成员同时写入数据，多种模式可以避免单点争用，对应用来说可能伸缩性更强。

高可用方案



上图为业内的一个常用的高可用方案。

通常在云数据库里，一个三节点的MGR集群本身不保证业务的写入重定向，那么在MGR集群上面加一个读写分离的中间件Maxscale。

将源代码重新编译之后，会打开一个它自带的保活机制，Maxscale会自动探测MGR集群里Primer节点状态，如果发生高可用切换或者是当前的Primary节点宕机之后，它会重新探测选取出来一个新的Primary节点，然后自动将业务写请求重定向到新的Primary节点上。业务只需要将Client端经过SQL解析连到Maxscale，然后经Maxscale做一个路由转发，即可实现一个灵活的高可用。

MySQL高并发场景实战

作者：凌洛

问题和挑战

阿里巴巴CEO逍遥子曾说过：“双11是商业界的奥林匹克。”

如上图所示，双十一购物始于2009年，历年的订单创建、支付笔数与交易总额都是成倍增长，这不仅带来许多商业机遇，也给后端技术、架构等各个模块带来技术沉淀。

双11为MySQL带来了高并发场景的问题与挑战，主要表现在：

1）洪峰般的并发

根据市场部部门的推广和引流、历年双11的经验，大促的起始时刻呈现接近90度上升趋势。在这么大访问流量下，所有的核心链路的增删改查都是在数据库上操作，对数据库有比较大的冲击，在大量线程并发工作时线程调度工作过多、大量缓存失效、资源竞争加剧、锁冲突严重，如果有复杂SQL或大事务的话还可能导致系统资源耗尽，整个数据库服务不可用，进而导致大促收到影响，甚至失败，比如：下单失败、网页无法打开、无法支付等。此外此类场景也会发生在在线教育、直播电商、在线协同办公等。

2）热点行更新

库存扣减场景是一个典型的热点问题，当多个用户去争抢扣减同一个商品的库存（对数据库来说，一个商品的库存就是数据库内的一行记录），数据库内对同一行的更新由行锁来控制并发。当单线程（排队）去更新一行记录时，性能非常高，但是当非常多的线程去并发更新一行记录时，整个数据库的性能会跌到趋近于零。

3）突发SQL访问

当缓存穿透或异常调用、有数据倾斜SQL、未创建索引SQL等情况发生时，在高并发场景下很容易导致数据库压力过大，响应过慢，导致应用链接释放慢，导致整个系统不可用。

4）智能化运维

双十一期间这些实例的水位管控，机器水位管控，高风险实例识别，高风险实例优化，在流量高峰期从收到报警、识别问题、解决问题至少需要十多分钟，如果处理不及时峰值已经过去，导致大促失败。

此外，结合业务还有商品超卖、资源的挑战等问题存在。

系统调优

为解决以上挑战与问题，我们需要做系统调优，主要从六个方面进行：容量评估、性能评测、架构调优、实例调优、内核调优和监控报警。

（一）容量评估

容量评估主要分为三个部分：经验评估、单元压测与全链路压测。

经验评估	单元压测	全链路压测
预估压力/单机性能=服务器数量	验证单元内容量	场景化压测
应用机器设置、数据库设置等	验证单个系统容量	解决分布式系统容量评估

经验评估

容量评估刚开始阶段是经验评估，根据以往经验值给出一个预估的压力，再除以单台机器的性能，大致可得出所需服务器的数量。根据服务器数量、应用机器数量与DB实例数量，可以得出整个数据库（如链接池）的设置，以及要扩充多少实例和应用机器等，判断能否支撑得住双11的峰值。需要针对上述的预估做一个判断验证，通过压测来完成。

单元压测

经过经验预估后，需要针对上述的预估做判断验证，可以通过单元压测来完成。由于是分布式的系统，需要预先针对单个实例、单个单元以及单个应用模块来进行压测。单元压测的主要功能是完成单元内的验证，系统整个的架构很多是异地多活，因此不但需要验证整个双11的流量，还要验证在单元内的容量是否充足，以及单个系统的容量，例如压测某一个模块、交易模块、优惠模块等容量是否充足。

全链路压测

等每一个应用模块验证完成之后，需要对整个链路进行压测，也就是全链路压测。全链路压测是基于场景化的仿真测试，其数据最接近业务的系统值，针对全链路压测，可以借助压测来验证整个分布式系统的容量是否充足。

（二）性能评测

1.性能评测的意义

1）目的

发现基础设施瓶颈、中间件瓶颈、系统容量瓶颈；

发现分布式系统短板。

2）用途

保障大促容量充足，保障业务正常运转；

保障核心功能、保证用户体验；

评估大促成本，包含人工成本、机器成本、运维成本等。

3）难点

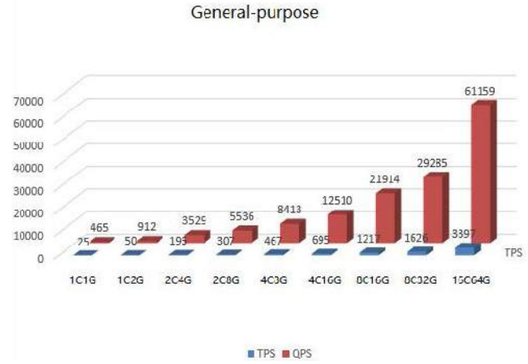
真实业务场景压测；

真实SQL模拟。

2.基准测试

```
sysbench
/usr/share/sysbench/oltp_read_write.lua --
mysql-host={host} --mysql-port={port} --
mysql-user={user} --mysql-password={passwd}
--mysql-db={db} --db-driver=mysql --
tables={count} --table-size={size} --report
interval={interval} --threads={threads} --
time={time} prepare/run/clean
```

此外还有mysqlslap，ab…



压测可以分为基准测试和仿真测试。对于MySQL而言，基准测试可以用sysbench或mysqlslap等。基于通用场景的测试，每个实例在不同的业务场景下达到的容量也不同，通常情况下，同一个实例的应用真实的业务场景的值不会超过其基准测试的值。

3. 全链路压测是大促备战核武器

真实业务场景的压测往往比基准测试更加复杂，阿里巴巴在双11的场景下整个压测过程大概可以分为4个步骤，如下图所示：

首先针对涉及到的业务进行模块梳理与技术架构梳理，以及容量预估是否充足。梳理完成之后需要准备测试环境，包含服务器准备、数据构造与业务请求准备等。第三步正式压测，可以通过压测来发现短板，验证容量是否充足。预案验证可以验证在某个压力场景下，降级某个预案会给系统减少多少压力。验证压测执行完之后，需要针对过程中发现的短板、架构以及容量等问题进行优化，这个过程并非一蹴而就，而是要经过多轮压测。

阿里巴巴集团及各BU每年压测4000+次，13年全链路发现700+问题，14年发现500+问题，15年发现400+问题。基于真实业务场景的全链路压测，已成为每年双11前筹备核心工作之一。

4.工具

对于上述的全链路压测操作，我们有一些现成的工具以供使用。

1）PTS（Performance Testing Service）

是面向所有技术背景人员的云化测试工具，可理解为全链路压测工具。有别于传统工具的繁复，PTS以互联网化的交互，提供性能测试、API调试和监测等多种能力。自研和适配开源的功能可以轻松模拟任意体量的用户访问业务的场景，任务随时发起，免去繁琐的搭建和维护成本。紧密结合监控、流控等兄弟产品提供一站式高可用能力，高效检验和管理业务性能。

2）DAS（Database Autonomy Service）

智能压测主要应用在以下两种场景：

为应对即将到来的短期业务高峰，验证当前的RDS MySQL规格是否需要扩容；

数据库迁移上云前，验证目标RDS MySQL的规格是否满足业务需求。

5.注意事项

根据以往的情况，性能评测时主要有4个方面需要特别注意：

1）参数

如果想对比一两个参数在不同情况下，例如打开和关闭、设置不同的值时对性能的影响，可以单独做测试。一旦测试出来最优值，需在压测之前将参数调到最优值。

2）网络

曾有用户拿线下的机器链接本地数据库，然后拿本地的应用机器链接RDS做性能对比，这种测试没有可对比性，主要原因是压测机到RDS的网络延迟和到本地数据库的延迟存在很大差异。

3）规格

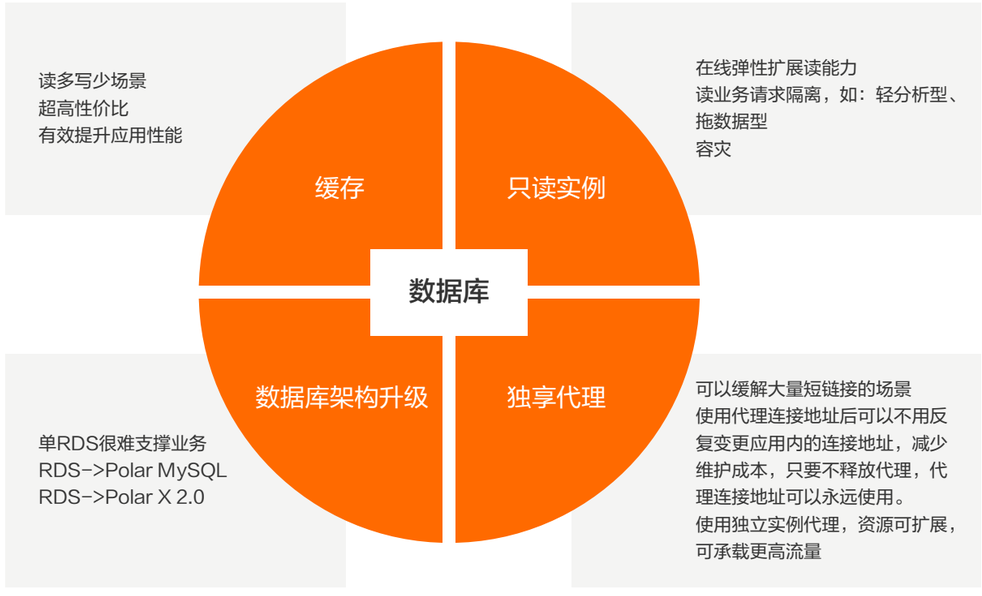
RDS的规格的性能差别较大,如果想测试CPU的性能，物理IO要少，数据大小在内存容量范围内即可。但大多数业务场景都是涉及到物理IO，所以在到测试数据时，数据量要大于内存的大小。

4）ECS的网络带宽

阿里云的ECS是限制网络带宽的，以往有用户在做测试时，RDS的资源没有用满，压力也上不去，经过定位发现是ECS的网络带宽打满了，因此在准备整个压测环境时，要将这些内容调好。

（三）架构调优

1.数据库架构调优



如果业务的访问都用数据库支撑的话成本高昂，缓存可以代替一部分关系型数据库在读方面的请求。基于原理的设计以及成本方面考虑，缓存的读性能比关系型数据库好，性价比较高。

到数据库层面，如果是读多写少，针对于单个实例很难支撑的情况下，可以借助于只读实例。只读实例可以实现在线弹性的扩展读能力，读的业务请求可以实现隔离，例如可以把轻分析型以及拖数据类型在只读实例内完成。

此外，每个只读实例都有一个单独的链接地址，如果把某一类的业务和其他的业务区分开，例如某一类的只读的这个场景，只到某一个实例访问，可以单独链接只读实例的链接串。

如果要是想从整个层面来控制主实例和只读实例的访问，可以借助负载均衡独享代理完成。独享代理可以缓解大量短链接

的场景，使用代理后不用反复变更应用类的链接地址，减少维护成本。使用独享代理之后，可以对线上的资源实现可扩展，承受更高的流量。如果是RDS的实例规格以及只读实例都已经升到最大，但仍然不能支撑业务发展的话，可以考虑把RDS的升级到Polar MySQL或者是分库分表PolarDB X 2.0，完成读写容量的扩展。

2. 降低只读实例和主实例的延迟

针对主实例和只读实例的数据一致性的问题，例如有延迟以及中断场景。针对线上延迟的问题我们做了分析，得出原因主要包括5个方面：

1）主实例的DDL

占40%，如Alter、Drop等。需要Kill DDL语句或用DMS的无锁变更等。

2）主实例的大事务

占20%，如大批量导入、删除、更新数据。需要将大事务拆分成为小事务进行批量提交，这样只读节点就可以迅速地完成事务的执行，不会造成数据的延迟。

3）主实例写入压力过大

占20%，如主实例规格较小、压力过大。这种情况需要升级主实例和只读实例的规格。

4）只读节点规格过小

占10%，这种情况升级只读实例规格。

5）其他（无主键）

占10%， RDS目前已经支持对表添加隐式主键，但是对于以前历史创建的表需要进行重建才能支持隐式主键。

3. 提升缓存命中率

在高并发场景下，写入压力过大，如何提升缓存的命中率?

根据以往的经验，有4种更新方式：Cache aside，Read through，Write through，Write behind caching。

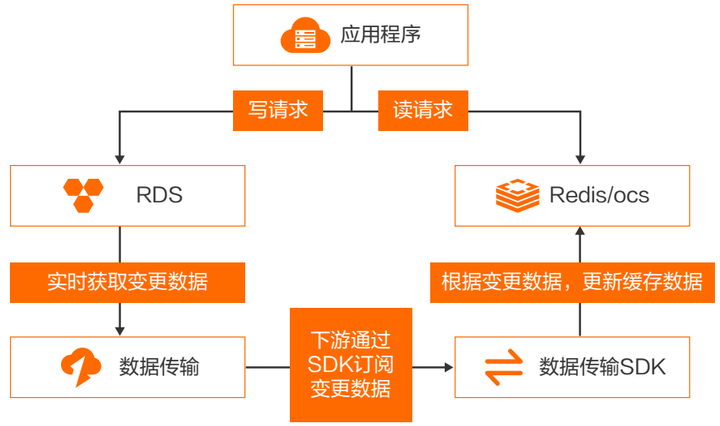
缓存的更新时，应用可以从Cache里面读取数据，取到后返回，没有得到从数据库里面取，成功后再返回缓存当中；

Read through是已读没有读到，就更新缓存；

Right through是在数据更新时，发现缓存中没有就更新缓存；

Write behind caching类似于底层的 Linux操作系统机制，如果要是用Write behind caching合并同一个数据的多次操作，以上几种方式都不能保证缓存命中率。

我们做核心系统时有一个小巧思，下面以产品的形式进行解释。



如上图所示，应用程序把写请求到 RDS,读请求到 Redis。RDS实时获取变更数据，通过订阅数据变更的方式拿到增量数据后，更新Redis，这个巧思有以下好处：

更新路径短，延迟低

缓存失效为异步流程，业务更新DB完成后直接返回，不需要关心缓存失效流程， 整个更新路径短 ，更新延迟低。

应用简单可靠

应用无需实现复杂双写逻辑，只需启动异步线程监听增量数据，更新缓存数据即可。

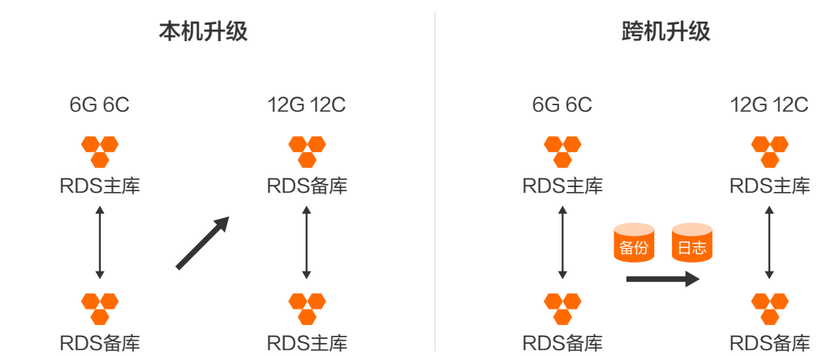
应用更新无额外性能消耗

因为数据订阅是通过解析DB的增量日志来获取增量数据，获取数据的过程对业务、DB性能无损。

（四）实例调优

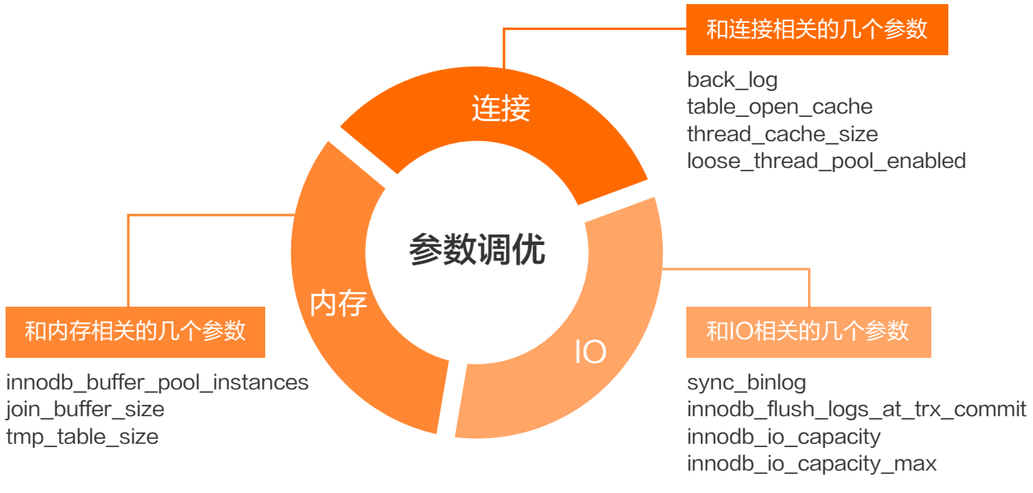
实例调优主要包含两部分：弹性扩容和参数调优。

1.弹性扩容



2.参数调优

参数调优主要从三个方面进行，分别问连接、内存和IO，如下图所示，这里重点阐述几个参数。



连接相关的参数，back_log值表示在MySQL暂时停止响应新请求之前的这段短时间内可以堆叠多少请求。如果在短时间内有大量的连接请求时，主线程会在一段时间内或者是瞬间检查连接数并启动新连接。

table_open_cache是所有线程打开表的数量，性能较小时会导致性能下降，但也不能设置过大，否则可能会造成OOM。

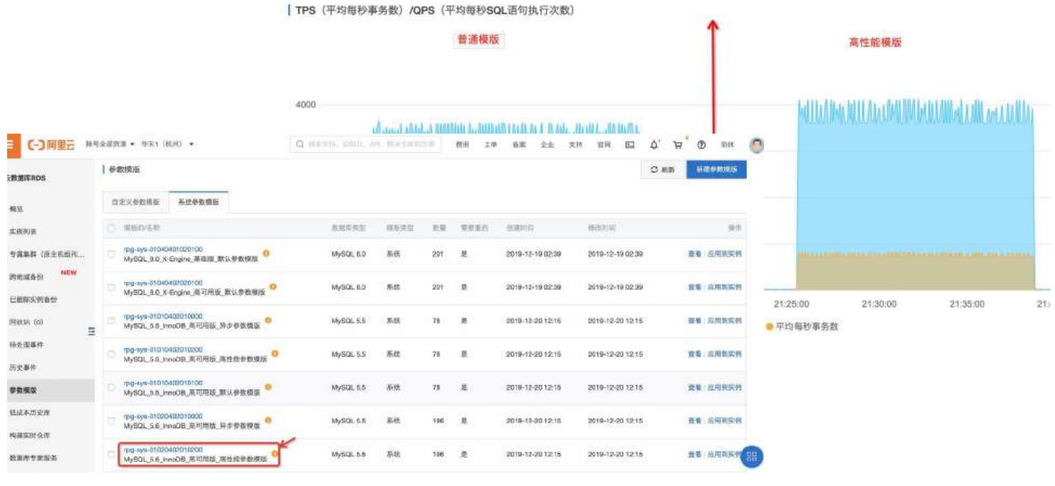
IO相关的参数，sync_binlog大家不会陌生，我们在每年的双11会将这些数值调到最优。io_capacity和刷脏是IO两个比较重要的参数。

内存相关的主要参数是innodb_buffer_pool_instances。在5.6之前，相对来说innodb_buffer_pool_instances只有一个时，如果内存越大，它的性能越不稳定，因为刷脏的时候要锁。当有buffer_pool_instances了之后，把 Buffer Pool 按buffer_pool_instances拆分。这种情况下，每一个内存都会维护自己的List和锁结构，性能会提升很多。此外还有和几个连接相关的参数，例如join_buffer以及临时表。

针对影响性能几个主要的参数，用户可以单独设置。这里我们已经将对性能影响较大的参数调到高性能模板中，应用时可以找到对应的参数，应用到对应的实例上即可。

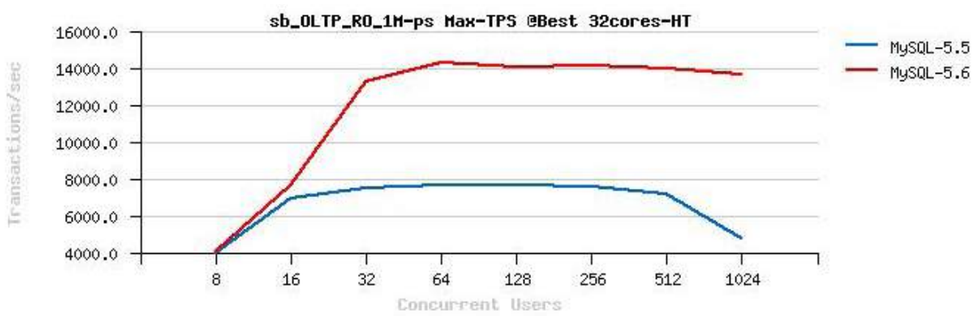
下图是普通模板与高性能模板的一个对比，可以看到，相比高性能模板，普通模板的性能要低25%左右。

路径：参数模板->系统参数模板->找到对应版本的参数模板->应用到实例



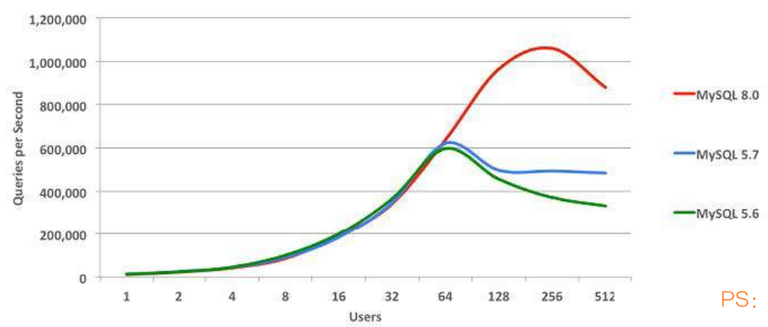
（五）内核调优

1.版本升级



MySQL 8.0: SysBench IO Bound Read Only (Point Selects)

2x Faster than MySQL 5.7



PS：升级有风险，需充分验证

如上图所示，目前官方维护的是5.6、5.7和8.0版本，每一个大版本的升级会带来一些新的特性，同时它的性能也会有上升，从表中可以看到8.0性能是最强的。但升级时也会遇到一些问题，例如执行计划有一些和原来版本不太兼容的地方。

2. AliSQL特性

1）内容调优的4个补丁

针对高并发与热点库存的场景，AliSQL提供了4个补丁解决，其中内存注释、语句队列以及语句返回这3个补丁是针对热点库存更新。

库存注释

库存注释

在秒杀等业务场景中，减库存是一个常见的高并发，串行化比较严重的模型，对数据库的吞吐能力产生巨大的挑战，AliSQL使用排队和事务性hint来控制并发控制和快速提交或者回滚事务，一方面控制并发，另外一方面快速结束事务释放事务锁，来提高减库存的吞吐能力。

