mysql

定位慢查询

超过1s 时间过长

方案1：

调试工具：Arthas

运维工具：Prometheus、**Skywalking**

方案2： 慢日志 （调试阶段）

设置开关

设置慢日志的时间阈值 超过就记录

如何分析慢**SQL**

可以采用EXPLAIN 或者DESC命令获取mysql如何执行select语句的信息

possible_key 可能使用到的索引

key 当前实际命中的索引

key_len 索引占用大小

这两个查看是否可能会命中索引

Extra额外的优化建议

Extra	含义
Using where; Using Index	查找使用了索引，需要的数据都在索引列中能找到，不需要回表查询数据
Using index condition	查找使用了索引，但是需要回表查询数据

下者意味着有优化空间

type sql连接的类型 左往右性能变差

null system const eq_ref ref range index all

null没用到表 无需关注数

system 查询系统中的表

const:根据主键查询

eq_ref: 主键索引查询或者唯一索引查询 （只返回一条数据）

ref:索引查询 等值匹配 多行记录

range:范围查询 适用于 索引范围查询，例如 `BETWEEN`、`>`, `<`, `IN()` 等。

index：索引树扫描

all:全盘扫描

index 和all意味着要优化

索引

全盘查 效率不高

添加索引 提升速度

> *B+*树

相比即存索引又存 记录的 B 树，B+树的非叶子节点可以存放更多的索引，因此 B+ 树可以比 B 树更「矮胖」，查询底层节点的 磁盘 I/O次数会更少。

B+ 树叶子节点之间用链表连接了起来，有利于范围查询，而 B 树要实现范围查询，因此只能通过树的遍历来 完成范围查询，这会涉及多个节点的磁盘 I/O 操作，范围查询效率不如 B+ 树

主键索引的 B+Tree 的叶子节点存放的是实际数据，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子 节点里； 二级索引的 B+Tree 的叶子节点存放的是主键值，而不是实际数据

> 聚集索引和二级索引

(聚集索引）主键查询：直接在 主键索引 所在的 B+ 树中查询，然后直接返回查询到的叶子节点（此时，叶子节点里面就是整行 记录）

如果没有主键和唯一索引 会自动生成隐藏的rowid作为聚集索引

二级索引查询：首先，在普通索引所在的 B+ 树中，查询到待查询记录的 主键； 然后，再根据这些查到的 主键， 执行“主键查询”（即，回表）

> 覆盖查询

覆盖索引（Covering Index）是指 查询所需的所有字段 都已经包含在索引中，无需回表（回到原表中查找数据），从而提高查询效率

比如说二级索引里面 是索引字段+存主键 如果是查字段+主键 那也是覆盖索引 直接从二级索引中返回

不过当查询的数据是能在二级索引的 B+Tree 的叶子节点里查询到，这时就不用再查主键索引查，这种在二级索引 的 B+Tree 就能查询到结果的过程就叫作「覆盖索引」 由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段。

> 超大分页怎么处理

limit分页查询 越往后 分页查询效率越低

分页查询 通常需要排序

分页查询（Pagination Query）是一种按页返回数据的查询方式，适用于大数据集查询，可以避免一次性加载所有数据，减少数据库压力，提高前端加载效率。

`LIMIT` + `OFFSET` 需要 扫描并丢弃 前面的数据，再返回后面的数据。

索引优化有限，MySQL 仍然需要读取数据到内存后再丢弃。 所以很慢 其实就是要接近于全盘扫 扫到限定位置为止

覆盖索引+子查询形式进行优化 本质上是先取需要的主键 再根据主键索引取行 少了 查的过程中所有行都扫一遍

> 索引建立原则

1.量大且查询频繁的表要索引

2.常查询条件 where order by group by 操作的字段建立索引

3.区分度高的列作为索引，尽量建立唯一索引

4.字符串长度长，可以用字段的特点 建立前缀索引

5.尽量使用联合索引 ，因为可以覆盖索引，避免回表

6.索引多也不是好事 删改效率低

7.如果不能存储Null值，要设置not null约束

> 索引失效

如何判断索引失效？ 执行explain

key和**key len** 为**null** 则为索引失效 或者**key_len**长度不对

1.违反最左匹配原则

查询从最左列开始 不能跳过索引中的列

如果是AC这种情况 跳过中间 那只有最左索引生效

2.范围查询的字段的右边的列，索引失效

3.在索引列上进行运算操作，索引失效

4.字符串不加单引号，索引可能失效

发生类型转换就会失效

5.%开头的模糊查询会失效，尾部%模糊查，索引不会失效

sql优化的经验

1.表的设计优化

2.索引优化--->创建原则和索引失效 最左匹配原则

3.sql语句优化

4.主从复制、读写分离

5.分库分表

> 表的设计优化

比如设置合适的类型 数值 内容的长度 节省存储成本

比如字符串类型 定长和不定长 效率

> *sql*语句优化

1.避免使用select *

2.避免索引失效的写法

3.用union all 代替union Union多一次过滤 效率低 union all返回重复的

4.避免在where 对字段进行表达式操作

5.连接优化 能用inner join 就不要用left join right join ，必须要用 一定要以小表为驱动

内连接会自动把小表放外边 但是left和right 不会

`LEFT JOIN` 和 `RIGHT JOIN` 需要返回左表或右表中的所有行，即使没有匹配的记录也会返回 `NULL`，因此它们的计算开销更大，特别是当表的数据量较大时。

> 主从复制、读写分离

master负责写 从节点负责读

事务

> *ACID*原则

原子性：操作同时成功或者同时失败 失败就回滚

一致性：商品库存：当库存减少时，总销售量应同步增加，数据间的约束关系保持一致。

隔离性：并发事务之间相互独立，不能互相干扰。一个事务未提交前，其操作对其他事务是不可见的。

持久性：事务一旦提交，所做的修改将永久保存在数据库中，即使发生系统故障。

> 隔离级别

脏读: 一个事务读到了另外一个事务还没提交的数据 （并发问题 抢跑）

幻读：一个事务查的时候没有 但是插入的时候又发现这行数据已经存在

不可重复读：同一事务内 两次读取同一条数据 但不一致+

读未提交<读已提交<可重复读<串行化

读未提交都解决不了

串行化解决所有问题 但效率低

读未提交：由于事务可以读取其他未提交事务的数据，所以可能会读取到“脏数据”。也就是说，如果一个事务修改了数据，但未提交，另一个事务就可以看到这个修改，但如果第一个事务最终回滚，则第二个事务就会读到无效的数据。

读已提交：事务只能读取 已经提交事务的数据，即它 不能读取未提交的数据。 如果在同一事务中读取相同数据的两次，可能会得到不同的结果。因为在这两次读取之间，其他事务可能已经修改并提交了数据，导致读取到的数据不同。

每次查询时，数据库