在秒杀业务中，从逻辑上来说减库存包含两个场景：拍下减库存和付款减库存。

拍下减库存表示客户拍下时把库存减掉，业务逻辑较为简单。付款减库存表示客户付完款后减掉库存，业务逻辑较为复杂，不是一条语句能够完成的，而是在事务中完成。在事务中，行锁冲突本来就较为严重，因此需要提升事务的性能。

通常的情况下，释放事务、开启事务和结束事务都是由应用来完成，如果是应用处理得过慢，或是网络交互的时间过慢，或者是网络拥堵，就会增加行锁持有的时间。

库存注释是把持有行锁的时间行锁释放，交给数据库自己来做。使用排队和事务性Hint来控制并发控制、快速提交或者回滚事务，成功就提交，失败就回滚，将控制权交给MySQL内核可以减少行锁持有的时间，快速地结束事务，提升减库存的吞吐能力。

语句队列

语句队列

MySQL的server层和引擎层在statement并发执行过程中，有很多串行化的点，在DML语句中，事务锁冲突比较常见，InnoDB中事务锁的最细粒度是行级锁，如果语句针对相同行进行并发操作，会导致冲突比较严重，系统吞吐量会随着并发的增加而递减。

AliSQL设计了针对语句的排队机制，相同的行或者不同的语句进行分桶排队，尽可能的把具有相同冲突可能的在一个桶内排队，减少conflict的开销。

InnoDB的事务锁是最细粒度的行级锁，如果语句针对相同行进行并发操作，会导致冲突比较严重，AliSQL将冲突放在Server层进行排队。对于相同行的冲突，如果让它在Server层一个桶（队列）内排队，会减少InnoDB层冲突检测的开销，进而减少引擎层和Server层上下文切换带来的消耗。

语句返回

可以减少事务里多次语句交互的时长

MySQL对于statement执行结果报文通常分为两类Resultset和OK/ERR，针对DML语句则返回OK/ERR报文，其中包括几个影响记录，扫码记录等属性。

语句返回

语句返回即更新完之后直接把结果返回，这个特性在PG和Oracle中都有，叫Returning，在MySQL没有，AliSQL吸纳了这个特性。

如果在一个事务中更新完一行记录之后，应用再发请求Select这一条结果再返回给应用，会多一次网络交互。如果直接把Update的结果返回，就可以减少这一次网络交互。如果一个事务里面有多条类似的语句的话，可以节省多次网络交互和应用判断的时间，带来事务性能与应用性能的提升。

线程池

将线程和会话分离，在拥有大量会话的同时，只需要少量线程完成活跃会话的任务即可。

阿里云RDS的线程池实现了不同类型SQL操作的优先级及并发控制机制，将连接数始终控制在最佳连接数附近，使RDS数据库在高连接大并发情况下始终保持高性能。

线程池

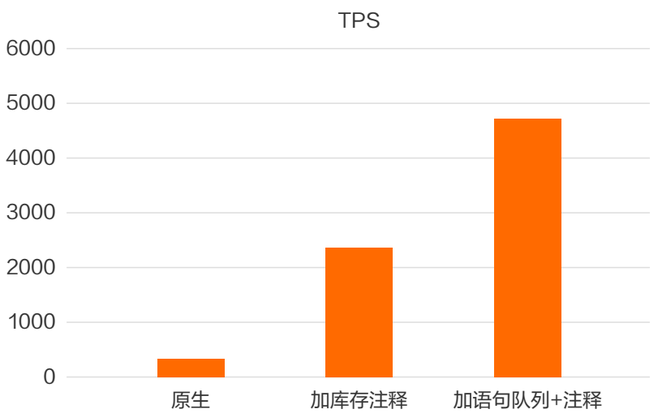
对于高并发场景，AliSQL使用线程池来解决。

在有大量线程进行并发访问时，线程池会自动调节并发的线程数量在合理范围内，避免线程池因为线程数量过多造成大量缓存失效。

线程池会将语句和事务分为不同的优先级，分别控制语句和事务的并发数量，减少资源竞争，根据不同语句的复杂性来控制它的优先级。

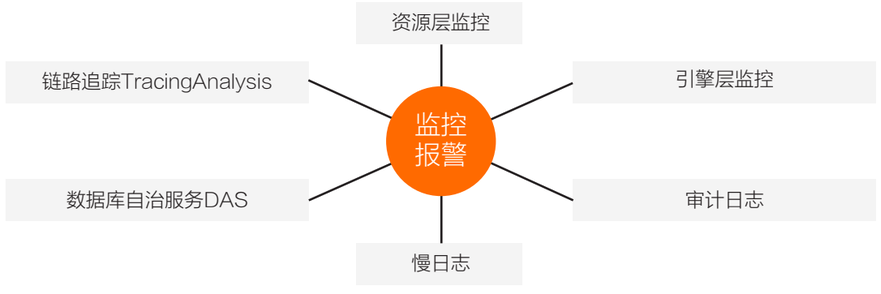
因此使用线程池可以将不同的SQL控制在一个合理的连接数范围，使数据库在高并发的场景下保持较高性能。

```
query = "UPDATE kucun SET c=c-1 WHERE id=1;"
db_query("BEGIN;")
rs = db_query(query)
db_query("COMMIT;")
```



上图是用库存注释与语句队列后的性能对比，可以看到和原生性能相比翻了40倍不止。

（六）监控报警



监控报警是系统调优里必不可少的一部分。

监控有RDS自带的监控，资源层监控，引擎层监控，慢日志的监控，收费的有审计日志，以及自治服务会基于上面RDS自带的以及审计日志做一些分析，在这个基础上做一些报警。此外，分布式监控可以监控不同组件的瓶颈。

流程管理和生态工具

基于生态工具的保障，可以从变更流程、稳定性和数据恢复这三方面来介绍。

（一）变更流程

在大促期间，通过增加审批节点与流程，很好地提升我们的稳定性。

增加审批节点

增加审批流程

可以通过设置DMS来执行的这些SQL，不让它超过一定的时间。也可以针对某一类的DDL或者是DML不允许执行，设置安全规则。

（二）稳定性相关

1.设置超时时间（通用）

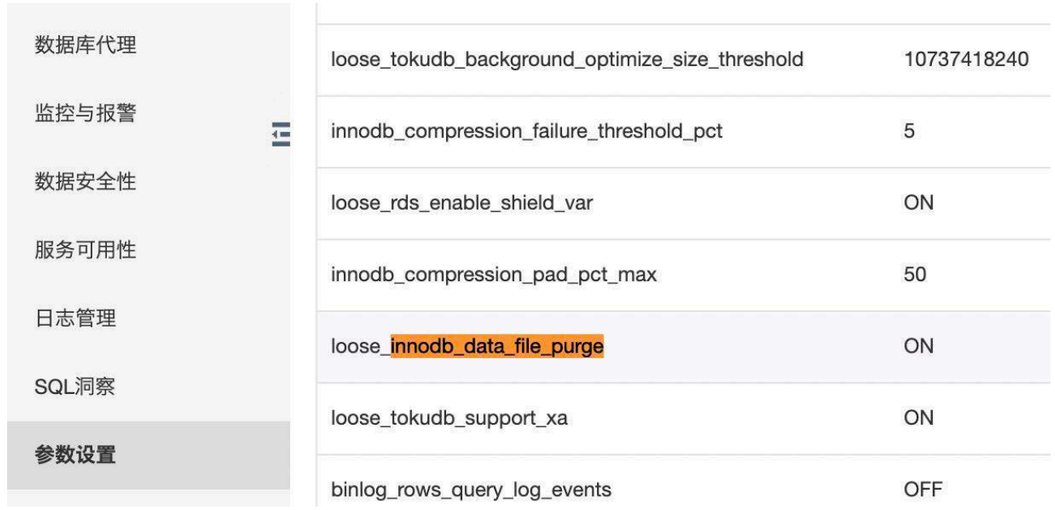
稳定性相关，不要因为人为的因素给系统造成压力，可以通过设置DMS来执行的这些SQL，不让它超过一定的时间，超过一定时间则Kill掉。

2. 禁用部分语法(企业版)

可针对DDL、DML设置部分语法不允许在大促期间执行。

3. 异步清理大表

· 内核特性



AliSQL支持通过异步删除大文件的方式保证系统稳定性。使用InnoDB引擎时，直接删除大文件会导致POSIX文件系统出现严重的稳定性问题，因此InnoDB会启动一个后台线程来异步清理数据文件。当删除单个表空间时，会将对应的数据文件先重命名为临时文件，然后清除线程将异步、缓慢地清理文件。

路径：RDS管理控制台->实例列表->参数设置

4. 突发SQL访问控制

· 内核特性--并发控制

当业务流量突然暴涨，或出现 Bad SQL 时，DBA要考虑做限流，止损恢复业务。AliSQL设计了基于语句规则的并发控制，Statement Concurrency Control，简称 CCL。

· DAS限流

为防止数据库压力过大，一般都会在应用端做优化和控制。但在以下场景，也需要在数据库端做优化控制，如：

1）某类SQL并发急剧上升；

2）有数据倾斜SQL；

3）未创建索引SQL。

（三）数据恢复

1.DMS数据追踪

在大促高峰期间，经常会人工做一些数据变更，如果数据变更出错的话，可以通过数据追踪恢复，最快五分钟内可完全恢复。

· 使用场景

1）MySQL 5.5/5.6/5.7/8.0版本；

2）DELETE/UPDATE/INSERT；

3）少量数据。

· 功能

1）在线搜索日志内容，无需手工下载 Binlog；

2）支持数据的插入/更新/删除日志搜索，无需手工解析Binlog；

3）支持逐条数据恢复，无需手工生成回滚语句。

· 支持的Binlog

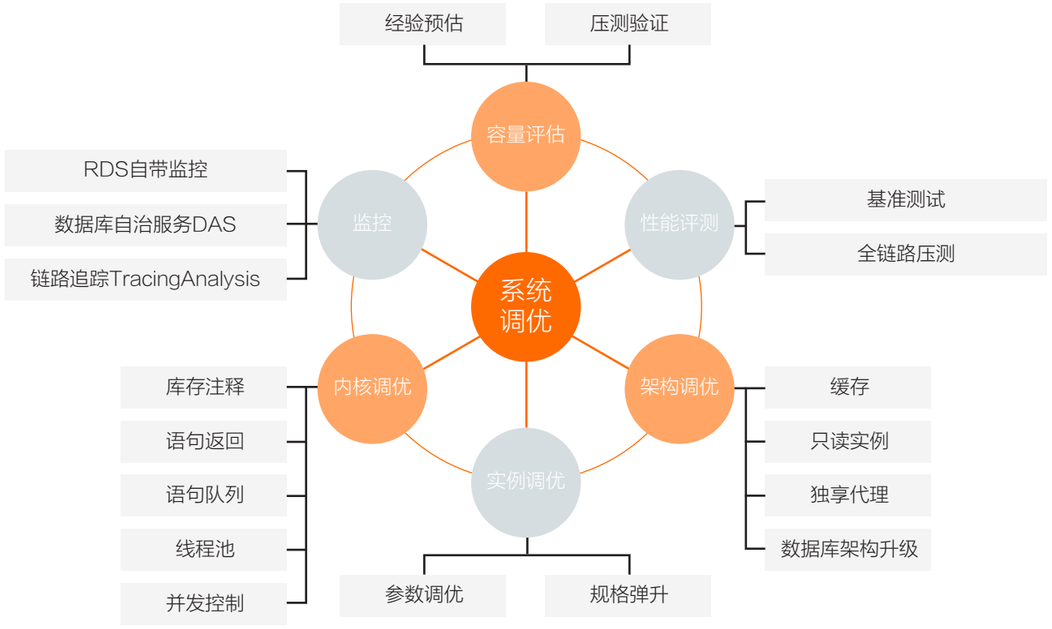
1）OSS Binlog（RDS会定时将Binlog备份到OSS上）；

2）本地热Binlog（数据库服务器上Binlog）。

2.控制台克隆实例/库表级别恢复

数据恢复还可通过备份来实现，可分为克隆整个实例恢复与库表级别恢复，还可以通过DBS数据库备份来恢复。

路径：RDS管理控制台->实例列表->单击实例ID->备份恢复



总结以上内容，系统调优主要从内容评估、性能评测、架构调优、实例调优以及内核调优等方面实现，可以借助监控报警来发现问题、解决问题，最终实现系统优化。

RDS MySQL Java 开发实战

作者：义泊

深入浅出ORM框架MyBatis

（一）为什么要选MyBatis

JDBC	Hibernate / JPA	MyBatis
简单直接	ORM	轻量+动态SQL+关联查询
开发效率低	屏蔽底层数据库差异	国内生态
重复冗余	关联查询、动态SQL不友好	不能屏蔽数据库差异

在以前是直接用JDBC进行数据库查询，优点是简单直接，缺点是开发效率低。用JDBC写程序，需要大量手工写代码，代码重复率较高，后来逐渐演化出ORM框架。

ORM框架最早期有Hibernate以及JPA规范，Hibernate能够屏蔽底层数据库差异，自动根据SQL语言生成对应底层不同数据库的方言，缺点是对关联查询支持与动态SQL能力不太友好，很难写出高效SQL。

国内目前流行的是轻量级MyBatis，对动态SQL以及关联查询的支持性较高，缺点是因为它绑定一个DB，手写SQL还要动态拼接，很难从一个DB自由的切换到另外一个DB，但由于平时很少切换DB，因此问题不是很大。

（二）MyBatis基本概念介绍

MyBatis主要分为三层：接口层，核心层与基础层。

1.接口层

是通过提供的API作为数据库进行增/删/改/查，都是MyBatis的API。

2.核心层

是SQL预处理、SQL执行、结果映射。

1）SQL预处理：是对代码里的变量进行绑定，以及动态SQL生成；

2）SQL执行：是把生成好的SQL，通过JDBC驱动，传到对应的DB里执行，而且要负责网络通信的部分；

3）结果映射：是把数据库返回的结果从关系型数据转换成Java对象数据。

3.基础层

包括日志、事务管理、缓存、连接池、动态代理、配置解析。

1）日志：是做框架里面的日志输出以及SQL语句输出；

2）事务管理：是对 JDBC事物、数据库事物做管理；

3）缓存：能够把结果集缓存在JVM的内存内部。优点是比较快，缺点是会占用堆内存。有条件的情况下，建议用户多使用分布式缓存；

4）连接池：能够加速查询，提高性能；

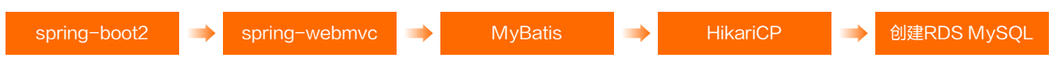
5）动态代理：在用MyBatis编程时，核心是通过接口执行数据库查询。而Mapper接口本身是没有实现的，通过注解或者XML配置SQL语句，动态代理会在运行时生成代理，当调用Mapper接口时，转换成实际的SQL语句；

6）配置解析：因为MyBatis里面有存在大量配置，需要配置新模块，读取XML配置，并把它映射为配置属性。

（三）MyBatis从0开始搭建工程

工程的搭建主要包括三部分：技术选型，项目依赖和工程结构。

1.技术选型



如上图所示，现在主流是用Spring框架做结合，还有底层有连接池也要去做结合。最新的选型是Spring-boot2，需要开发Web工程，选择Spring-Webmvc框架，持久层或ORM映射层用MyBatis框架，连接池选择HikariCP，HikariCP是Spring-boot2的默认连接池，数据库使用RDS MySQL，也是国内比较流行的云数据库。

有了以上选型之后，可以通过Spring官方网站上面的链接：“https://spring.io/quickstart”填入相关信息，就可以生成框项目模板。模板下载之后在IDE里面打开导入项目即可。

2.项目依赖：


```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.4.1</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>2.1.4</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>
```

第一部分依赖：“spring-boot-starter”，依赖的作用是自动管理开始spring-boot项目的默认依赖，已经集成在Starter里面了。

第二部分依赖：用到了Spring-webmvc，是最新的2.4.1版本。

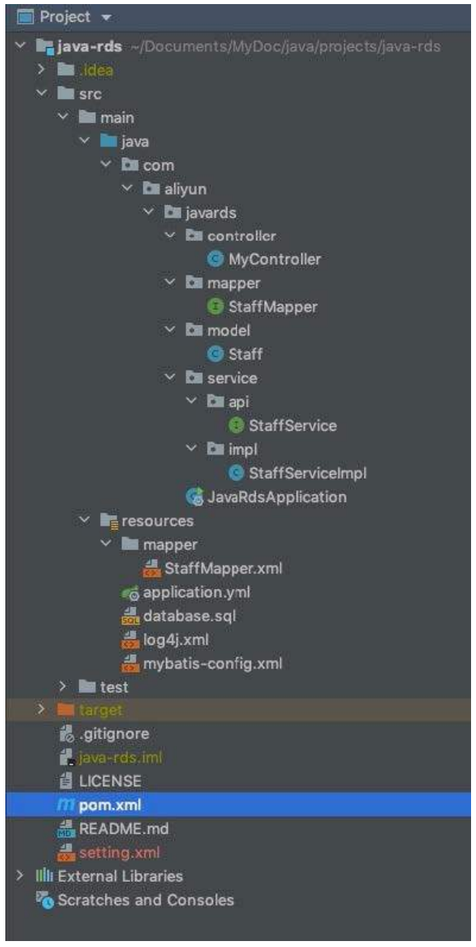
第三部分依赖：因为用到MyBatis，所以还需要依赖 MyBatis starter。需要注意的是MyBatis并没有Spring官方的 Starter，而是MyBatis社区提供的 MyBatis - spring-boot-starter。

第四部分依赖：JDBC实现，“mysql-connector-java”因为JDBC是一套规范API，具体实现由各个数据库的厂商实现。

第五部分依赖：测试框架。

第六部分依赖：为了减少代码量，还用到了Lombok。

3.工程结构



上图为工程结构图，从上往下看：

第1层：控制器层Controller，这里有一个自己的控制器。

第2层：Mapper，主要是数据库增/删/改/查的接口。

第3层：Model层，主要是在Java里面的对象定义。

第4层：Service层，包括 API层和Impl层。

API层主要是接口的定义； Impl层主要是对接口的实现。

在接口里面调用Impl层，或者调用业务层实现业务逻辑组装和编排。

第5层：是Java应用的启动入口。选择用XML方式配置MyBatis，所以在Resource目录里面，需要增加Mapper目录，mapper里面放入StaffMapper文件。

第6层：是Spring的配置文件，还有Log4j配置文件，还有Mybatis配置文件。Mybatis配置的也可以通过编程的方式实现。

第7层：有Pom文件。

通过以上操作，从0开始搭建的MyBatis工程就完成了。当工程启动后，在浏览器里面输入“http://localhost:8001/query?name=yanglong”链接，这里的端口还有路径都是自己定义的，可以验证工程是否正常运作。

连接池框架剖析和最佳实践

（一）为什么要用连接池?

不用连接池	连接池
建连接较耗时（两层握手）	性能更佳
连接数有限	连接数受控+健康探测
大量TIME_WAIT	监控和管理灵活

不用连接池会存在以下问题：

1）如果不用连接池，每一次查询都需要建立连接，有两层握手。第一层是TCP层的握手，第二层是 MySQL协议握手。两层协议大概需要有多个TCP数据包，这些都需要时间，在数据库内部还需要处理，建立连接是费时间的操作；

2）对于当代的应用来说，应用服务器一般有很多台，而数据库服务器相比之下可能少一些。大量应用服务器会存在一个问题，在业务流量高峰期存在对DB的连接，而DB能够承载的连接数有限。所以说如果不用连接池，那么这个连接的数量就不受控制，严重情况下可导致DB性能降低；

3）如果不用连接池，意味着每次执行SQL语句时，都需要创建TCL链接和关闭TCL链接，而关闭动作是在应用端完成，导致应用服务器上存在较多TimeWait状态的TCP连接。TimeWait状态的连接数达到一定数量之后会引起应用问题，例如端口不够用。

用连接池的优点：

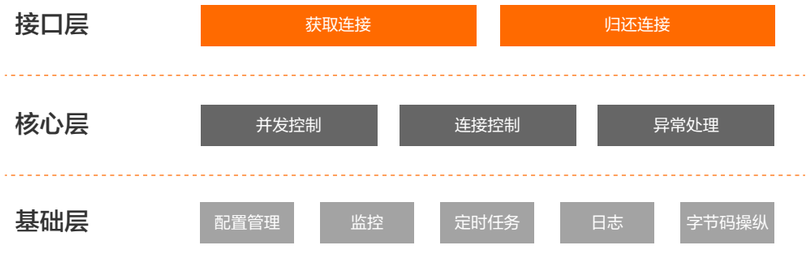
1）不用每次都建立连接，而是直接从连接池里取连接，性能更佳；

2）连接池可以控制连接数量，以及当连接出现问题时，连接池能够去自动探测连接是否存活，如果连接中断，连接池会自动重建；

比如应用使用RDS的MySQL，需要对MySQL的实例进行配置变更。如升规格，提高磁盘空间，遇到问题者压力大时，希望能够重启MySQL，有了连接池就能够自动处理好这些问题。

3）连接池能够对连接进行灵活的管理，对连接池配置与连接池状态监控，看到连接池里面的各种连接数量和性能指标。

（二）连接池架构



连接池架构分为：接口层、核心层、基础层。

接口层：对于MyBatis是从连接池里获取连接，连接用完之后关闭，调用连接的Close，归还连接。

核心层：负责并发控制、连接控制、异常处理。

1）并发控制：连接池里的连接数量有限，应用里面的线程数量多于连接池的连接数量。

第一种情况，当连接池里的连接都处于活跃状态时，下一个请求，想要继续得到连接需要等待，因为数量有限，需要排队。

第二种情况，同一个链接，不能分配给多个线程，否则可能会事务混乱。

2）连接控制：需要能够动态调整连接池大小，同时连接池保证连接池里面的连接数量在期望范围内。

3）异常处理：出现异常时，比如底层数据库重启，网络中断，或者连接里面发生了协议层引擎层面错误，连接已经不能再使用，这个时候连接池自动处理这些问题，将连接关闭并重新创建链接。

基础层：包括配置管理、监控、定时任务、日志、字节码操纵。

1）配置管理：连接池里面有很多的配置项，虽然常用的不多，但是可配置的点很多，需要进行解析管理。

2）监控：连接运行时需要统计和监控，最好能够提供查看的页面。

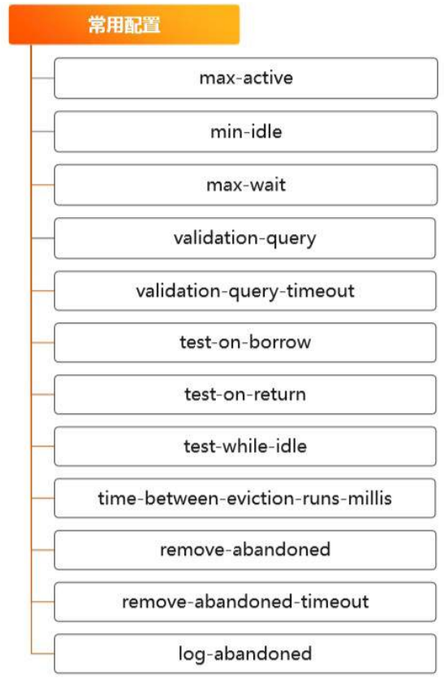
3）定时任务：连接池里空闲连接数量超过一定的程度，释放空闲连接，是通过定时任务完成。

4）字节码操纵：在Java框架里面会存在大量的字节码操纵，动态生成代理。

（三）Druid最佳实践

1.参数配置

如下图所示，常用配置包括：



1）Max-active：指的是连接池里允许的最大活跃连接数，这个值根据应用实际情况调整。

2）Min-idle：关掉多余连接，保留有效连接，节省数据库的资源，这个值根据应用实际情况调整。

3）Max-wait，指应用线程等待连接的超时。可以配几秒范围，根据业务应用实际情况进行判定。

4）Validation-query，指的是连接池探测当前连接是否是健康的SQL语句。如果是较新的JDBC，不会发SQL语句，而是发Ping命令。

5）Validation-query-timeout，指的就是探测超时的时间。

6）Test-on-borrow指连接从连接池里取出时，连接池是否需要对连接进行健康探测。建议关闭False。

7）Test-on-return，建议关闭False。

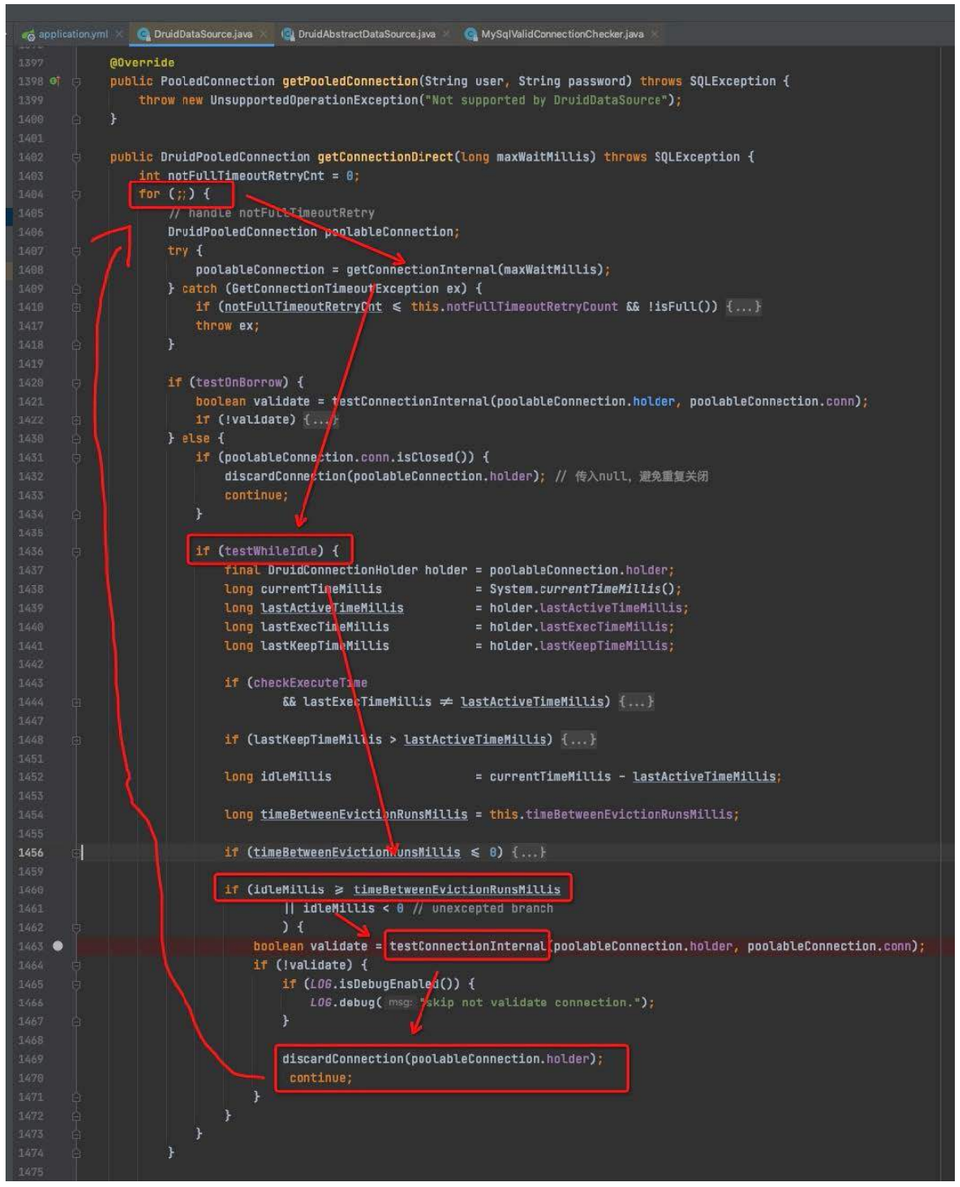
8）Test-while-idle，指的是控制当连接处于空闲状态时，是否需检测连接的健康状态。建议打开True。

9）Time-between-eviction-runs-millis指的是触发空闲连接健康探测阈值，需要跟上面的Test-while结合起来。

10）Remove-abandoned，泄露连接强制回收，默认是False关闭。

11）Remove-abandoned-timeout，指的是强制回收的触发时间阈值。配置时间不要太短，因为业务长时间使用连接，所以超时时间要比业务实际合理时间要高。配置参数单位是“秒”。

12）Log-abandoned，指的是关闭被泄露连接时输出堆栈。当一个连接被探测为连接泄露且强制关闭的时候，是否要在日志里面输出获取连接的线程的堆栈。



如上图所示，当用户开启了空闲连接的健康探测时，在Druid的源码内部会出发怎么样的逻辑?

入口是MyBatis连接Druid时，getConnectionDirect是一个循环，在循环里从连接池里取一个空闲连接，当探测空闲连接健康开关开启时，Druid会去检测连接的空闲时间是否超过配置的阈值。

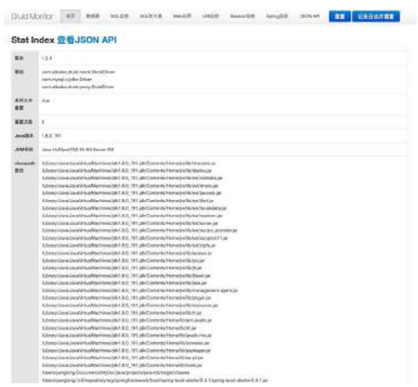
如果超过了，循环会发起一个连接的健康探测。如果探测出来连接有问题，Druid会直接把连接关闭。在循环里面，会回到循环开头，从连接池里再拿一个连接，直到拿到一个可用的连接，如果连接池里所有的连接都有问题，会重新创建连接。

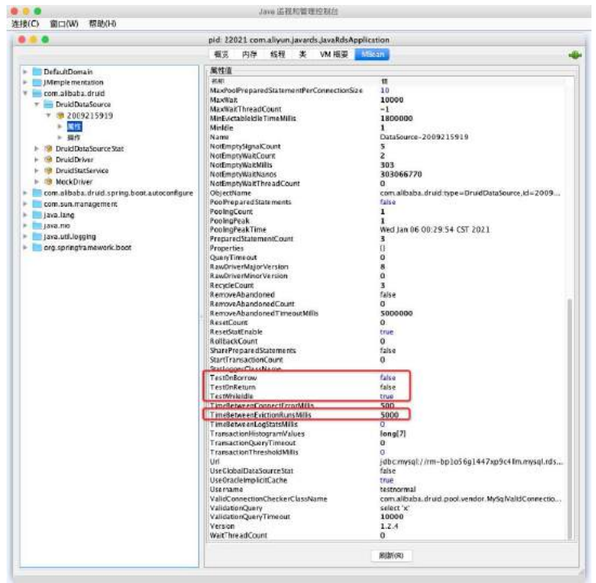
2.监控

在配置中开启监控选项，就会记录连接池的内部状态。

```
web-stat-filter:
    enabled: true
stat-view-servlet: # http://localhost:8001/druid/datasource.html
    enabled: true
    login-username: admin
    login-password: admin
    allow: 127.0.0.1/32
```

http://localhost:8001/druid/datasource.html





基于Web的页面管理页面可以去监控连接池的运行状态。如上图所示，可以查看连接时的配置参数、版本、内路径，也可以查看连接实名的连接数、 SQL监控线程监控。通过API可以也拿到这些监控数据。

除此以外，通过JMX直接连到应用内部，查看JMX对象的一些属性，如图内有些参数是可以修改的，有些只能看的还有一些操作，一些API可以去调用。

3.连接泄露诊断

· 现象

1）正常请求拿不到连接报错；

2）响应时间增大；

3）应用不可用；

· 原因

1）连接被取出后没有正常归还，导致连接长时间被占有但没有使用；

2）一般都是代码问题；

· 解决办法

1）开启连接池泄露诊断；

2）修改代码；

```
com.alibaba.druid.pool.GetConnectionTimeoutException: wait millis 10002, active 1, maxActive 1, creating 0
    at com.alibaba.druid.pool.DruidDataSource.getConnectionInternal(DruidDataSource.java:1738) ~[druid-1.2.4.jar:1.2.4]
```

```
remove-abandoned: true
remove-abandoned-timeout: 5
log-abandoned: true
```

```
2021-01-07 10:44:31.544 ERROR 33943 --- [estroy-49619396] com.alibaba.druid.pool.DruidDataSource :
abandon connection, owner thread: Thread-11, connected at : 1609987461833, open stackTrace
at java.lang.Thread.getStackTrace(Thread.java:1559)
at com.alibaba.druid.pool.DruidDataSource.getConnectionDirect(DruidDataSource.java:1477)
at com.alibaba.druid.pool.DruidDataSource.getConnection(DruidDataSource.java:1388)
at com.alibaba.druid.pool.DruidDataSource.getConnection(DruidDataSource.java:1378)
at com.alibaba.druid.pool.DruidDataSource.getConnection(DruidDataSource.java:99)
at com.aliyun.javards.controller.MyController.lambda$leak$0(MyController.java:42)
at java.lang.Thread.run(Thread.java:748)
at  com.alibaba.druid.pool.DruidDataSource.getConnectionInternal(DruidDataSource.java:1738)
~[druid-1.2.4.jar:1.2.4]
```

如上图两行日志，连接池里的连接全部被应用取走之后，新的应用与新的请求去取连接的时候就会得不到连接造成超时报错。一旦连接式的连接全部泄露，后续所有的请求都会因拿不到连接而报错。

可以通过在Spring的配置文件里面增加Druid的连接泄露诊断的参数。开启之后，当出现连接泄露时，Druid会在日志里面打印出被泄露出去的连接对应的线程堆栈。可以直接看到是什么的业务代码，在什么情况下去拿了链接而没有归还，有助于定位问题。

Java应用性能问题诊断技巧

应用性能问题的诊断主要从以下三方面入手：内存、CPU、网络。

（一）内存

1.内存

· 现象

1）OutOfMemoryError: Java heap space；

2）频繁FULL GC；

· 原因

1）内存泄露；

2）堆大小配置不合理；

· 解决方法

1）jvisualvm；

2）jstat；

3）jmap；

4）mat。

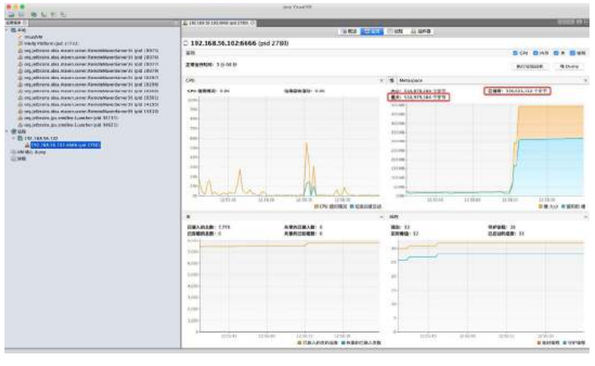
```
java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOfRange(Arrays.java:3664) ~[na:1.8.0_271]
    at java.lang.StringBuffer.toString(StringBuffer.java:669) ~[na:1.8.0_271]
    at com.aliyun.javards.controller.MyController.genString(MyController.java:74) ~[classes!/:0.0.1-SNAPSHOT]
    at com.aliyun.javards.controller.MyController.leakMem(MyController.java:63) ~[classes!/:0.0.1-SNAPSHOT]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_271]
```

```
[root@localhost ~]# jstat -gcutil 2266 1000
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  60.21  99.42  93.35  90.26    109    0.556     9    0.514    1.070
  0.00   0.00  60.44  99.42  93.35  90.26    109    0.556     9    0.514    1.070
 97.04   0.00 100.00  99.92  93.03  89.86    109    0.556    18    1.149    1.705
 95.44   0.00 100.00  99.90  93.48  90.91    109    0.556    20    1.326    1.882
 97.12   0.00 100.00  99.90  93.48  90.91    109    0.556    20    1.326    1.882
```

如上图所示，这段日志是当出现内存耗尽的时候，结果会报出来一些错误，应用会出现频繁的FULL GC。

2.内存-JMX

```
java
-Xmx512m
-Djava.rmi.server.hostname=192.168.56.102
-Dcom.sun.management.jmxremote.port=6666
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
-jar target/java-rds-0.0.1-SNAPSHOT.jar
```



诊断结内存的问题的方法：

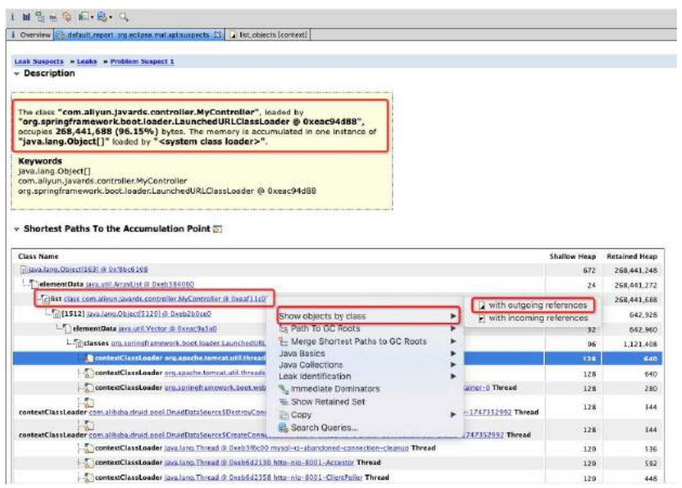
可以打开JMX通过启动参数以-D开头这4个参数进行远程连接，连接之后可以看到最大堆的大小以及实际已经使用的情况。

3.内存-Jmap

同时通过JMX对堆进行一个Dump，文件会在Jvm运行所在主机的对应的目录上。

jmap -dump:format=b,file=/share/jvm.hprof 2780

Dumping heap to /share/jvm.hprof ...
Heap dump file created



第二种是 jmap -heap 2780进程号，可以看到Jmap的堆的最大大小是512兆，同时看到老年代的使用情况是78%。通过命令把Jmap的堆内存Dump到文件中做后续分析。通过命令提示正在进行Dump，完成后会有一个Heap Dump File Created日志。

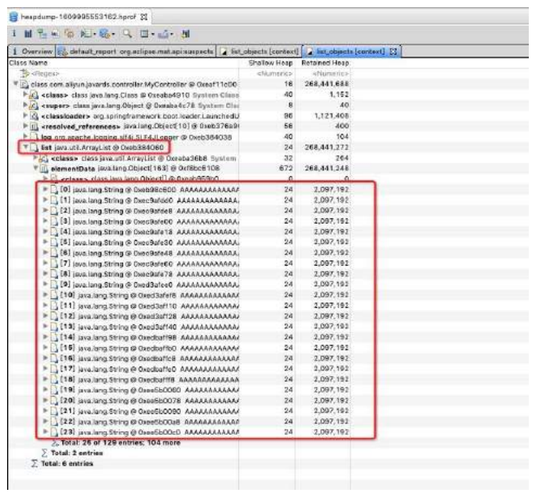
用MAT打开堆的Dump文件，然后用内存泄露分析模式，通过结果可以看到 Controller里面有大量的对象，MAT的下载地址https:/www.eclipse.org/mat/。

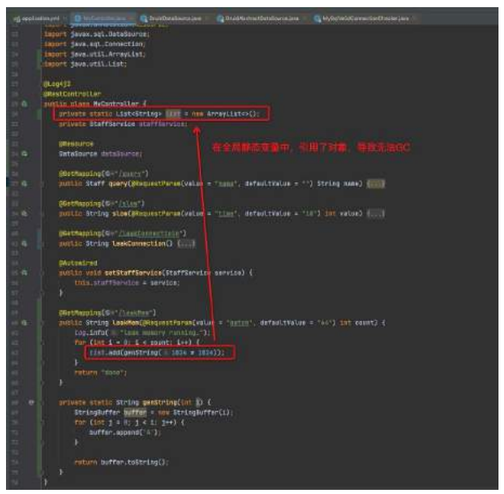
4.内存-结合代码确认问题

确认最终的问题，打开MAT打开堆的Dump文件之后，通过内存泄露的分析，可以找到MyController下的list对象，里面存在大量字符串，每个字符串的大小约为2M，大概有100个，占用了268兆，可以看到具体的字符串的内容。

结合代码，看到在代码中第63行申请了一个1M长度，2M字节大小的内存，并且把内存放到一个全局的静态变量中进行应用，所以GC无法回收，因为是一个强引用无法回收，否则将导致内存泄漏。

通过Jmap把堆Dump出来，再通过MAT工具对内存进行分析，找到占用内存的对象，再通过对象里的一些引用关系，就能够找到代码里面创建对象，找到出现问题的代码，完成内存泄露的定位。





（二）CPU

CPU问题定位：

· 现象

1）应用响应缓慢；

2）Java进程CPU占用高；

· 原因

1）存在大量消耗CPU的逻辑；

2）循环；

3）复杂计算；

· 解决方法

1）Top；

2）Jvisualvm ；

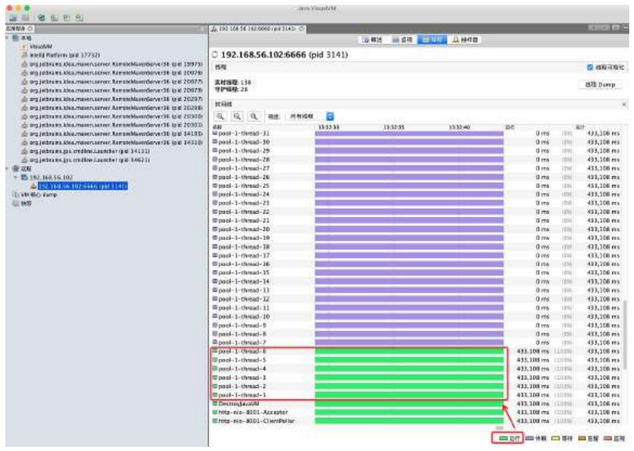
3）Async-profiler；

1.CPU-JMX或Jstack

```
top - 00:30:55 up  2:24,  3 users,  load average: 6.63, 5.40, 2.82
Tasks: 101 total,   2 running,  99 sleeping,   0 stopped,   0 zombie
%Cpu0   97.4 us,  0.9 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  1.7 si,  0.0 st
KiB Mem :  1014764 total,   435512 free,   335112 used,   244140 buff/cache
KiB Swap:  1048572 total,  1045236 free,     3336 used.   535512 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3141 root      20   0 2719316 177804  14172 S  99.0 17.5   8:26.81 java
    1 root      20   0  128012   5440   2948 S   0.0  0.5   0:01.25 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    4 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    5 root      20   0       0      0      0 S   0.0  0.0   0:00.47 kworker/u2:0
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.53 ksoftirqd/0
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   0:01.72 rcu_sched
```

如上图所示案例，一个一核的主机，CPU使用率接近100%，主机的负载达到了6点多，Java进程的CPU使用率可以到99%，通过选择JMX或者Jstack。



JMX连上去之后去检查每个线程是否在执行，这里通过JMX可以看到线程池1里的线程号1~6在长期的运行，因此这可能就是问题线程。

同时可以通过Jstack查看Jvm里面每个现场的堆栈，但是通过Jstack有一个缺点就是当应用里面线程非常多的时候，Jstack的结果会非常大，难以分析。

2.CPU-Async-Profiler

下载对应平台已编译好的代码，解压后找到对应的Java进程，通过这一串命令，对Java进程做一个系统剖析，剖析完之后会生成一个火焰图，通过火焰图能够准确地看到应用的热点代码。

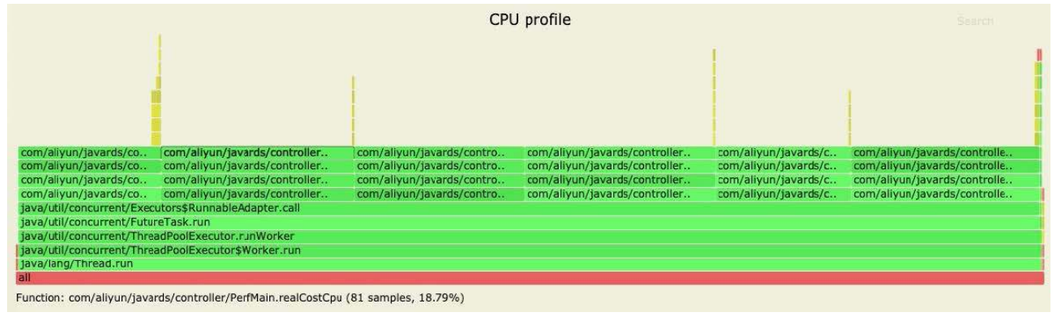
```
tar -xf async-profiler-1.8.3-linux-x64.tar.gz
cd async-profiler-1.8.3-linux-x64
./profiler.sh -e itimer -d 10 -f /share/cpu-flame-graph.svg --title "CPU profile"
3141
```


这里对上列参数做一个说明：

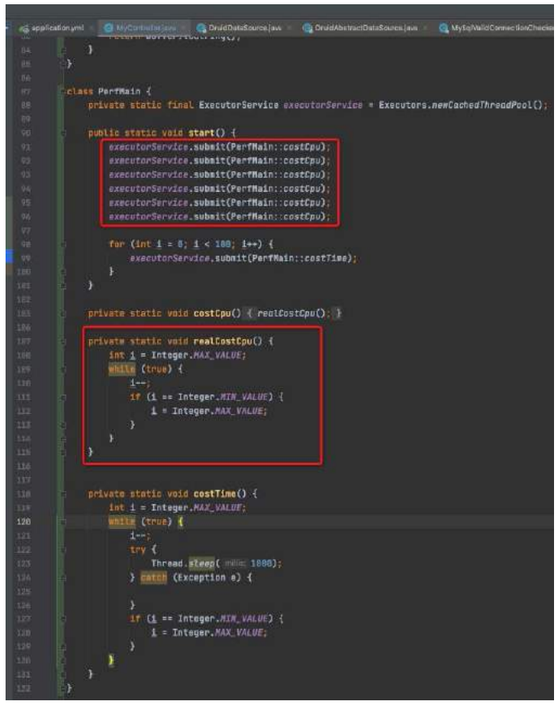
增加了-e itimer后，会不依赖perf_events，只剖析JVM一般就够了；

-d 10 表示剖析持续10秒，可以根据实际情况调整；

最后的3141是Java进程的pid。



通过生成的火焰图之后，用浏览器打开可以看到里面有6个线程长期霸占CPU。火焰图从上往下看，下面的方法是处于栈底的，上面的方法是属于栈顶的，栈顶的方法就是正在执行的方法，而栈顶上面的最宽的方法就是占用CPU最多的方法，所以可以看到这里有6个线程，里面有6个栈，每个栈上都正在执行方法在消耗CPU。



结合上图代码，通过堆栈来对应到源码。可以看到，通过JMX可以找到可疑的线程，通过Async-profiler可以生成火焰图，直接定位到存在性能问题的线程以及它的堆栈，通过堆栈就用源码找到真正有问题的代码。

（三）网络

1.常用命令

网络问题包含以下常用命令：

1）查看当前主机IP：ip a

2）查看当前主机名：hostname

3）检查目标IP是否可达：ping

4）检查目标端口是否可达：telnet

5）查看网卡：ifconfig

6）查看路由表：route -n

7）查看从当前主机发往目标主机中间会经过哪些路由：traceroute -i

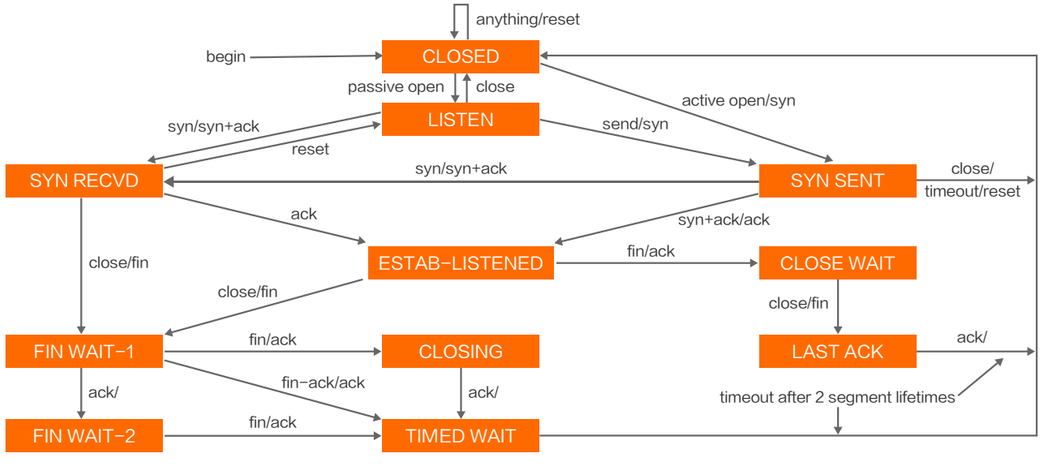
8）查看当前主机的网卡流量：iptraf-ng

9）查看以IP为单位的网络流量排名：iftop -n

10）查看当前主机上监听的端口：netstat -tpnl

11）查看当前主机上的TCP连接：netstat -tpn

2.TCP状态机



TCP状态机是TCP连接的核心部分，只有深入理解TCP状态机，才能灵活运用TCP的命令与工具，以及理解输出的结果与意义。

3.TCP状态说明：

CLOSED：表示当前连接已经关闭。

LISTEN：表示当前正监听中，随时准备接受连接请求。

SYN_SENT：表示已经发送出建立TCP连接的数据包，等待对方回应。

SYN_RECVD：表示接受到了建立TCP连接的数据包，准备给对方发送SYN + ACK。

ESTABLISHED：表示已经建立TCP连接。

FIN WAIT-1：表示主动关闭连接的一方已经发出了FIN包。

CLOSE WAIT：表示被动关闭的一方收到了FIN包。

FIN WAIT-2：表示主动关闭的一方收到了FIN的ACK包，等待对方发出的 FIN包。

LAST ACK：表示被动关闭的一方发出了FIN包，开始等待对方发出ACK。

TIMED WAIT：表示主动关闭的一方已经发出了ACK，此时主动关闭的一方要等待2倍Maximum segment lifetime，在此期间，任何因为网络延迟或者拥堵而未及时到达的包将会被丢弃，以防止下一个连接收到了上一个连接的包。

4.实战的场景

a.配置问题引起的应用阻塞

```python
import requests

s = requests.Session()
r = s.get('http://192.168.1.1')
print(r.status_code)
```

现象是一段Python（其它语言相同）程序会阻塞，应用僵死。

诊断：

```
netstat-tpn|grep python
tcp 0 1
10.0.3.15:41570 192.168.1.1:80 SYN_SENT 15349/python
2.7
```

可以看到Python程序在等待主机192.168.1.1的ACK，而这个主机根本就不存在因此无法访问。再结合目标端口是80，定位到是程序HTTP请求的目标主机错误。

b. 某个数据库的连接数暴涨，想查到连接来源

```
netstat -tpn|grep 3310|grep -i estab|awk '{print $4}'|cut -d ':'
-f 1|awk '{a[0]=a[0]=a[0] + 1} END {for (i in a) print i " " a[i]}'
10.0.3.15 999
```

可以看到，来自10.0.3.15的连接数达到了999个，结合代码就可找到问题。

c. 线上应用没有打印SQL到日志中，DB也没有开审计日志，如何能看到SQL信息?

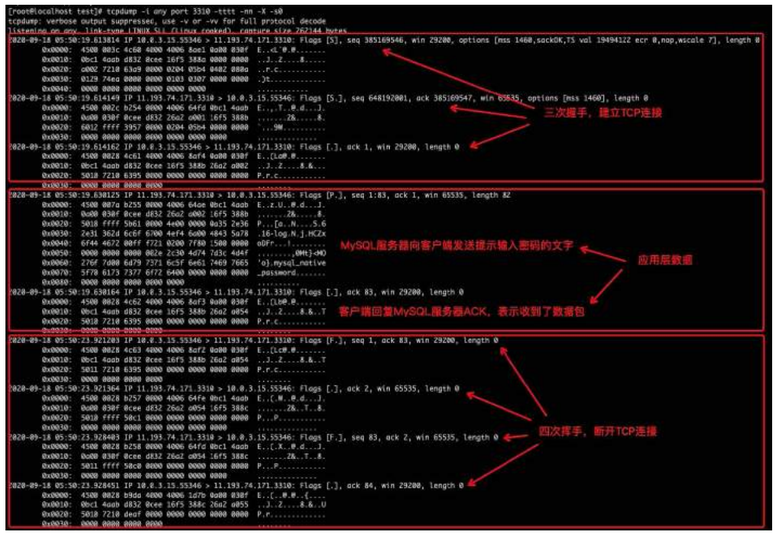
```
命令：tcpdump -i any port 3310 -tttt -nn -A -s0
2020-09-18 17:58:21.284833 IP 11.193.115.110.56454 > 11.193.115.110.3310: Flags
[P.], seq 1313:1606, ack 1238, win 6360, options [nop,nop,TS val 400339442 ecr
400339440], length 293
....E..Y..@.@.....sn..sn........u.............
........!.... select cu.id from test_table cu where cu.ins_name = 'yunji-ddr12' and
cu.is_deleted = 0 and ((cu.ins_type in (0, 2, 20) and cu.is_tmp = 0) or (cu.ins_type = 1
and cu.parent_id > 0)) and cu.status in (0,, 35, 36)
```

可以在应用服务器执行tcpdump，-i any表示监听任意网卡，port 3310表示去dump 3310端口，-tttt表示在每个包前面增加一个时间戳，-nn表示不对端口和IP进行反向域名解析，-A表示以应用层的方式输出日志，-s0表示Buffer的控制。

通过这个命令，从网络上可以清晰的看到真实的SQL语句，这对于排查问题非常有帮助。

d. 传说中的TCP3次握手和4次挥手，到底是什么样

命令：tcpdump -i any port 3310 -tttt -nn -X -s0



如上图所示，用户还可以通过Tcpdump来查看3次握手和4次挥手的网络数据。上图最上方三个包就是我们说的三次握手，它能够去建立TCP连接，建立成后MySQL会向客户端发送提示输入密码的文字，同时客户端会向MySQL客户端发送一个响应，表示收到数据包。

如果这个时候关闭连接就会产生4次挥手，分别是用户客户端向MySQL端发送一个关闭连接包文，MySQL会马上响应一个ACK，同时发出一个关闭连接包文，用户客户端也向MySQL响应一个ACK，这就是传说中TCP的3次握手和4次挥手。

MySQL查询优化

作者：苏坡

优化目的与目标

（一）为什么要优化

优化的目的主要可分为以下四个：

1）提高资源利用率；

2）避免短板效应；

3）提高系统吞吐量；

4）同时满足更多用户的在线需求。

简单来说，优化的目的是为了提高资源的利用率，让资源充分发挥价值。常见场景下，一台服务器有4大资源：CPU、内存、网络和磁盘，一旦其中某个资源出现问题，整个服务器提供服务的能力就会变差。优化的最终目的是为了同时满足更多用户的在线需求。

（二）MySQL优化目标

MySQL优化目标主要有3个：

第一，减少磁盘IO，在数据库中主要是来自于像全表扫描这种扫描大量数据块的场景，然后就是日志以及数据块的写入所带来的压力。

第二，减少网络带宽，主要是包括两个方面，第一，SQL查询时，返回太多数据；第二，插入场景下，交互次数过多。

第三，降低CPU的消耗，主要包括三个方面，第一，MySQL本身的逻辑读，第二，额外的计算操作，比如排序分组（order by group by），第三，是聚合函数（max,min,sum...）。

总结如下：

减少磁盘IO

· 全表扫描

· 磁盘临时表

· 日志、数据块fsync

减少网络带宽

· 返回太多数据

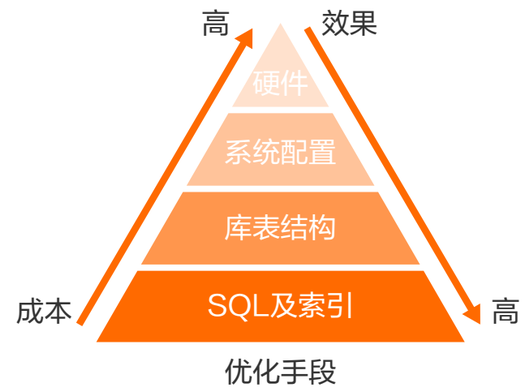
· 交互次数过多

降低CPU消耗

· 排序分组。order by, group by

· 聚合函数。max,min,sum...

· 逻辑读



上图所示的金字塔，从下往上列了4个查询的优化手段，依次是SQL及索引优化、库表结构优化、系统配置优化、硬件优化。对于单个MySQL来讲，从下往上优化，成本是逐步提升的，但效果反而越来越差。通常来说，SQL及索引调优往往不需要花费过多成本，却可以取到显著效果。

优化流程及思路

（一）关注的指标

SQL优化常规流程及思路。需要关注以下六个指标：

第一：**CPU使用率**，是SQL查询关键资源指标，CPU的消耗一般来自于数据扫描与显式计算。

第二：**IOPS**，是衡量磁盘压力的指标，它指的是每秒IO请求的次数，对数据库来说，IOPS是物理读写的关键资源指标。

第三：**QPS/TPS**，指MySQL数据库的吞吐量，也能在一定程度上反映应用系统的业务压力。

第四：**会话数/活跃会话数**，一般在应用配置问题，没有合理使用到连接池，或者SQL执行效率较差的时候出现这类的指标

异常问题，这些情况会导致数据库的Server端产生大量的会话，甚至会积压大量的活跃会话。

第五：**Innodb逻辑读/物理读**，这是主要用于反映数据库实例整体查询效率的引擎指标。

第六：**临时表**，通常来说，产生临时表往往意味着SQL执行效率的下降

总结如下：

· **CPU使用率**

SQL查询关键资源指标

数据扫描、显式计算

· **IOPS**

每秒IO请求次数

物理读写关键资源指标

· **QPS/TPS**

吞吐量

业务压力

· **会话数/活跃会话数**

应用配置

执行效率

· **Innodb逻辑读/物理读**

反映整体查询效率的引擎指标

· **临时表**

导致SQL执行效率下降的特殊行为

（二）合理监控

事实上我们实际分析问题的时候，可能还会涉及到很多其他的资源指标，而这些指标数据都需要通过一个合理的方式来获取，在比较传统的时代，是通过Top、Iostat、Sar、Dstat、show status等命令去看。

下图所示是袋鼠云EasyDO智能运维平台，可以看到整个业务系统里各个数据库示例，包括CPU、IOPS等我们所关心的指标。

（三）MySQL优化流程

第一步，**构建完备的监控体系**。为了获取性能数据，分析及诊断问题，需要建立一套相对完备的监控体系。对于这块，首先需要有细致合理的告警，其次有多维度图形化指标，只有做到这两点，才可以暴露整个系统的性能缺陷，从而掌握大规模资源。

第二步，当出现问题，或者当我们发现资源指标趋势跟预想不一致的时候，需要**分析定位问题**，这个过程就是性能诊断。一般关注5点，第一，发生异常时间区间；第二，系统日志以及数据库的错误日志；第三，Slow Log日志；第四，通过合理手段对SQL执行统计；第五，Session会话分析。诊断分析之后，定位到某些会话或者某些SQL语句，可以看到异常行为。

第三步，**分析业务逻辑**，包括3点，第一，读写需求，请求量是不是正常；第二，事务精简，事务是不是有设计上的缺陷；第三，资源调用关系，比如SQL执行本身不慢，但是因为资源调用关系，出现锁等待的问题。

以上问题分析清楚之后，接下来才是要对真正有性能问题的SQL进行优化。

第四步，**SQL优化**，关于这块主要包括4点，第一，Explain查看SQL执行计划；第二，SQL改写；第三，索引调整；第四，参数调整。

总结如下：

· **构建完备的监控体系**

细致合理的告警

多维度图形化指标

暴露性能缺陷，掌控大规模资源

· **分析定位问题**

异常时间区间

System log、DB Error Log

Slow Log

SQL执行统计

session

· **分析业务逻辑**

读写需求

事务精简

资源调用关系

· **SQL优化**

explain

SQL改写

索引调整

参数调整

（四）SQL优化原则与方法

1.优化原则

SQL优化原则主要有两点：减少数据访问量与减少计算操作。

减少访问量：数据存取是数据库系统最核心功能，所以IO是数据库系统中最容易出现性能瓶颈，减少SQL访问IO量是SQL优化的第一步；数据块的逻辑读也是产生CPU开销的因素之一。

· 减少访问量的方法：创建合适的索引、减少不必访问的列、使用索引覆盖、语句改写。

减少计算操作：计算操作进行优化也是SQL优化的重要方向。SQL中排序、分组、多表连接操作等计算操作，都是CPU消耗的大户。

· 减少SQL计算操作的方法：排序列加入索引、适当的列冗余、SQL拆分、计算功能拆分。

关于SQL优化方法，包括5点

1）创建索引减少扫描量；

2）调整索引减少计算量；

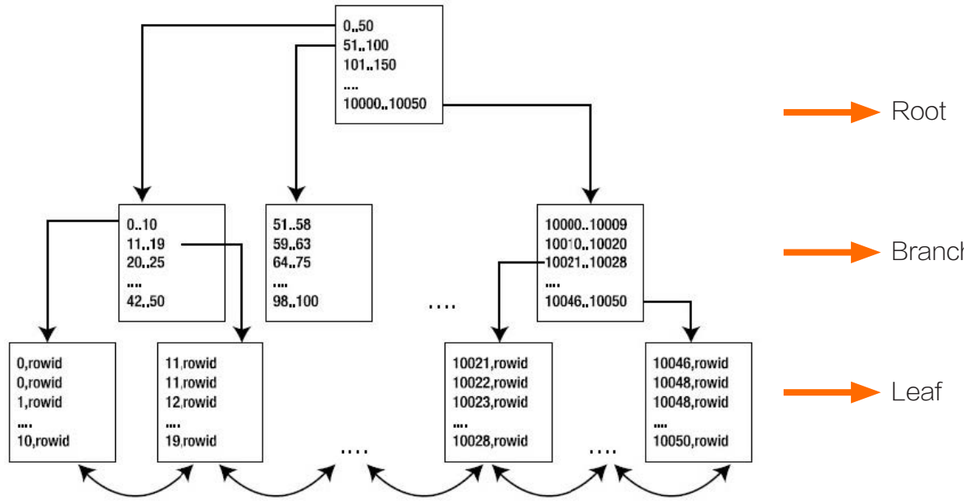
3）索引覆盖（减少不必访问的列，避免回表查询）；

4）SQL改写；

5）干预执行计划；

原理剖析

（一）B+ Tree index



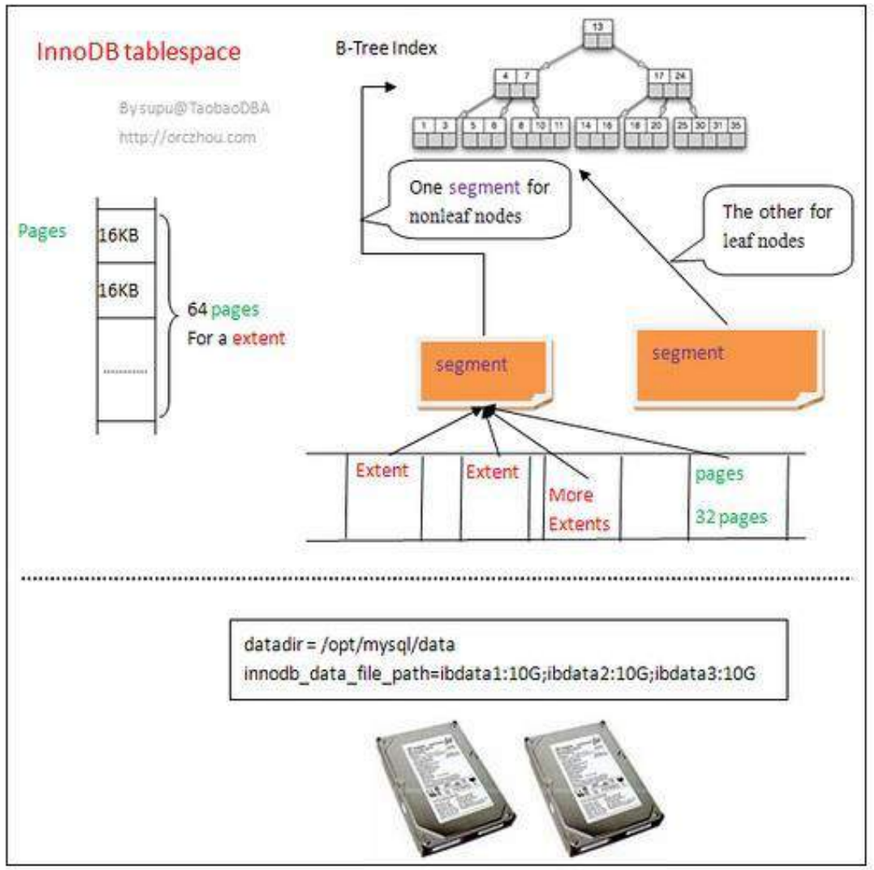
如上图所示，B+ Tree Index索引分为3个部分：根、枝、叶。

核心特点是根和枝不存储数据，行高比较固定。通过“B+Tree”索引取数据，必然经过根枝叶三个节点路径，取数据的代价比较稳定；另外一点，叶子节点上的数据是有序存储的。

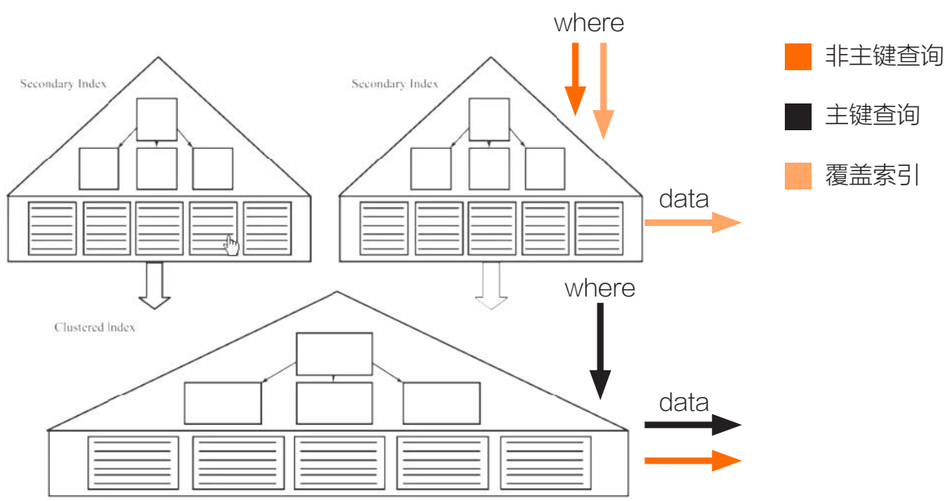
（二）Innodb Table

Innodb 是MySOL的核心存储引擎，Innodb Table是IOT有序存储，核心概念为：Innodb的表数据按照“B+ Tree”的结构进行组织，表数据本身是“B+ Tree”索引的叶子节点。

如下图所示，每张表，也就是每个存储段，实际是在MySOL里构建了一个“B+ Tree”索引的树状结构，段的物理存储跟其他关系数据库的存储方式一样分区和块。



（三）索引检索过程



如上图所示有三个流程，上面两块是二级索引，下面是属于主键索引，也叫聚集索引，是Innodb表的数据本身，依次看这三个流程：

第一，**非主键查询**，入口是从二级索引，通过二级索引，第一个过程返回聚集索引的ID；第二个过程是回表，相当于再做一次数据检索，然后从聚集索引中获取数据。

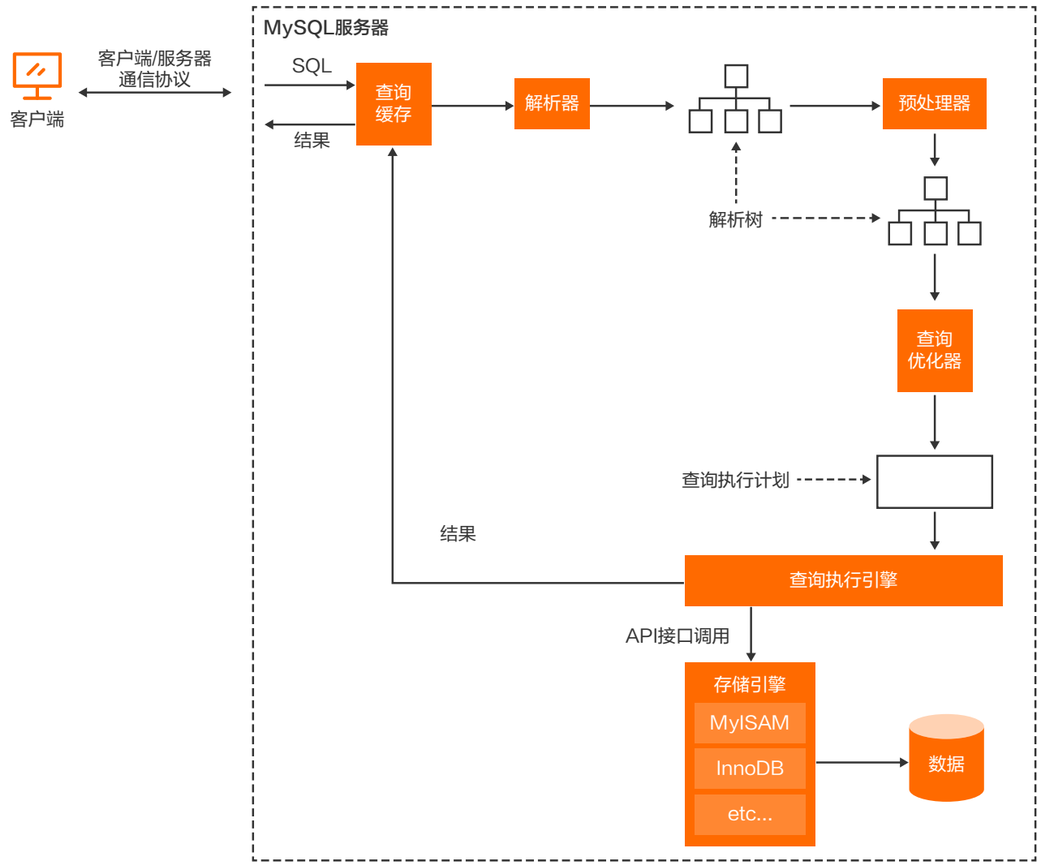
第二，**主键查询**，入口是直接通过聚集索引的ID，可以在聚集索引中获取数据。

第三，**覆盖索引**，入口是二级索引，直接从二级索引当中获取数据。

MySQL的行为

（一）MySQL SQL执行过程

1. 执行过程示例



如上图所示，MySQL的执行的过程包括：

1）客户端提交一条语句；

2）先在查询缓存查看是否存在对应的缓存数据，如有则直接返回(一般有的可能性极小，因此一般建议关闭查询缓存)；

3）交给解析器处理，解析器会将提交的语句生成一个解析树；

4）预处理器会处理解析树，形成新的解析树。这一阶段存在一些SQL改写的过程；

5）改写后的解析树提交给查询优化器。查询优化器生成执行计划；

6）执行计划交由执行引擎调用存储引擎接口，完成执行过程。这里要注意，MySQL的Server层和Engine层是分离的；

7）最终的结果由执行引擎返回给客户端，如果开启查询缓存的话，则会缓存。

2.SQL执行顺序

(8) SELECT (9) DISTINCT <select_list>

(1) FROM <left_table>

(3) <join_type> JOIN <right_table>

(2) ON <join_condition>

(4) WHERE <where_condition>

(5) GROUP <group_by_list>

(6) WITH {CUBE|ROLLUP}

(7) HAVING <having_condition>

(10) ORDER BY <order_by_list>

(11) LIMIT <limit_number>

关于SQL的执行顺序，在某些时候也可以给我们一些指导性建议。比如Where条件和Order by，在通常情况下，SQL语句先获取数据，再做Select操作，先获取数据再返回到Server端结果集的存储区之后进行排序，从这里我们可以假设如果通过索引获取数据，那么在取数据时，数据排序就已经完成，相当于MySQL存储引擎的层面已经做了优化，而不需要再增加额外的排序计算操作。

（二）MySQL优化器与执行计划

1.查询优化器

查询优化器的主要作用是用来负责生成SQL语句的执行计划。优化器是数据库的核心价值所在，它是数据库的“大脑”，优化SQL某种意义上就是理解优化器的行为。

在MySQL里面，优化的依据是执行成本，它的本质是CBO，也就是说执行计划的生成是基于成本的。目前MySQL优化器没有那么完善，执行成本主要基于行数而定。优化器工作的前提是了解数据，工作的目的是解析SQL，生成执行计划。

总结如下：

负责生成 SQL 语句的有效执行计划的数据库组件；

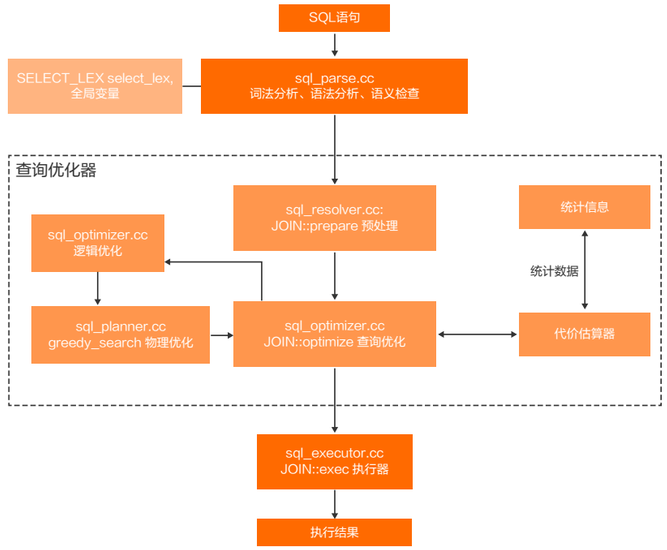
优化器是数据库的核心价值所在，它是数据库的“大脑”；

优化SQL，某种意义上就是理解优化器的行为；

优化的依据是执行成本（ CBO ）；

优化器工作的前提是了解数据，工作的目的是解析SQL，生成执行计划。

2.查询优化器工作过程



如上图所示，查询优化器工作过程包括：

1）词法分析、语法分析、语义检查；

2）预处理阶段(查询改写等)；

3）查询优化阶段，可详细划分为逻辑优化、物理优化两部分

逻辑优化：把SQL交给查询优化器之后，会去做相应的改写动作。

物理优化：过程是优化器生成获取数据去扫描数据的路径。

4）查询优化器优化依据，来自于代价估算器估算结果(它会调用统计信息作为计算依据);

5）交由执行器执行。

（三）查看和干预执行计划

在MySQL里查看SQL的执行计划直接通过Explain关键词就可以了，或者我们可以添加Extended关键字，它会展示MySQL优化器的逻辑优化改写过程。

1.执行计划

· explain [extended] SQL_Statement

```
mysql> explain select * from payment where rental_id > 1000 order by payment_date;
+----+-------------+---------+------+-------------------+------+---------+------+-------+-----------------------------+
| id | select_type | table   | type | possible_keys     | key  | key_len | ref  | rows  | Extra                       |
+----+-------------+---------+------+-------------------+------+---------+------+-------+-----------------------------+
|  1 | SIMPLE      | payment | ALL  | fk_payment_rental | NULL | NULL    | NULL | 15511 | Using where; Using filesort |
+----+-------------+---------+------+-------------------+------+---------+------+-------+-----------------------------+
1 row in set (0.00 sec)
```

当我们认为SQL的执行计划不合理时，可以通过适当的手段，强制加索引或者强制驱动表的顺序，通过这种hints方式干预SQL的执行计划。另外MySQL查询优化器的一些关键特性，我们也可以通过控制优化器开关的参数，从而控制优化器相关的行为。

2.优化器开关

· show variables like 'optimizer_switch'

```
mysql> show variables like 'optimizer_switch'\G
*************************** 1. row ***************************
Variable_name: optimizer_switch
        Value: index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine
_condition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_ac
cess=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,duplicateweedout=on,subquery_materialization
_cost_based=on,use_index_extensions=on,condition_fanout_filter=on,derived_merge=on
1 row in set (0.01 sec)
```

3.Processlist

另一种观测MySQL行为的常用手段就是Processlist。通过Processlist，我们可以看到当前在MySQL中执行的所有SQL语句，有没有异常的会话或比较特殊的SQL状态。查看会话操作可以通过2种途径：

第一，show [full] processlist;

第二，information_schema.processlist。

这里列出了几种常见的异常行为：

1）Copy to tmp table

出现在某些Alter Table语句的Copy Table操作。

2）Copying to tmp table on disk

由于临时结果集大于tmp_table_size，正在将临时表从内存存储转为磁盘存储以此节省内存。

3）Converting HEAP to MyISAM

线程正在转换内部Memory临时表到磁盘MyISAM临时表。

4）Creating sort index

正在使用内部临时表处理Select查询。

5）Sorting index

磁盘排序操作的一个过程。

6）Sending data

正在处理Select查询的记录，同时正在把结果发送给客户端。

7）Waiting for table metadata lock

等待元数据锁。

常规优化策略

（一）Select优化

1.Order by

Order by查询的两种情况：

1）Using index，是针对查询优化器的两种行为来去区分的。Using index就是说MySQL它可以直接通过索引去返回有序的记录，而不需要去经过额外的排序的操作；

2）Using filesort需要去做额外的排序，在某些特殊的情况下，可能还会出现临时表排序的情况。

优化目标：尽量通过索引来避免额外的排序，减少CPU资源的消耗。

· 主要优化策略：

1）Where条件和Order by使用相同的索引；

2）Order by的顺序和索引顺序相同；

3）Order by 的字段同为升序或降序。

注：当Where条件中的过滤字段为覆盖索引的前缀列，而Order by字段是第二个索引列时，只有Where条件是Const匹配时，才可以通过索引消除排序，而between…and或>?、<?这种Range匹配都无法避免Filesort操作。

当无法避免Filesort操作时，优化思路就是让Filesort的操作更快。

· 排序算法

1）两次扫描算法。两次访问数据，第一步获取排序字段的行指针信息，在内存中排序，第二步根据行指针获取记录。

2）一次扫描算法。一次性取出满足条件的所有记录，在排序区中排序后输出结果集。是采用空间换时间的方式。

注：需要排序的字段总长度越小，越趋向于第二种扫描算法，MySQL通过max_length_for_sort_data参数的值来进行参考选择。

· 优化策略

1）适当调大max_length_for_sort_data这个参数的值，让优化器更倾向于选择第二种扫描算法；

2）只使用必要的字段，不要使用Select *的写法；

3）适当加大sort_buffer_size这个参数的值，避免磁盘排序的出现（线程参数，不要设置过大）。

2.Subquery

对于子查询，一般的优化策略是做等价改写，在MySQL查询优化器中也叫反嵌套。在MySQL里，查询优化器本身也可以做一些简单查询的反嵌套操作，但在绝大部分情况下还是需要去做一些人为的干预。

· Subquery优化总结：

1）子查询会用到临时表，需尽量避免；

2）可以使用效率更高的Join查询来替代。

· 优化策略

等价改写、反嵌套。

如下SQL：

```
select * from customer where customer_id not in (select customer_id from payment)
```

改写形式：

```
select * from customer a left join payment b on a.customer_id=b.customer_id where b.customer_id is null
```


如上图所示，SQL语句用Not In的这样的方式，在子查询里执行Select语句。对于这个SQL语句，直接把Not In改写成Left Join，从而提升它的执行效率。在MySQL里，一般情况下Join的效率比子查询要高。

3.Limit

分页查询，就是将过多的结果在有限的界面上分多页来显示。

其实质是每次查询只返回有限行，翻页一次执行一次。

· 优化目标

1）消除排序；

2）避免扫描到大量不需要的记录。

SQL场景（film_id为主键）：

```
select film_id,description from film order by title limit 10000,20
```

此时MySQL排序出前10020条记录后仅仅需要返回第10001到10020条记录，前 10000条记录造成额外的代价消耗。

对于分页查询的优化的策略

· 优化策略一

“覆盖索引”

```
Alter table film add index idx_lmtest(title,description);
```

记录直接从索引中获取，效率最高。

仅适合查询字段较少的情况。

· 优化策略二

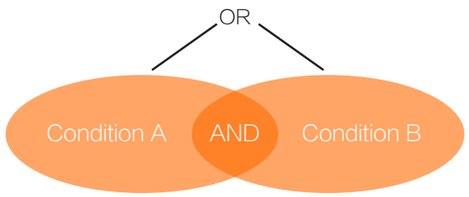
“SQL改写”

```
select a.film_id,a.description from film a inner join (select film_id from film order by title limit 1000,20) b on a.film_id=b.film_id;
```

优化的前提是Title字段有索引。

思路是从索引中取出20条满足条件记录的主键值，然后回表获取记录。

4.Or/And Condition



And结果集为关键字前后过滤结果的交集；

Or结果集为关键字前后分别查询的并集；

And条件可以在前一个条件过滤基础上过滤；

Or条件被处理为UNION，相当于两个单独条件的查询；

复合索引对于Or条件相当于一个单列索引。

· 处理策略

1）And子句多个条件中拥有一个过滤性较高的索引即可；

2）Or条件前后字段均要创建索引；

3）为最常用的And组合条件创建复合索引。

（二）Join优化

1.Nested-Loop Join算法

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions, send toclient
    }
  }
}
```

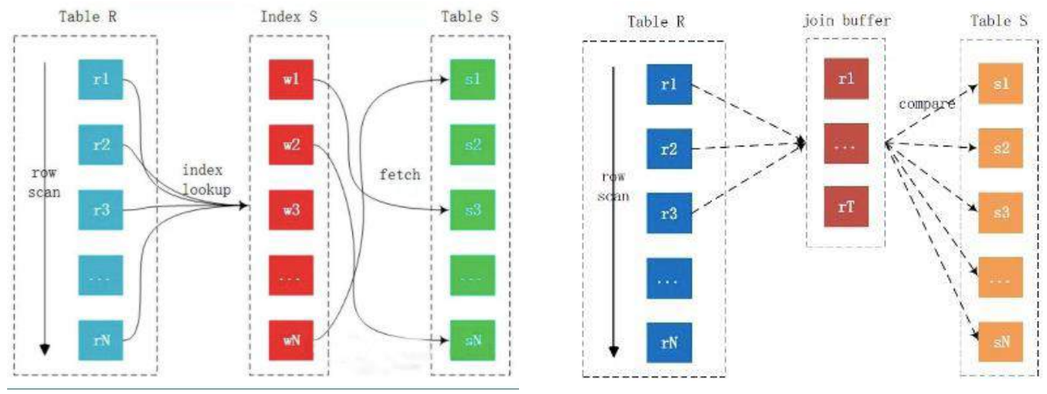
在MySQL里，关于join的典型算法就是Nested-Loop Join，也就是嵌套循环。

如上图所示，T1、T2、T3三张表Join，首先扫描T1表找到匹配条件的行，然后根据T1、T2的关联条件，再扫描　T2表找到匹配条件的的行，T2、T3也做同样的操作。

既然本质是嵌套循环，那么我们主要需要注意两点：

1）关联字段索引：每层内部循环仅获取需要关心的数据。

引申算法：Bloack Nested-Loop。



如上图所示，下方是Bloack Nested-Loop，在MySQL里有一个特性叫join_buffer，当两张表关联，如果不能够通过索引去做关联条件的匹配，这时候就会产生join_buffer的使用。

当SQL的Join语句，执行计划里出现Bloack Nested-Loop时，通常情况下，需要看关联条件是否有索引，或者是其他原因而导致关联条件的匹配没有正常使用到索引。一旦SQL语句执行计划出现Bloack Nested-Loop，绝大部分场景下都意味着SQL执行效率会大幅下降。

2）小表驱动原则：外层循环的结果集尽量小，目的是为了减少循环的次数。

2.关联字段索引的必要性

案例：

```
mysql>  select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;

+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (1 min 50.26 sec)

mysql>
mysql> explain  select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+----------------------------------------------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   4000 |   100.00 | NULL                                               |
|  1 | SIMPLE      | a     | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 127042 |    10.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)

mysql>
```

如上图所示，对于这一条Select语句，是两个表Join关联。SQL语句执行计划时，出现了join_buffer，执行计划extra部分就是前面所说的Bloack Nested-Loop。

我们可以看到，通过b表关联访问a表时，Rows是127042，整个访问过程的代价特别大，对于这种场景，优化策略是给关联条件添加索引。如下图所示：

```
mysql>
mysql> alter table film2 add index(film_id);
Query OK, 0 rows affected (0.32 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain  select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref             | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+------+----------+-------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL    | NULL    | NULL            | 4000 |   100.00 | NULL        |
|  1 | SIMPLE      | a     | NULL       | ref  | film_id       | film_id | 2       | sakila.b.film_id |  125 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql>  select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;
+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (0.31 sec)
```

后面可以看到，通过b表访问a表时，执行计划里Key使用到了刚才所添加的索引，Rows从127042下降到125。前者执行时间接近两分钟，后者只需要0.31秒，执行效率大幅提升。

3.小表驱动原则

忽略b表的索引，使b表作为驱动表，如下图所示：

```
mysql> explain  select count(*) from film2 a  join film_category2 b ignore index(film_id) on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref             | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+------+----------+-------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL    | NULL    | NULL            | 4000 |   100.00 | NULL        |
|  1 | SIMPLE      | a     | NULL       | ref  | film_id       | film_id | 2       | sakila.b.film_id |  125 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql>
mysql>   select count(*) from film2 a  join film_category2 b ignore index(film_id) on a.film_id=b.film_id;
+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (0.31 sec)
```

同样的SQL语句，这里增加忽略索引的hints，目的是为了通过b表做驱动表，可以看到Rows是4000×125，执行时间是0.31秒。

忽略a表的索引，使a表作为驱动表：

```
mysql>
mysql> explain   select count(*) from film2 a ignore index(film_id)  join film_category2 b on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+--------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref             | rows   | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+--------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | ALL  | NULL          | NULL    | NULL    | NULL            | 127042 |   100.00 | NULL        |
|  1 | SIMPLE      | b     | NULL       | ref  | film_id       | film_id | 2       | sakila.a.film_id |      4 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+---------+---------+-----------------+--------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql>
mysql>   select count(*) from film2 a ignore index(film_id)  join film_category2 b on a.film_id=b.film_id;
+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (0.53 sec)
```



```
mysql>   select count(*) from film2;
+----------+
| count(*) |
+----------+
|   128000 |
+----------+
1 row in set (0.05 sec)

mysql>   select count(*) from film_category2;
+----------+
| count(*) |
+----------+
|     4000 |
+----------+
1 row in set (0.00 sec)
```

这时b表成了被驱动表，Rows为127042×4，总行数接近的情况下后者驱动表行数有明显增加。

这两个SQL语句做关联时，无论通过a表还是b表驱动，最后关联时都通过索引进行数据检索。但是由于驱动表的大小问题，导致了执行效率的不同，后面一条语句执行的时间是0.53秒，比前者慢了一倍左右。

（三）Insert优化

关于Insert的插入优化策略主要有2种：

1）优化策略一：

“减少交互次数”

如批量插入语句：

```
insert into test values(1,2,3);

insert into test values(4,5,6);

insert into test values(7,8,9);

...
```

可改写为如下形式：

```
insert into test values(1,2,3),(4,5,6),(7,8,9) ···
```

2）优化策略二：

“文本装载方式”

通过LOAD DATA INFILE句式从文本装载数据，通常比Insert语句快20倍。

常规优化策略

MySQL查询优化

1.关于MySQL的查询优化目的和目标：

优化的目的是让资源发挥价值；

SQL和索引是调优的关键，往往可以起到“四两拨千斤”的效果。

2.关于优化的流程和思路：

充分了解核心指标，并构建完备的监控体系，这是优化工作的前提；

SQL优化的原则是减少数据访问及计算；

常用的优化方法主要是调整索引、改写SQL、干预执行计划。

3.关于MySQL的核心概念及原理：

Innodb的表是典型的IOT，数据本身是B+ tree索引的叶节点；

扫描二级索引可以直接获取数据，或者返回主键ID；

优化器是数据库的大脑，我们要了解优化器，并观测以及干预MySQL的行为。

MySQL 开发规约实战

作者：芦火

前言

语句规范要建立在结构规范的基础上。

（一）字符集

1.统一字符集，建议UTF8mb4

常用的字符集包括：Latin1、gbk、utf8、utf8mb4。

常用字符集	描述	默认校对规则	最大长度	备注
latin1	cp1252 West European	latin1_swedish_ci	1	早期官方默认字符集
gbk	GBK Simplified Chinese	gbk_chinese_ci	2	非国际标准
utf8	UTF-8 Unicode	utf8_general_ci	3	alias for utf8mb3
utf8mb4	UTF-8 Unicode	utf8mb4_0900_ai_ci	4	官方8.0默认字符集

2.统一排序规则

目前互联网上以UTF8mb4字符集为主，是官方8.0默认字符集。在之前的5.5、5.6、5.7版本是建表使用的Utf8，排序规则是默认“utf8_general_ci”。在8.0之前UTF8mb4独有的默认排序方式是“utf8_general_ci”,在8.0之后默认规则为“utf8mb4_0900_ai_ci”，所以有时会出现不同版本间字符集排序规则不兼容的问题，需要注意。

（二）字段

1. 统一字段名/类型

统一字段名是为了解决**业务歧义问题**。MySQL内部系统Information_schema下边的Tables表，假设按照不规范的命名如“table_name”字段可能就会命名成“name”。如果这时Columns表中的COLUMN_NAME字段也命名为“Name”字段，在查询的时候，可能会导致意义上的混乱。统一类型是为了**解决隐式转换问题**，包括表的连接、查询都会存在隐式转换问题。

2.字段长度 varchar(255)

常见的问题是字段长度都配置为varchar(255)，在不知道业务将来存多少长度的情况下，先设成255，**在开发阶段可能比较方便，但存在性能隐患**。比如索引评估，在一个255列长的字段上建索引，实际索引评估会考虑列长，如果默认255长

度，会导致索引使用时评估不准确。

再比如字段，如果字段有2个255或3个255要做复合索引时，虽然真实的值可能每个字段长度只存了10或20，在默认参数配置下会发现索引由于太长建不出来，对线上维护与后续业务开发都有影响。

3.定义 id int primary key

PK　在业务中建议强制必须建立。可以保证主从架构下的数据一致性以及避免复制时性能问题。另一个是主键要如果采用数值型建议使用无符号类型，一般来说在一个表里ID肯定是自增的，不存在负值，如果定义一个有符号Int，会导致Int可用的值少一半。因为Int最大的值在有符号情况下是21亿，如果定义成无符号最大可以到42亿。说明数据快速增长时，有符号类型导致ID或某个自增长满的问题。

4. 禁止Null值

Null & Null =?。比如在排序场景下，两个行按某个允许null的列值做排序，如果不存储有意义的值，默认为null值的情况下，会导致一个随机的顺序，实际上就是业务上的乱序。 又比如无主键表情况下，会导致复制数据不一致的问题，所以要禁止空值。

（三）索引

0%的语句性能问题都可以靠索引解决，但索引有几个问题：

第一，**单列索引要充分评估**，比如有20个列，每个列上都有1个单列的情况，会造成对写入的影响很大，同时单列索引的建议一定要评估可选择性。

第二，**定期Review索引有效性**，索引是不是在业务中真正使用在MySQL里相对不好定位，失效索引在业务快速发展频繁变更的场景下会很常见，随着新业务新添加很多新索引，这时要看新的索引是不是已经覆盖之前的旧索引，此时旧索引实际上是没有用的。维护无效索引要多一份IO成本，删除除重复索引保留有效的即可。

第三，**不要走极端**，包括两点：复合索引所有列与所有列都建单列索引。

比如一张表有七八个列，只在单列有索引，因为索引有回表不回表的区别，所以直接建立一个包含所有列复合索引，这个方法不可取，虽然提升了查询的效率，但等于又另外维护了一张所有字段都要排序的表。

所有列都建单列索引，实际上跟是复合索引所有列是一样的。主要消耗会出现索引维护上。

索引有关内容，请关注【MySQL表和索引优化实战】课程。

SQL语句编写规范

（一）规范语法

不兼容语法：

Select * from sbtest.sbtest1 group by id;

Select id,count(*) from sbtest.sbtest1 group by id desc;

MySQL是一个相对成熟的产品，但它支持的一些语法并不标准，比如“Select * from sbtest.sbtest1 group by id;”在传统的数据库如Oracle和其他关系型数据库里中是非法的语法，系统不支持，而“Select id,count(*) from sbtest.sbtest1 group by id desc;”在 8.0版本已经淘汰。随着MySQL语法越来越规范化，在版本升级后，这种不兼容语法可能会带来应用或语句报错，因此在实际环境下不建议使用。

（二）别名

Select id,count(*) id_count from sbtest.sbtest1 group by id;

所有返回列要给有意义的命名，与列名原则一致，强制AS关键词，防止造成语意不清。

（三）执行顺序

执行顺序如下：

1 .FROM,（-including JOIN ）

2. WHERE

3. GROUP BY

4. HAVING

5. WINDOW functions

6. SELECT

7. DISTINCT

8. UNION

9. ORDER BY

10. LIMIT and OFFSET

· 语句性能应注意两个方面：

1）数据流的流向；

2）order by limit场景。

从执行顺序上看，在SELECT之前的所有子语都是在做数据筛选，SELECT以后开始执行运算，用户应注意数据流的流向。order by limit一般是在最后运行，如果在一开始运行，有时候会造成返回数据量过大，进而导致执行时间过长。

· 数据返回逻辑应注意两个方面：

1）数据的筛选机制；

2）left join where场景。

许多用户的概念中都是先做WHERE再做JOIN，这是错误的。例如在做Left Join时，先获得所有数据，再通过Where筛选数据。用户应注意梳理流向，才能最优地输出数据。

（四）如何判断语句是否已最优:explain

```
+----+-------------+-------+------------+-------+----------------+---------+---------+-----------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type  | possible_keys  | key     | key_len | ref       | rows | filtered | Extra                                               |
+----+-------------+-------+------------+-------+----------------+---------+---------+-----------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | c     | NULL       | range | idx_pk         | idx_pk  | 100     | NULL      | 1460 |     1.11 | Using index condition; Using where; Using MRR |
|  1 | SIMPLE      | b     | NULL       | ref   | i_test         | i_test  | 302     | b.c.KeyNo |    1 |   100.00 | Using where; Using index                            |
+----+-------------+-------+------------+-------+----------------+---------+---------+-----------+------+----------+-----------------------------------------------------+
```

如上图所示，用户可以通过Explain判断语句是否已最优，其中Type与Extra的主要类型与含义如下：

· Type

1）ALL：Full Table Scan 全表扫描；

2）index: Full Index Scan，索引扫描；

3）range:范围扫描；

4）ref: 表示连接匹配条件；

5）eq_ref: 类似ref，区别就在使用的索引是唯一索引；

6）const: 常量查询，比如pk等值；

7）system是Const类型的特例；当查询的表只有一行的情况下，使用system。

从性能角度来看，从上往下性能越来越高，根据《开发手册》，此处最低要求是到Range范围扫描。

· Extra

1）Using filesort 排序；

2）Using index 使用索引可以返回请求列；

3）Using index condition 通过索引初步过滤；回表再过滤其它条件；

4）Using temporary 临时表；

5）Using where 单独出现时；一般代表表上出现全表扫描过滤；

6）Using index & Using where 使用索引返回数据；同时通过索引过滤。

Extra反映了执行计划的真实执行情况。

结合上图执行计划分析，C表是外部驱动表，索引方式为idx_pk，Type是Range，Extra有Using index condition、Using where以及Using MRR，表示进行全表扫描，通过索引初步过滤，回表B再过滤其他条件。B表是从外表取数据做内循环，索引方式为i_text，扫描的列为c.b.KeyNo，这种情况说明这个执行计划相对完善。

（五）禁止与建议

SQL的语句编写包含一些禁止项与建议项语句，用户深入了解与熟练掌握这些内容能够更好地开展业务。

1.禁止项

1）select *，返回无用数据，过多IO消耗，以及Schema 变更问题；

2）Insert语句指定具体字段名称，不要写成insert into t1 values(…)，道理同上；

3）禁止不带WHERE，导致全表扫描以及误操作；

4）Where条件里等号左右字段类型必须一致，否则可能 会产生隐式转换，无法利用索引；

5）索引列不要使用函数或表达式，否则无法利用索引。

如where length(name)=‘Admin’或where user_id+2=5；

6）Replace into，会导致主备不一致；

7）业务语句中带有DDL操作，特别是Truncate。

2.建议项

1）减小三表以上Join；

2）用Union all 替代Union；

3）使用Join 替代子查询；

4）不要使用 like ‘%abc%’，可以使用 like ‘abc%’；

5）Order by /distinct /group by 都可以利用索引有序性；

6）减少使用event/存储过程，通过业务逻辑实现；

7）减小where in() 条件数据量；

8）减少过于复杂的查询和拼串写法。

（六）用数据库的思维考虑SQL

我们提倡用户用数据库的思维考虑SQL，由于数据库要处理的是数据集而非单行数据，因此与开发的逻辑不太一样。

在开发逻辑中，有时候会希望通过用一个语句解决所有问题，但这在数据库中会导致SQL语句过大甚至上万行，过于复杂的查询使得执行计划不稳定。因此我们倡导少即是美，每一层结果集都要最大限度地减小。

数据库中无法用开发应用的逻辑写语句，而应把所有的运算、判断应用逻辑都放到SQL实现。存储过程使用过重的话，会导致难以调适、定位问题。同时，应减少单条数据集处理，减少数据访问与扫描。

对于新Feature，在未经过充分测试的情况下，应谨慎考虑使用到生产中，防止造成Bug或存在性能上的问题。

（七）SQL改写

SQL有一些编写规范，这些规范是在优化日常问题时总结而来，下面举例说明。

1.SQL改写-join

select count(a.id) from sbtest1 a left join sbtest2 b on a.id=b.id

```
mysql>  select count(a.id) from sbtest1 a left join sbtest2  b on a.id=b.id;
+--------------+
| count(a.id) |
+--------------+
|    20000000 |
+--------------+
1 row in set (19.34 sec)
```

如上图所示，请注意Join键为PK，也就是左表右表应该是1对1的关系，在Left Join的情况下，可以理解成返回的数据全部是左边的数据，也就是“a”表的数据，执行时间大概为20秒。

```
mysql> explain select count(*) from sbtest1 a left join sbtest2  b on a.id=b.id;
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref         | rows     | filtered | Extra       |
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | index  | NULL          | k_1     | 4       | NULL        | 19728432 |   100.00 | Using index |
|  1 | SIMPLE      | b     | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | sbtest.a.id |        1 |   100.00 | Using index |
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql> explain  select count(a.id) from sbtest1 a ;
+----+-------------+-------+------------+-------+---------------+------+---------+------+----------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key  | key_len | ref  | rows     | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+------+---------+------+----------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | index | NULL          | k_1  | 4       | NULL | 19728432 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+------+---------+------+----------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> select count(a.id) from sbtest1 a ;
+-------------+
| count(a.id) |
+-------------+
|    20000000 |
+-------------+
1 row in set (3.32 sec)
```


如上图说是，可以将“select count(a.id) from sbtest1 a left join sbtest2 b on a.id=b.id;”简化为“select count(a.id) from sbtest1 a”，执行时间缩减到3秒左右，大幅提升执行效率。

SQL改写一般会出现在复杂查询的Join场景中，除去显式Join（left join与right join），还包括半连接（exists,in）和反连接（not exists,not in）。

此类查询过慢时，请参考执行计划，考虑是否可通过SQL改写优化。

2.SQL改写-分页统计

分页统计是一种常见的业务逻辑，例如我们现在有一条分页语句：

select a.id from sbtest1 a left join sbtest2 b on a.id=b.id limit 200,20;

取总数据量：

select count(*) from

(select a.id from sbtest1 a left join sbtest2 b on a.id=b.id) as a;

```
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref         | rows     | filtered | Extra       |
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | index  | NULL          | k_1     | 4       | NULL        | 19728432 |   100.00 | Using index |
|  1 | SIMPLE      | b     | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | sbtest.a.id |        1 |   100.00 | Using index |
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
```

分页统计是一种常见的业务逻辑，比如有1万条数据需要分页，常见的处理方法是把以上所有的语句逻辑框起来，在外面加“Count”，这种做法会导致语句冗余，且执行时间长。改写的方法有：

改写1：

select count(a.id) from sbtest1 a left join sbtest2 b on a.id=b.id;

改写2:

select count(a.id) from sbtest1 a;

```
mysql> select count(a.id) from sbtest1 a left join sbtest2  b on a.id=b.id;
+--------------+
| count(a.id) |
+--------------+
|    20000000 |
+--------------+
1 row in set (15.06 sec)

mysql> select count(a.id)  from sbtest1 a;
+--------------+
| count(a.id) |
+--------------+
|    20000000 |
+--------------+
1 row in set (0.25 sec)
```

如上图所示，两种改写方式的执行计划与最初写法的语义逻辑上无本质区别，第一种改写后执行时间为15秒，第二种改写后执行时间为0.25秒，且语句更加简单。

· 此类改写目的：

1）精简语句，简化语句逻辑；

2）进一步寻找优化空间。

事务的使用与优化

（一）事务是什么?

事务是指访问并可能更新数据库中各种数据项的一个程序执行单元(Unit)。

事务应该具有四个属性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability），这四个属性通常称为ACID特性。

目前生产环境所用的隔离级别较多，主要有以下四种：

1）Read Uncommitted

2）Read Committed（一般采用）

3）Repeatable Read （官方默认）

4）Serializable

关于事务需要强调一点：**大事务不等于长事务**。

例如：

1）Insert table batch

它是个大事务，但它可能并不长。

2）Begin

insert single data

sleep(3600)

Commit

这是个长事务，但不是大事务。

同时要说一下，有些DDL本身是原子性的，包括加列、建索引，事务可能大且长。

（二）事务的问题

长事务和大事务可能存在以下问题：

1.Undo 异常增长

导致Ibdata空间问题，增加存储成本，也会使得Hitory List过长，导致严重的性能问题。

2.Binlog 异常增长

由于单个事务不拆分存放，会导致某一个或者某一些Binlog非常的大，做复制或主从时会产生一定问题。

3.Slave延迟

DDL类，写入等，DDL是语句级回放，Slave要等到执行结束后再继续。

4.锁问题

死锁、阻塞。

针对大事务与长事务做出优化：

· 大事务

1）大事务拆分为小事务；

2）DDL拆分（无锁变更）。

· 长事务

1）合并为大事务（特别合适应用于写入场景，对写提升很大，而且数据不会特别长）；

2）事务分解（不必要的请求摘除）；

3）应用侧保证一致性。

事务使用基本原则：在保证业务逻辑的前提下，尽可能缩短事务长度。

（三）事务问题定位

1.长事务问题定位

定位命令：Information_schema.innodb_trx

例如：

SELECT trx.trx_id, trx.trx_started,

trx.trx_mysql_thread_id FROM

INFORMATION_SCHEMA.INNODB_TRX trx WHERE

trx.trx_started < CURRENT_TIMESTAMP - INTERVAL 1

SECOND。

2.锁问题定位

8.0以前：information_schema.innodb_lock_waits、innodb_locks。

8.0及以后：performance_schema. data_lock_waits、data_locks。

```
+--------------------------------+---------------------+------+-----+---------+-------+
| Field                          | Type                | Null | Key | Default | Extra |
+--------------------------------+---------------------+------+-----+---------+-------+
| ENGINE                         | varchar(32)         | NO   |     | NULL    |       |
| REQUESTING_ENGINE_LOCK_ID      | varchar(128)        | NO   | MUL | NULL    |       |
| REQUESTING_ENGINE_TRANSACTION_ID | bigint(20) unsigned | YES  | MUL | NULL    |       |
| REQUESTING_THREAD_ID           | bigint(20) unsigned | YES  | MUL | NULL    |       |
| REQUESTING_EVENT_ID            | bigint(20) unsigned | YES  |     | NULL    |       |
| REQUESTING_OBJECT_INSTANCE_BEGIN | bigint(20) unsigned | NO   |     | NULL    |       |
| BLOCKING_ENGINE_LOCK_ID        | varchar(128)        | NO   | MUL | NULL    |       |
| BLOCKING_ENGINE_TRANSACTION_ID | bigint(20) unsigned | YES  | MUL | NULL    |       |
| BLOCKING_THREAD_ID             | bigint(20) unsigned | YES  | MUL | NULL    |       |
| BLOCKING_EVENT_ID              | bigint(20) unsigned | YES  |     | NULL    |       |
| BLOCKING_OBJECT_INSTANCE_BEGIN | bigint(20) unsigned | NO   |     | NULL    |       |
+--------------------------------+---------------------+------+-----+---------+-------+
```

开发中常见问题与最佳实践

（一）分页问题

分页的传统写法：select * from sbtest1 order by id limit M,N。

它存在的问题点：需要扫描大量无效数据后，再返回请求数据，成本很高。

根据业务需求，有以下三种解决方法：

1）select * from sbtest1 where id > #max_id# order by id limit n;

适用顺序翻页的场景，每次记录上一页#max_id#带入下一次查询中。

2）select * from sbtest1 as a inner join (select id from sbtest1 order by id limit m, n) as b

on a.id = b.id order by a.id;

适用只按照id进行分页，无Where条件。

3）select * from sbtest1 as a

inner join (select id from sbtest1where col=xxxx order by id limit m, n) as b

on a.id = b.id order by a.id;

适用于带Where条件，同时按照ID顺序分页。此时，需要在Where条件上创建二级索引。

（二）大表数据清理

1.数据清理场景

一般的数据清理场景为历史数据清理，例如数据归档、Delete等。

这里经常存在的问题有：

1）单次Delete行数过多，容易导致锁堵塞、主从复制延迟、影响线上业务；

2）易失败，死锁、超时等。

解决的建议方案：

1）伪代码

Select min(id),max(id) from t where gmt_create<$date

For I in “max(id)-min(id)/1000

Delete from t where id>=min(id) and id<min(id)+1000 and gmt_create<$date

.....

2）定期Optimize Table回收碎片。

2.全表数据清理

常用的方法：用Truncate删掉整张表的数据。

存在问题：大表（如：>100G），Truncate期间会造成IO持续抖动。

解决方案：硬连接方式后Truncate，异步Trim文件。

（三）隐式转换问题

隐式转换问题发生在比较值类型不一致的场景下，除去一些规定情况，所有的比较最终都是转换为浮点数进行。

Create table testtb(id varchar(10) primary key);

Select * from testtb where id=1;

此类问题在编写sql语句时很难发现，上线可能会导致严重的性能问题。

（四）循环

开发环境中的循环分为外部循环与内部循环。

1.外部循环

外部循环在应用侧实现，主要问题来自每次请求的RT。

例如：

```
for i=0;i++;i<500
  insert（db 交互）
  next
rt=single rt* total count
```

建议Batch一次写入。

2.内部循环

内部循环常用在存储过程，事务无法保证。

例如：

```
While do
  insert;
  Commit;
end while
```

存在频繁Commit问题，造成IO上的冲击。

或:

```
Begin tran
While do
  insert;
end while
Commit
```

无法保证数据一致性，以及事务过长。

（五）存储过程中的事务处理

```
create procedure insertTest(IN num int)
BEGIN
  DECLARE errno int;
  declare i int;
  declare continue HANDLER for sqlexception set errno=1;
start transaction;
set i=0;
while i<num do
  INSERT testfor VALUES(i);
  set i=i+1;
end while;
if errno=1 then
  rollback;
else
  commit;
end if;
end;
```

以上方为例，在BEGIN Train后，下面最终有一个Commit。如果这里是一个重复键值，但前面已经插了10条数据，这10条数据是不回滚的，所以这个事务要直接在这里声明捕捉错误，然后回滚整个事务，才能完成这整个存储过程的回滚。

（六）常见问题

1. Where 后面的列顺序是不是要符合最左原则?

Where a=1 and b=2 等价于 Where b=2 and a=1

最左原则指的是索引顺序，不是谓词顺序，以上两个条件都匹配(a,b) 复合索引。

2. Join 的顺序是否指定左边为驱动表?

Inner Join场景下，在执行计划中按统计信息的预估自动选中驱动表，Left Join ,Right Join 时左右写的顺序才有显式意义。

3.业务上有随机返回的需求，能否用order by rand()

一般不建议，如果结果集非常小，勉强可用，但结果集大时由于随机数排序，会产生Sort操作甚至溢出到磁盘，有很大性能损耗，此类需求可以考虑伪随机算法。

4.Delete数据之后，为什么磁盘空间占用反而大了?

Delete数据并不能清理数据文件空间，反而会导致Undo,Binlog文件的增长，使用Optimize收缩。

5.Binlog是否一定要Row格式?

在主从场景下，Binlog使用Row格式是为了保证主从数据一致性。

单机场景下，Binlog做为增长数据备份使用，同时也包括一些语句级数据恢复的功能。

6.死锁、阻塞的区别

通常说的阻塞，主要是由于锁获取不到，产生的请求被阻塞，一般需要手动解锁(Kill或等待)。

死锁不等于阻塞，虽然死锁中阻塞是必现的，但是会自动回滚事务解锁，不用手动处理，但需要业务判断语句逻辑。

以上两种情况都是由于业务侧逻辑出现，并非内核原因。

7.做DDL时是否会锁表

所有的DDL都需要锁表，只是操作顺序和操作获取时间的问题。如下图所示，允许并发DDL是No，就证明对业务有一些阻塞。

Operation	In Place	Rebuilds Table	Permits Concurrent DML	Only Modifies Metadata
Creating or adding a secondary index	Yes	No	Yes	No
Dropping an index	Yes	No	Yes	Yes
Adding a `FULLTEXT` index	Yes*	No*	No	No
Changing the index type	Yes	No	Yes	Yes

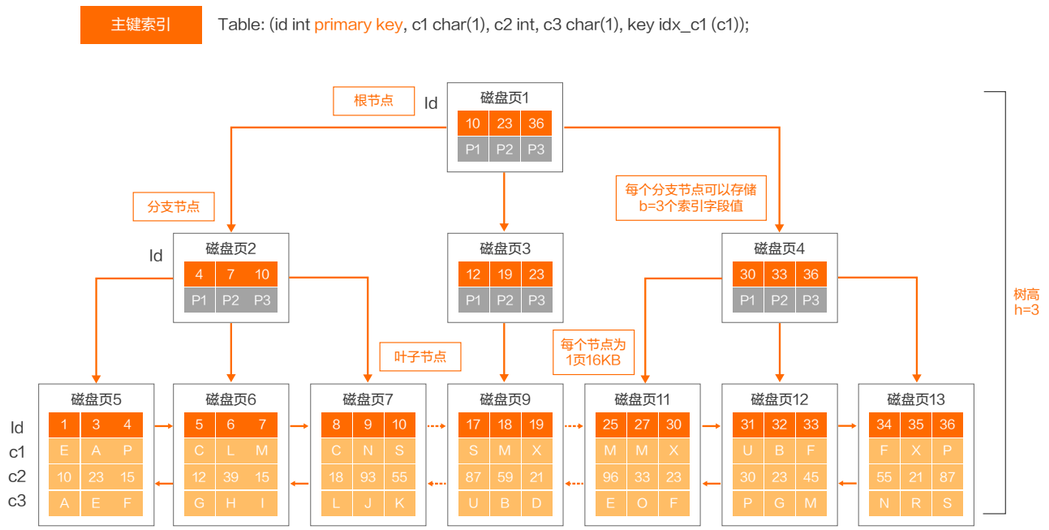
详情参考官方文档：https://dev.mysql.com/doc/refman/5.6/en/innodb-online-ddl-operations.html

RDS for MySQL表和索引优化实战

作者：田杰

主键索引

作为数据库的使用者，每天都需要跟数据库打交道，避免不了接触两个概念，一个是表，一个是索引。日常思维中，表是用来存储表中的数据，索引是用来加速查询访问。下面来看一下RDS for MySQL在InnoDB引擎下面，数据的物理组织是如何组织?



如上图所示这张表，它的主键是1个名为 id 的整型字段、4个字节、id作为主键，后面跟着一个单字符的c1，然后还有一个int类型的c2，然后单字符的c3，同时在 c1字段上有索引，这是很简单的一张表。这个数据是如何组织?

在InnoDB引擎下，数据是存储在主键中，就是指数据是通过主建进行物理组织，跟Oracle本身默认的堆表不一样，Oracle本身默认创建的表如果不指定的话是一个堆表，真的有一个对象、物理结构、数据结构，以堆的数据结构来存储数据，同时主建是另外一个数据结构，是两份数据。对于 MySQL在InnoDB引擎下面，本身数据是存储在主键的叶子节点中，如下图所示，“c1、c2、c3” 3列数据都存储在主键的叶子节点。

整个主键的数据结构是B+-Tree，B指的是Balance Tree多路平衡树，而不是Binary tree二叉树。多路平衡树和二叉树之间区别在于:

· 二叉树只有左分支和右分支，而且不限定左分支的深度和右分支的深度，也可以指树的高度，不限定左分支右分支的高度否是一致。

· 多路平衡树指，从最上面的根节点到任何一个叶子节点路径长度（即树高）是必须一致的。多路指是每一个节点，不论分支节点、根节点，下面可以挂多个子节点。

同时在整个的结构里面，在Oracle的体系中，对于每个存储数据的基础单元叫块“block”，在MySQL中做称为页“page”。在MySQL里面，如果不特意指定，默认都是16KB作为一页；从磁盘上访问，即使是读取一行数据，都要读16KB的数据到内存中。

数据组织结构有几个关键点：

第一个关键点：**数据是存储在主键中，必须显式定义主键**，如果不显式定义主键会出现两种情况：

· 第一种情况是在使用 DTS 来传输数据时，无法判断数据是不是重复，是不是唯一的。主键的定义是非空、唯一。

· 第二种情况是把RDS for MySQL的备份还原到线下的数据时，没有主键的表，读取时字段对不上。因为阿里的MySQL，为了避免出现没有定义主键而导致的问题，默认隐式的增加一个字段，把物理备份还原到本地时，这张表会多出一个字段，会导致恢复后这张表不可访问。

第二个关键点：表的每个数据块大小都是16KB，分支节点也是16KB，分支节点上的条目数越多，树越扁，树高越小，查询的越快。因为根和分支节点都需要内存运行，树高越高，需要读的16KB的块数就越多，导致查询会更慢。块的尺寸一定的情况下，里面带的条目数越多，树高越小，访问数据的代价越低。条目数取决于主键的数据类型，如下图所示，int类型4个字节，big int是8个字节，如果使用的是字符串，在utf8字符集的情况下，是char类型，要3个字节。所以关键点是主键尽量使int或者是big int这种整形，本身它很小，通常情况下一个16KB的块能放几百个int类型的条目，树高很容易控制在很小。

第三个关键点：本身它是一棵平衡树，平衡树的要求就在于，从根到任何一个叶子节点高度都要一样，树高是固定值，这棵树不断地被增、删、改的情况下，为了保证根到叶子节点的树高一致，建议用auto_increment正向递增或者使用sequence，比如在有 PolarDB-X 场景下，建议采用sequence对象来获取主键自增值，保证每次的写入的操作，尤其是插入操作，性能一致。

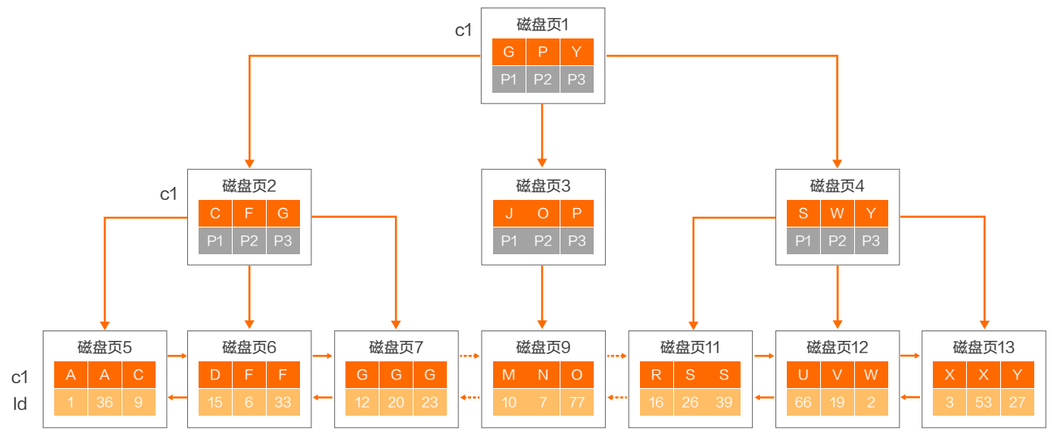
二级索引

除了主键以外的索引，其他都称之为二级索引。二级索引跟的主键索引一样，也是B+-Tree。

二级索引在MySQL里与Oracle在设计上不一样，如下图所示，对“c1”字段做了一个索引，但实际上数据存储在主键中，所以数据寻址时，没有必要放数据的真正物理地址，在Oracle里，是数据的物理地址。在MySQL里面直接放主键的值，因为知道主键的值，就能定位到这行记录。虽然索引放的是“c1”，但是真正存储时，节点中是把主键的值都要存起来，这种数据类型导致主键的数据长度不能太长，否则会有问题。

补充一下，因为主键除了单项递增数据类型要小以外，如uuid、Md5不建议用来做主键，因为长度太长。如果长度太长，磁盘块里头放的会很小，树结构会变得很臃肿，相同存储相同的数据量，索引会占很大的空间。对于 IO产生很大的影响，相同的硬件条件下，访问数据速度要慢，开销大、成本高。

二级索引　Table: (id int primary key, c1 char(1), c2 int, c3 char(1), key idx_c1 (c1));



数学分析

假设一张表里有一亿行记录，每一个16KB页，能够存放的条目数叫做平衡因子/分支因子（Balance factor）。

每个16KB分支节点可以存储的索引字段个数，取两个比较极限的情况，“b=2”每一个16KB的块里只能放两条记录，类似于二叉树，“b=100”每一个记录里能放100个条记录，就每一个16KB的块里能放100条记录。

树高(h)，树高影响物理IO次数（需要读取多少个 16 KB 页），从磁盘上获取数据到内存，需要花多少个物理IO访问到数据，数据库的增、删、改、查（select、update、insert、delete、replace），所有操作都是发生在内存。

索引B树高度：从根节点到任何一个叶子节点的节点个数。树高(h)固定的计算公式：

$$计算公式：h = □\log_b{n}□ = □\log{n}/\log{b}□$$

是跟 “b”和“n”相关的取两个的对数，“b”是平衡因子，在“b=2”的情况下，树高是27，在“b=100”的情况下，定位1行记录的开销需要读27个16KB的块到内存，b定义成100，只需读4个16KB的块到内存，开销差异很明显。

存储空间的差异更明显，只说主键“b=2”情况下，需要花费781,250MB的存储容量；“b=100”的情况下，花费15,625 MB的存储容量。

一般情况下，RDS公共云当前规格最大能到470GB内存，其中70%~75%左右是分配给InnoDB用来缓存磁盘块，“b=100”的情况下，表的数据都保存在内存里，也不会超出内存最大值。“b=2”时，只考虑了主健，不考虑任何二级索引，核心表数据也需要781GB，会需要出现物理IO换入换出数据，因为超出了内存空间。同样的存储，两种情况的代价是完全不一样，性能表现截然不同。

索引的作用

通过图书馆的模型介绍索引的作用，去图书馆内找一本书，正常的情况下，需要一个目录，通过查目录，查询到所需的书在哪个书架上面的哪个位置，这样可以快速找到，这个目录就相当于索引，它是一个从空间换时间，通过提前做准备好其他空间，来减少访问时间。需要注意的是这里只是寻找一本书，访问一个数据，如果访问的数据量，需要获取的数据量占总数据量的一定比例之后，就会引起质变。

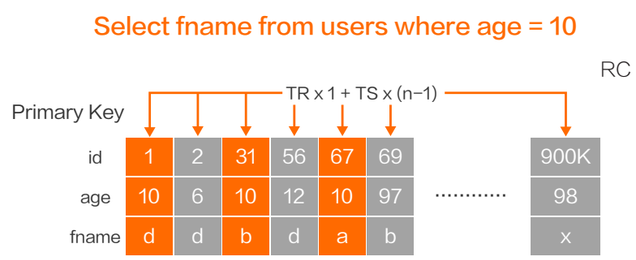
优化是提高访问效率。

两个概念：TR、TS。

TR是随机访问一个16KB的块需要的时间，TS是顺序的去读一个16KB的块需要的时间。

通过user用户表，(id int primary key, age int, fname char(1));

通过Select fname from users where age = 10进行一个简单的查询。



这里我们忽略树高，假设数据已经都在内存，因为不知道树读取的位置是在哪，那么下一次对根来说就是一个随机读，读到根在忽略树高且树高合理的情况下，需要花费n-1个顺序读来遍历表，因为没有索引。

建索引，首先考虑谓词条件， where字节后的条件是什么，根据什么条件去访问数据，根据这些条件来考虑如何去优化访问，如上图所示，age没有索引，就全面扫描，但在不同的情况下

TR = 10 ms

TS = 0.01 ms

RT = TR * 1 + TS * (n − 1)

= 10 ms * 1 + 0.01 ms * (900K −1)

= 10 ms + 9000 ms

= 9.01 Sec

RT = TR * 1 + TS * (n − 1)

= 10 ms * 1 + 0.01 ms * (9000 −1)

= 10 ms + 90 ms

= 100 ms

RT = TR * 1 + TS * (n − 1)

= 10 ms * 1 + 0.01 ms * (9 −1)

= 10 ms + 0.08 ms

= 10.08 ms

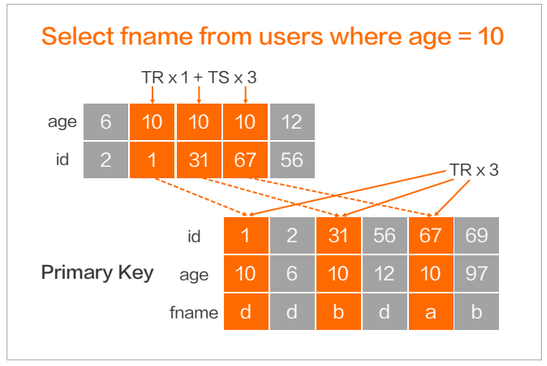
全表扫描不一定不是好事，取决于访问要获取的数据量和表内总数据量的多少。

如果表里没太大的数据量，没有必要去访问索引，根据公式推理

结论就是访问的谓词字段上没有索引，优化器会选择全表扫描方式来获取数据。全面扫描方式不一定不好，在选取的数据量较小的情况下，其效果取决于表里的数据量；随着表内数据量的上升，TS的访问的代价会快速的增加，会导致时间快速增加。

随着数据量增加，RT时间增加，表的数据量越来越多，查询越来越慢，可以考虑在age字段上面加一个索引来优化数据访问，如下图所示：

Users (id int primary key, age int, fname char(1), key idx_age (age));



RC

TR = 10 ms

TS = 0.01 ms

RT = TR * 4 + TS * 3

= 10 ms * 4 + 0.01 ms * 3

= 40 ms + 0.03 ms

= 40.03 ms

Full Table Scan : 9.01 Sec

Index Seek : 40.03 ms

9.01 * 1000 ms / 40.03 ms = 225

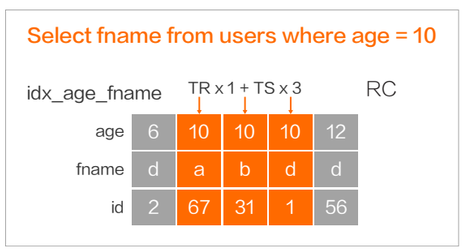
由此推出，查询慢，在谓词条件对应的情况下建索引，前提是获取的数据量是占总表的数据量很小的一部分，索引才是生效的。

实际上不管是Oracle，还是MySQL的优化器也好，根据查询，先估访问的数据量大概会占总表的数据量的多少，根据版本不一样，会有略微变化。如果查询获取的数据量超过了总表数据量的一定比例之后，就不能再走这种索引访问形式，索引访问形式会带来大量的回表访问，会引起大量的随机读。

总结谓词条件上面尽量建索引来加速访问，节省CPU，从时间来衡量，就是减少访问时间，减少查询的处理时间。

前提是访问的数据要占总表数据量很小的一部分。

如何进行优化，如下表所示，公式中最大的开销是在4个随机读上面，主要来源于回表的3个随机读，产生3个随机读的原因是在于索引里没有fname，没有需要的数据，如果把fname放到索引里，Users (id int primary key, age int, fname char(1), key idx_age_fname (age,fname));这个时候访问就会变成了1个随机读，访问的根，3个顺序读，找到3个数据，通过扫索引的3行数据，就满足了查询，公式就会变成1个随机读加3个顺序读，响应时间会是10个毫秒，这个索引叫做覆盖性索引，能够完全满足查询，不用回表，是在对于查询来说是最高效的一个访问形式。



TR = 10 ms
TS = 0.01 ms
RT = TR * 1 + TS * 3
= 10 ms * 1 + 0.01 ms * 3
= 10 ms + 0.03 ms
= 10.03 ms
Full Table Scan : 9.01 Sec
Index Seek : 10.03 ms
9.01 * 1000 ms / 10.03 ms = 900

如上图所示，从9秒钟优化到10个毫秒，900倍的性能提升，OLTP类型的应用，最核心的表最频繁的查询建议建覆盖性索引。

建组合索引的时候，区分度越高的字段考虑放在最前面，因为区分度越高，中间结果的索引片会越小越薄。原理就是只有在选择很小量的数据的时候，索引才是高效的。

反推就是中间生成的结果集或者中间的节点索引片越薄，包含的数据越小，索引对查询的加速是越高的。

最左原则，要求能匹配到的谓词条件不能出现在索引的第二个位置，谓词条件一定要匹配到第一个字段才可以，就是在匹配索引跟查询的谓词的时候，按照从左到右来匹配。如果对已知查询进行优化，在建组合索引的时候，建议把几个条件里区分度最高的放在最左边，其次，跟区分度类似的字段，尽量的往前放，经常被更新的字段尽量往后放，

最后，如果相同区分度都差不多，看它的谓词条件操作符，如果是等值的，不管是大于等于小于等于都可以，只要带等号的尽量往前放，因为如果带着等号，条件会被用来生成中间的临时索引片或者中间的结果集。其后面字段的条件依然能被用来生成中间的结果集。

fname是否能被用来生成结果集，需要看谓词条件，第一个的谓词条件，如果是等值带等号，第二个字段肯定能被用来生成中间索引片（结果集），第三个字段能不能被用来生成结果集，要看第二个是不是等值条件，就是说如果带等值条件，尽量往前放，归结起来就是：

1、如不是经常被更新的字段，区分度越高的谓词条件尽量往前放；

2、带等值条件的尽量往前放；

举例，上图如果写的是不等于d，那么age可以用来生成中间结果索引片，lname也能用来生成中间索引片，如果这里fname的谓词条件，就不能用来生成索引片，只能用来做引擎下推的过滤，或者是 server层的过滤，通过已经存储好的索

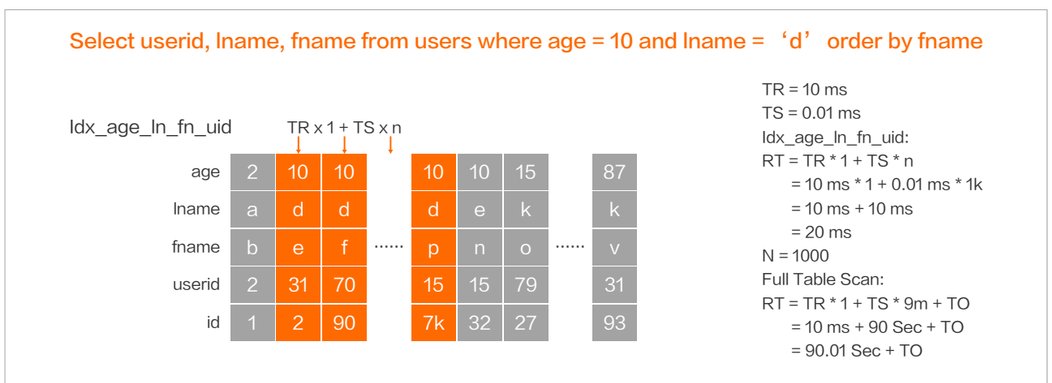
引去生成中间结果集基本上是CPU消耗是非常低的，但如果靠CPU去做过滤，生成结果集是非常耗CPU的，这是两种完全不同的访问方式。

还有如果排序之前所有的索引前置的字段都是等值的情况下，就可以直接通过索引来避免它的排序。

覆盖性索引并排序例子：

索引的作用

Users (id int primary key, userid int, age int, fname char(1), lname char(1),
key idx_age_ln_fn_uid (age,lname,fname,userid));



TR = 10 ms
TS = 0.01 ms
Idx_age_ln_fn_uid:
RT = TR * 1 + TS * n
= 10 ms * 1 + 0.01 ms * 1k
= 10 ms + 10 ms
= 20 ms
N = 1000
Full Table Scan:
RT = TR * 1 + TS * 9m + TO
= 10 ms + 90 Sec + TO
= 90.01 Sec + TO

RDS for MySQL数据库的日常运维开发的使用的规范和建议

#	分类	内容	举例	说明
1	基本	不要在数据库做运算	order by rand(), md5()	节省CPU资源，避免打满
2		控制单表数据量	1000 万行，5 GB 以内	优化访问速度和DDL时间
3		控制单表数量、字段数量	20万、50个以内	避免宽表导致行迁移
4		平衡范式，适当冗余	避免3 个以上表Join	避免过度表Join
5		拒绝3B	大的事务、SQL、批量	避免延迟、空间等问题
6		一次查询，多次复用	多个相同查询并发执行	通过Redis 缓存结果复用
7	字段	合适的数据类型	tinyint ~ bigint unsigned	节省存储空间，避免行迁移
8		避免Null	int(11) not null default 0	字段设置默认值，避免异常
9		避免& 拆分Text / Blob	content text	大字段拆分到单独表
10	索引	合理索引，组合索引优先	key (a,b,c) / key (a)	1. 依据查询决定索引 2. 控制单表索引数量 3. 左前缀匹配原则 4. 最频繁查询建立覆盖索引 5. Join 字段上建立索引
11		避免索引计算	unix_timestamp(c1) > xxxxxxxx	1. 避免索引失效 2. 避免大量调用函数
12		无符号整型自增类型主键	int unsigned auto_increment	1. 避免无主键 2. 避免uuid 做主键
13		避免外键	foreign key	应用实现业务逻辑
14	SQL	保持语句简单	多个子查询嵌套	1. 拆分或应用实现 2. 一个SQL 使用一个CPU
15		保持语句符合DB “思维”	子查询 -> 表Join	便于优化器选择最优执行计划
16		尽量减少返回数据量	select *	1. 只选择需要的字段 2. 避免框架自动生成类似语句
17		提高分页查询效率	limit 300000,50	避免分页查询带来性能影响
18		字段类型要一致	int = ‘3’;	避免隐式转换
19	行为	隔离线上、线下环境	开发、测试、预发布、生产	1. 各个环境隔离 2. 测试后才可以上线
20		禁止未审核SQL 上线	未测试引入性能问题	专人审核评估
21		统一字符集、字符序	utf8 = latin1	1. 避免性能问题 2. 避免异常
22		统一命名规范	表名order	1. 要望文生义 2. 避免使用关键字
23		大数据量操作有报备	凌晨拉取全量数据，凌晨清理数据，凌晨加载大量数据	1. 避免性能问题 2. 考虑数据安全
24		隔离应用运维操作	开发直接操作生产库	1. 避免误操作 2. 避免性能问题
25		术业有专攻	报表直接使用生产库	1. OLAP 尽量使用独立环境 2. 模糊查询考虑Open Search
26		尽量使用InnoDB 引擎	Engine=Memory	综合看InnoDB 引擎比较均衡

从研发角度深入了解RDS AliSQL内核2020新特性

作者：楼方鑫（黄忠）

关于内核

（一）回归内核

在上云的过程中，可能经历决策阶段、价格商谈阶段、数据迁移阶段和系统运行阶段。前三个阶段可能会经历较长的时间，但相比最后一个系统运行阶段，仍然是比较短暂的，需要认识到系统平稳高效运行的重要性。在这个阶段我们发现一个稳定的数据库内核非常重要，因为它会伴随你最长的时间，为你的业务和应用保驾护航。阿里云基于对数据库内核的重要性判断，投入大量精力进行内核研发和增强，并取得了不错的效果。

（二）内核研发

2020年初，阿里云对数据库内核研发方向进行深入思考，最终决定从两个方向着手，一是从用户/客户角度，二是从技术角度，下面分别介绍我们对这两个角度的思考。

1.客户角度

客户角度

客户角度分为四个方面：实用性、高性能、稳定性、安全性。

首先，我们希望加一些非常实用的功能。例如Outline功能，可以临时修改一个SQL的执行计划，把它从错误的计划变成正确的执行计划。还有Flashback功能，当数据有误操作的时候，可以快速找回误操作之前的数据，这些功能具备十分强的实用性。

其次是提高性能，我们希望在同等硬件条件下AliSQL有更高的QPS和更低的RT，这对应用十分重要。

第三是稳定性，我们希望提升高并发场景下的稳定性，并具备更快的问题分析和响应能力。

第四个是提升安全性，比如说我们对于通信与数据存储进行加密。我们实现了防止随意删表功能，对删表流程做特殊权限控制。我们研发回收站，当用户有删除操作时，先将东西放到回收站而不是直接删除，同时对回收站设置特别的权限控制。

阿里云在考虑研发客户需求内容时，会对照以上四个要求对需求进行认真思考和优先级排定。

2.技术角度

技术角度

技术角度分为四点：云场景、通用性、连续性、领先性。

首先觉得应该从云上用户场景出发，希望我们的技术能够让所有的云用户受益。过去大家对AliSQL的认知可能是通过电商业务场景的秒杀功能开始的，但现在我们认为不应受制于电商业务场景，而要从云上多个行业多种需求出发。

第二是功能和技术须具备通用性。通用性指功能的使用不受场景限制，适用于所有场景，并且用户无需修改SQL或应用代码。

第三点希望这些功能在不同的版本之间具有连续性，我们的技术或功能在RDS　56/57/80三个主流版本上保持行为的一致性。

第四个是技术领先性，或者称为独创性。我们希望这些技术或功能是其他版本或分支都没有的功能和技术创新，是一些独到的原创需求。

（三）内核特性

在这里我们按客户角度对技术和功能进行分类，分别列出AliSQL的重要技术和功能。

1.实用性

实用性方面实现四个功能，分别是是动态线程池技术，Flashback查询，SQL Outline和自动热点排队。

2.高性能

高性能方面实现了四个功能，分别是索引锁优化提升性能，Binlog In Redo，高效MySQL查询缓存和多块读技术。

3.稳定性

稳定性有四方面改进，分别是快速DDL，性能监控的插件，BP动态伸缩与页面淘汰优化。

4.安全性

在安全性方面，增加了对国密算法的支持，并实现防删表的功能和DDL回收站。

大家在后面阅读的过程中，也可以对照前面所讲的技术角度的要求来检阅AliSQL的各项技术和功能。

（四）版本对齐

功能	RDS 80	RDS 57	RDS 56	功能	RDS 80	RDS 57	RDS 56
动态线程池	✓	✓	✓	多块读	✓	✓	✓
Flashback查询	✓	✓	✓	快速DDL	✓	✓	✓
SQL Outline	✓	✓		性能监控插件	✓	✓	✓
自动热点排队	✓	✓	✓	BP动态伸缩	✓	✓	
索引锁优化	✓	✓	✓	页面淘汰优化	✓	✓	✓
Binlog In Redo	✓	✓	✓	防删表	✓	✓	✓
高效查询缓存	✓	✓	✓	DDL回收站	✓	✓	✓

✓ 已完成 ✓ 研发中 ✓ 决策中

上表为每个技术和功能在56/57/80三大主流版本上的实现情况。我们研发首先是从8.0版本开始，之后是5.7版本，最后是在5.6版本，其中有两个功能不在5.6版本实现。

第一个是SQL Outline，因为这依赖于Hint技术， 5.6版本原生框架里不支持Hint技术。

第二个是BP动态升缩，5.6版本原生框架里同样没有提供该项支持这块。

除了上述两项功能，其他的功能三个版本均可对齐，表明我们目前实现的功能大多具备良好的延续性。

（五）全网功能

在上述功能中，有部分功能达到数十万实例开启，称为全网功能。

如上图所示，全网功能有四个，分别是动态线程池、索引锁优化、快速DDL和性能监控插件。全网功能的要求是在数十万实例上开启使用，运行一年且没有出现问题。我们接下里的目标是在2021年让2020所有功能都成为全网功能。

内核本质解析

（一）实用性

实用性主要从四个方面体现：

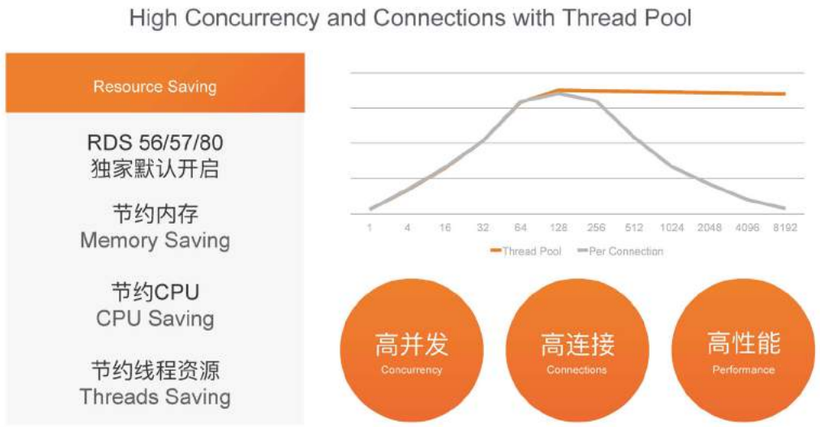
动态线程池；

Flashback查询；

SQL Outline；

自动热点排队；

1.动态线程池



动态线程池的重要性不言而喻，随着数据库的使用越来越深入，数据量也越来越大，应用到数据库连接数也越来越高。

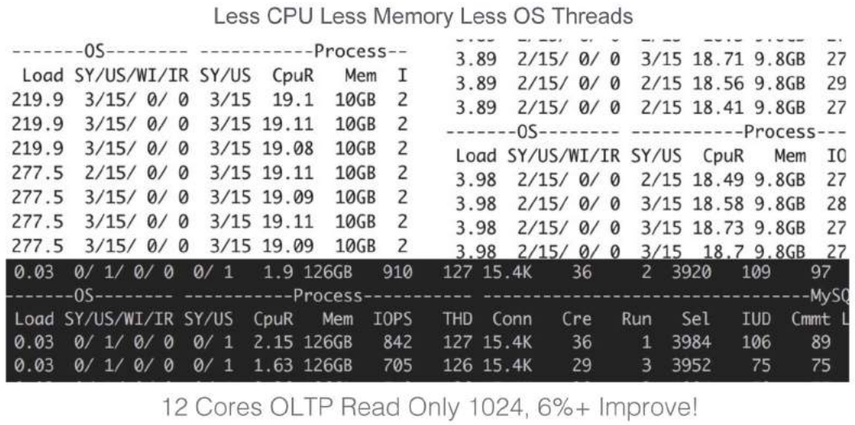
在一般情况下，性能会随着连接数变大而下降，上图灰线横坐标是数据库的连接数，纵坐标处理能力，简称QPS。在达到一定并发以后，QPS开始下降，但在动态线程池下，不同的连接数下不会出现QPS的下降，可保证系统的平稳运行。

AliSQL的线程池有四个非常重要的特性，首先是RDS 56/57/80都已经默认开启（到目前为止默认开启线程池阿里云RDS应是独家），第二是节约内存，第三是节约CPU，第四个节约线程资源，这些功能解决了许多问题。

首先是高并发的问题，当突然来了一波突发数量，例如秒杀场景，一下子来了数千到上万的请求，线程池可以保证性能不下降。

第二个是解决了连接数的上限问题，可以让RDS支持更高的连接数，通过线程池技术，连接数在技术上最高达到了50万。

第三个是高性能，如上图曲线所示，在上1000个连接以后，差距会越来越明显，到8000个链接时，原来的SQL几乎罢工，这种情况通过动态线程池得到解决。



线程池收益

上图为使用线程池之后的收益情况案例。左图和右图分别代表不开启线程池和开启线程池的区别。

当没有开启线程池时，可以看到Load数值很高，Process CPU的利用率与内存也存在不少区别。当没有开启线程池时，内存是10个GB，开启线程池后Load降到3.98，内存节约了200MB，这是一个非常显著的提升，这200MB内存可能决定了你的实例会不会触发OOM Killer。

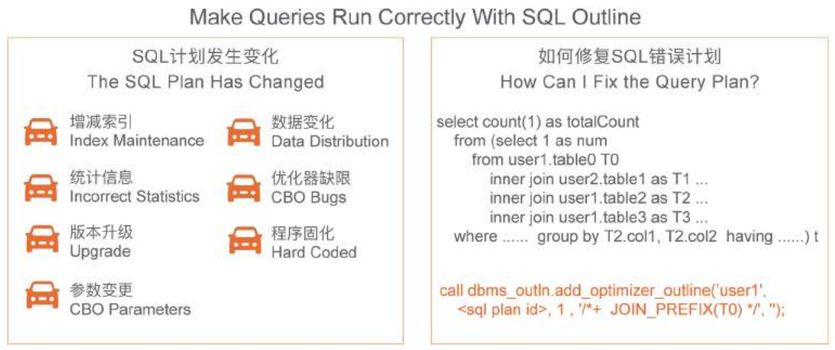
上图下方黑图是一个真实的备库实例的性能数据，它的数据库连接数有15400个（见Conn列），在开启动态线程池后，后端只需要127个线程（见THD列）就可以提供非常平稳的服务。

2. Flashback查询

很多时候用户可能会遇到以下场景，如有些表不小心做一些误操作，例如Update语句的Where条件写错，或者根本就没有指定条件，或者Delete的语句，这些非常具有伤害性。

当遇到这些情况，原来的解决办法是解析Binlog进行数据恢复，现在我们在SQL语句里加了一个“as of timestamp”语法，使得我们可以回到过去的时间点，将误操作之前的数据查出来，然后用这些数据进行快速恢复。大家可以与用Binlog进行Flashback的步骤作比较，应当能体会Flashback的简单和高效。

3. SQL Outline



如上图所示，可以看到SQL计划很多时候可能会发生变化。

当用户做了SQL索引的增减索引，做了统计信息的重新收集，做出数据库版本的升级，调了一些参数，数据分布的变化，优化器的缺陷，程序固化等，这些情况都会造成SQL计划的改变，我们可以通过解决Outline这些情况。

上图右边是一个真实案例，这个SQL多个表的关联（JOIN）顺序出错导致性能很差。通过Outline技术，只需要调用一个存储过程，给SQL设置一个Hint，让执行计划回到准确的轨道。Outline技术为业务和应用开发提供了一个行之有效的重要应急手段。

4. 自动热点排队

自动热点排队是我们对AliSQL秒杀技术的一个升级。

秒杀技术是对同一行的更新，在事务上排队且不让它进入事务层，因为同一行的更新进入到事务引擎层时，会使得事务引擎死锁检测成本非常高。这时候我们第一版本加了几个Hint，用来告诉SQL要用什么ID进行排序，这个ID通常指主键。

现在通过语法分析，可以把这一部分透明化的，不用去更改SQL，可以自动识别语句是走的是主键或者UK索引，如果是走的主键或者UK索引表示单行更新，那么可以自动把Where调节值取出来进行Hash，然后自动分配一个队列。这样只城要象右边所示那样开启开关就可以开启热点排队功能了。

（二）高性能

· 高性能主要从四个方面体现：

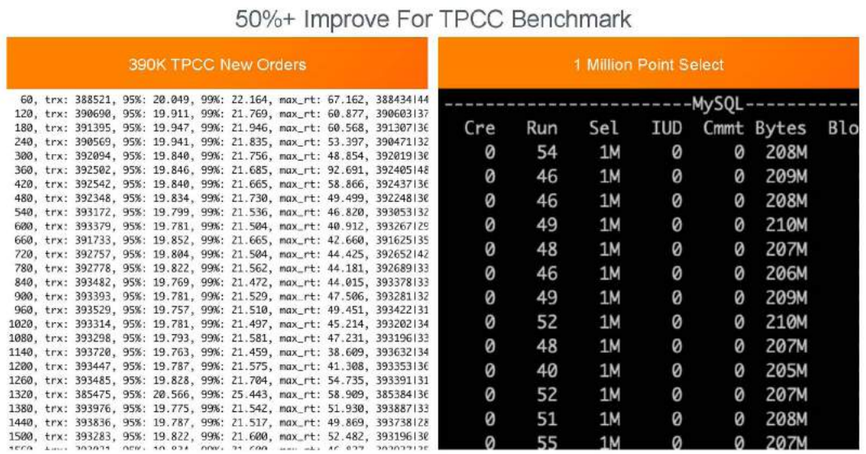
1）索引锁优化；

2）Binlog In Redo；

3）高效查询缓存；

4）多块读；

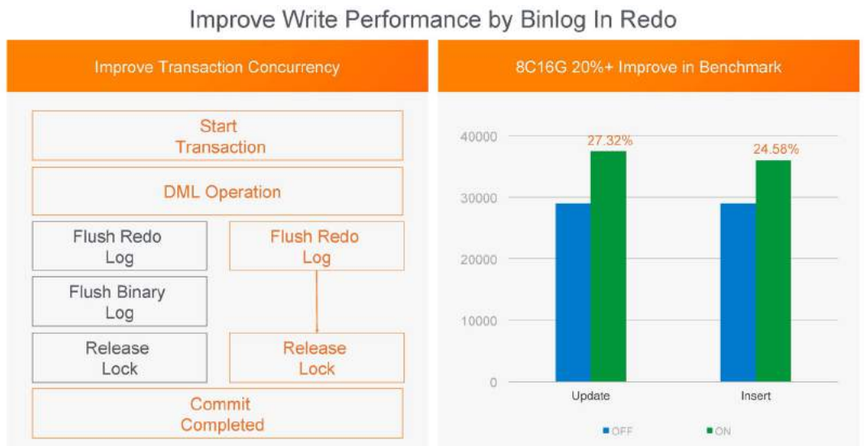
1.索引锁优化



当数据库的数据间插入时，可能会导致索引的分裂，这个操作在MySQL里执行成本非常高。我们在这块做了优化之后，大幅减少索引分裂的概率和深度。这里的深度是防止这种情况从叶子间里一直分到根，导致分裂层层传递直到根结点，这样带来好处是50%的TPCC性能的提升。

上图是一个测试，在实验室里我们可以把MySQL TPCC的值压缩到39万，左边是真实运行的截图，这表明压测数据是非常真实的。右边是我们的点查（Point Select）测试，可以看到索引锁优化对点查也有帮助，轻松达到上百万点查。

2. Binlog In Redo



Binlog In Redo技术可以提升Commit的效率。

MySQL为了保证数据的一致性，它要求Redo Log和Binlog要同时刷出去也就是Redo Log和Binlog的两阶段提交机制。

当一个事务开始之后，到提交阶段首先会Flush RedoLog，接下来会Flush Binary Log，然后才是真正的释放所持有的锁。我们在使用存储或者云盘时，可能会让性能会变得不那么理想。

我们研发的Binlog In Redo优化了这个问题，我们把Binlog先写在Redo里，只要刷一次Redo之后便可以保证后续的Binlog不会丢失，因此整个事务提交过程中的落盘操作变成只需一次。如上方左边的流程图所示，Binlog In Redo有效提高了Commit的响应速度。

这个带来了几个好处，一个是增强事务并发性，另一个是用户事务提交的性能得到提升。在我们的测试中，性能可以提升25%左右，这个技术适用于任何业务场景，无须应用更改SQL语句或调整代码。

3.高效查询缓存

MySQL本身具备查询缓存功能，这里说我们重新研发的原因在于社区8.0把查询缓存剔除，我们觉得缓存其实是个非常好的东西，因此我们在8.0基础上重新研发。

如上图左边所示，如果没有缓存的时候，一条SQL语句进来要经过解析、检查权限、优化器对SQL进行分析，得出执行计划，然后真正地执行。执行过程中可能涉及事务引擎，还可能访问磁盘上的数据，所以说这个链路非常长。

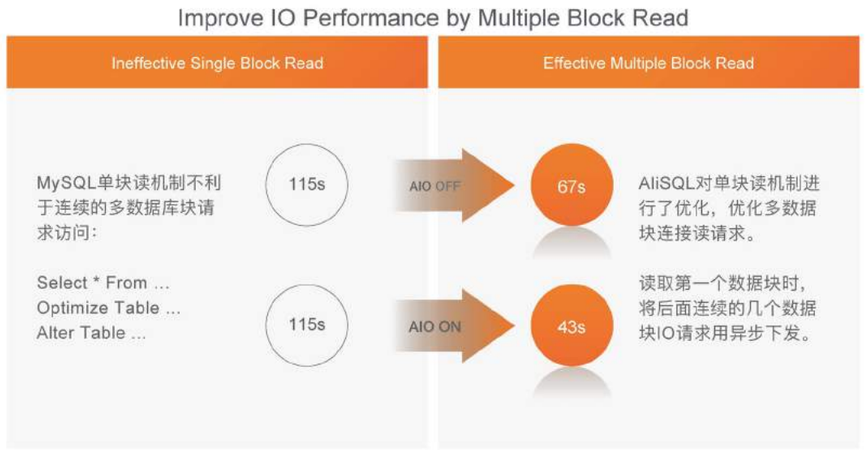
我们这里的查询缓存是对结果集的一个缓存，所以我们称之为Consistent Result Cache。Consistent是跟事务机制结合在一起，意味着从缓存查出来的数据满足事务隔离级别。

我们在4C8G的机器上测试时，发现可以有100%的提升。我们测试了三个配置，第一个是将MySQL原生查询缓存关闭的情况下，在最高峰值的时候同比其他配置不足一半。

中间橙色是MySQL原来的Query Cache，这个是我们在5.7版本上做的。原来Query Cache因为比较陈旧的并发设计，因此在现在的硬件条件下起到了反作用，这也是社区8.0将它去掉的原因，但是我们解决了其中最关键的问题，设计了一个独到性的缓存维护与更新维护机制，达到100%的性能提升，并且缓存失效时对DML的性能基本无影响。目前此功能已经可以自行开启，这些开启的参数都在控制台可以修改。建议适当缩减Buffer Pool的大小，空出一部份内存给高效查询缓存使用。

4.多块读

我们通过深入的代码阅读和分析，发现MySQL在处理IO的时候，基本上都是逐块读取。例如要查询一张大表，或者要去Optimize打表，或者对于这个表加一个索引时，会发现系统会逐块读取与处理，这个过程十分浪费时间。我们考虑到可以在读第一个块的时候，将后续的块也读进来，我们将这个称为多块读。



如上图所示，我们做了个测试。上图左边这是一个10G的表，采用原来的机制耗时115秒，右边为使用多块读技术，我们发现即使在AIO关闭的情况下，耗时仍会从115秒缩减到67秒。而当AIO打开的时候，由于本身对AIO有合并作用，提升效果更加明显，从115秒缩减到43秒。

（三）稳定性

· 稳定性方面实现了四个功能：

1）快速DDL；

2）性能监控插件；

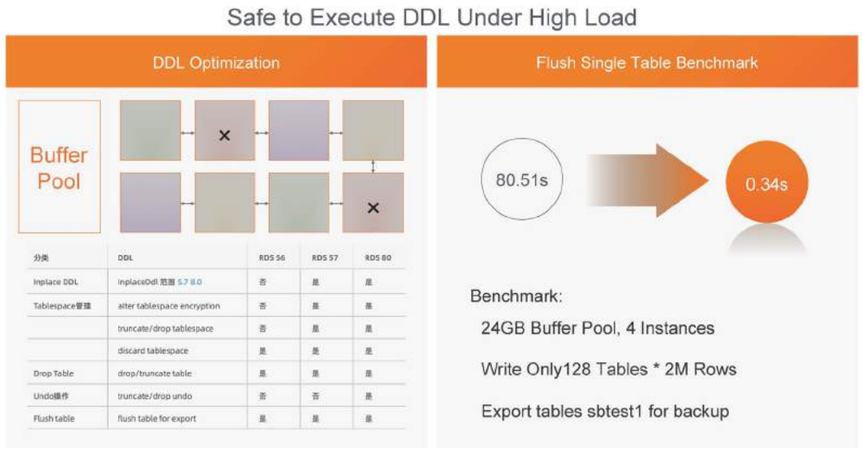
3）BP动态伸缩；

4）页面淘汰优化；

1.快速DDL

以前有用户过来找到我们，表示在高业务环境下做的DDL抖动很大。我们发现在MySQL的Buffer Pool中，很多表或者索引的数据是混合存放在同一个Buffer里。当用户要做DDL的时候，可能要把整个Buffer Pool扫描一遍，扫描效率很低。基本上所有的DDL和对表进行重组等操作都会走这个逻辑，整个过程十分缓慢。

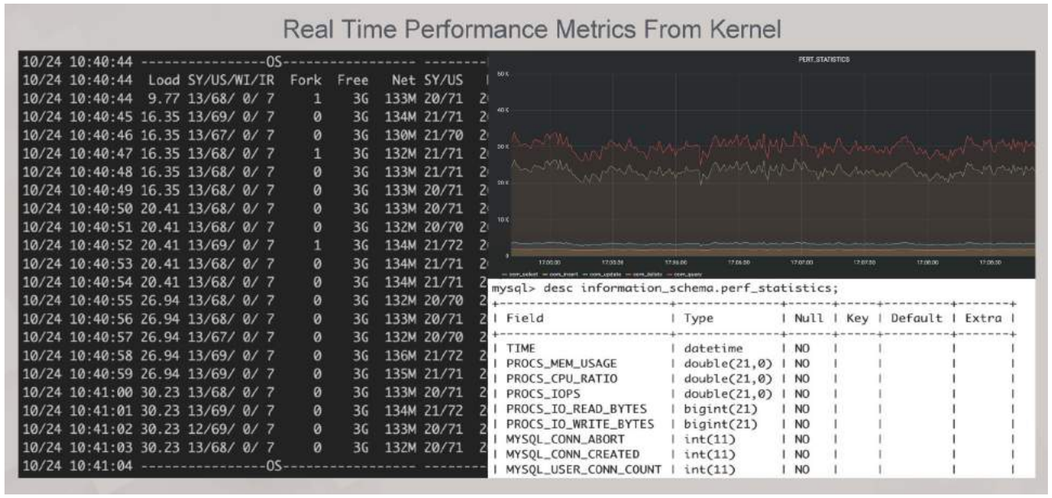
因此，我们在Buffer Pool加入一个新的导航机制，用户可以快速定位到相应的操作对象，扫描效率得到大幅度提升。



上图是一个测试的场景，是一个24G的Buffer Pool，Instances是4个。我们开启了一个压测，128张表，每张表200万条数据，然后在过程中做Export Tables，这个操作会把Buffer Pool扫描一遍。当没有快速DDL特性时，过程耗时80.51秒，当有快速DDL后，过程耗时0.34秒，对比显著。

目前快速DDL在RDS 56/57/80默认开启，是一个非常流行的全网功能，正在有效地降底在RDS上做DDL的性能抖动风险。

2. 性能监控插件



数据库中我们会经常遇到一些新的问题需要去分析，此时一个实时的性能数据就显得尤为重要。

一般情况我们会登到MySQL中用“show global status”来查看，但“show global status”执行成本高昂，很难保证每秒钟都有新的数据输出。因此，我们将性能数据的收集做到内核中，然后从内核的角度，可以直接读取内存位置，然将数据写到文件里，这样可以做到每秒钟一个数据，无论负载情况如何，基本上都非常准时。

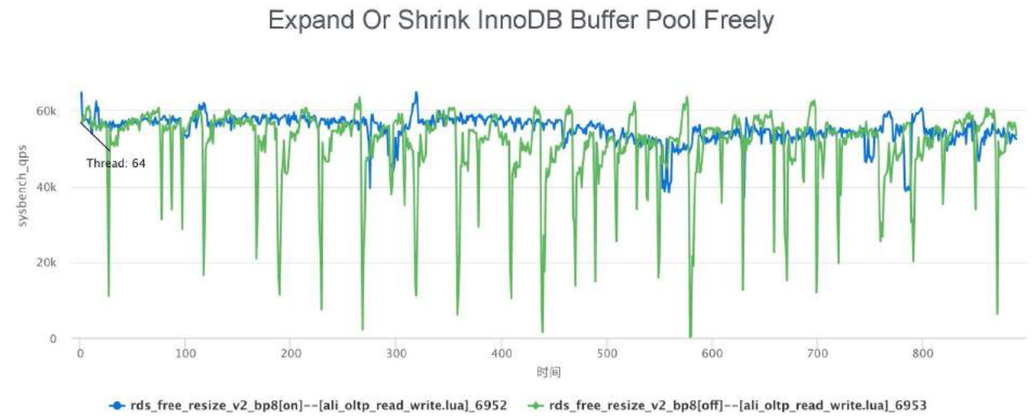
这些数据有两种存储方式，第一个是存在格式化文本文件里面，我们采用三个文件，每个文件到100MB就切换，这样可以对过去的性能信息进行回溯。

另外我们提供了一个内存表，这个表的可查询时长是可自定义的，用户可以直接访问这些表。例如上图右上方的曲线，我们把对实例的Select、Insert、Update、Delete和Query语句，用曲线画出来这些数据。用户们都可以访问，当大家在分析新问题的时候，我们可以跟用户基于同一份数据来进行交流，加快问题分析的速度。

3. BP动态伸缩

我们发现，对Buffer Pool进行伸缩调节的时候，扩大没有问题，但缩小的时候会有很大问题。

我们为什么要调Buffer Pool？MySQL内存使用分成两块，一块是数据缓存，一块是程序会话内存，每一个会话都会用到一些内存。当连接数据多的时候，其实Buffer Pool需要调小一点，当连接数比较少的时候，Buffer Pool可以调大一点，动态的Buffer Pool机制能够更好地利用内存，因此我们需要解决Buffer Pool动态伸缩的问题。



Adjust Buffer Pool Size Dynamically By Work Load

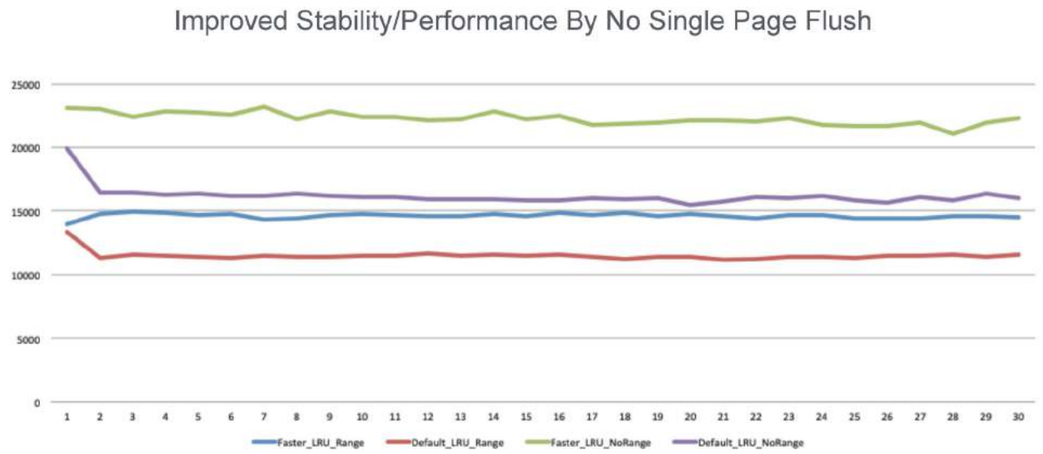
上图为一个测试，绿色线是原生的MySQL，我们在做Buffer Pool缩小的过程中，发现它抖动得非常厉害，很多时候甚至可能会跌到底，我们对这个问题进行深入分析，并进行技术突破。

蓝线是BP动态伸缩的技术下的反馈，虽然也有抖动，但抖动幅度大幅减少，最坏的情况跌20%。这个测试是在业务高峰时段进行，如果在业务不那么繁忙的阶段，我们有信心做到没有抖动。

4. 页面淘汰优化

我们发现，当MySQL在内存紧张时容易引入单页淘汰，也就是当要读一个数据块时，如果在内存里找不到空的数据页，则会淘汰一个数据页。

此时是逐页淘汰，效率非常低下。在这里我们做了一个机制改进，将淘汰一个页的概率降到最低，提升了淘汰页面的效率，用同时淘汰多页代替逐页淘汰，性能得到大幅提升。



上图为一个测试，4条线是2种场景，上面两条线是关闭范围查询（Range Scan）的读写场景。高的线是我们页面淘汰优化之后的线，相比优化前提升了20%。下面两条线是打开范围查询（Range Scan）的情况。由于有些SQL较为复杂，会查询比较多的记录，所以QPS会低一点，我们发现提升幅度也是差不多在15～20%。

数据库数据大于内存是一种常态，因此这里的优化是很大的提升，意味着可以用更低的资源配置得到更高的QPS（获得更高的CPU执行效率），有效节约了成本。

（四）安全性

· 安全性实现了三个功能：

国密(SM4)支持；

防删表保护；

DDL回收站；

1.国密(SM4)支持

由于数据安全风险很大，国家对加密算法也有一定的要求与标准，因此我们增加了对国密SM4算法的支持，这点就不展开详细讲述了。

2.防删表保护

2020年的时候，业内发生数件数据库表被意外删除的事情，阿里云居安思危，为防止此类事情发生，我们设计了一个防删表的功能。

这个功能可以设置删除表的权限，例如需要收到某个用户的指定账号，或者要Super权限，或者需要通过内部管控平台操作，或者要求删除操作只能在本地登录进行，通过不同的权限设置可以起到不同的安全级别效果。

从保护内容方面可以分为两类，第一类是数据类的，包括Drop Table、Truncate Table、Drop Partition、Truncate Partition、Exchange Partition和Drop Tablespace。还有一类是对存储过程、视图以及定义的保护，当启用这个级

别的时候，用户可以禁止他人删除视图的定义、存储过程、触发器，甚至可以禁止删除Binary Log。

多样可选的设置相结合，可以起到很好的数据保护作用，确保用户的数据库安全运行。

3. DDL回收站

出于对DDL方面的考虑，我们做了个DDL回收站。

如上图左边所示，在AliSQL 8.0版本数据库中，我们有一个“recycle_bin”保留数据库名字，字面意思是回收站。

回收站这个概念大家非常常见，DDL回收站的概念与Windows回收站类似，但我们对回收站的权限做了特别的控制，需要有特别的权限才能进行彻底删除。

在AliSQL 8.0中，当开启回收站功能后，用户的删除操作不是直接将表删除，而是将这些表先移到回收站。

上图右边一般来说没有权限，它需要拥有更高的权限之后才可以进行删除操作。通过这样的两级机制，可以有效防止表被删掉。当用户不想开启防删表策略时，可以开启回收站，当用户发生误操作误删表时，可以从回收站快速找回数据。

阿里云开发者电子书系列

微信关注公众号：阿里云数据库
第一时间，获取更多技术干货

阿里云开发者“藏经阁”
海量免费电子书下载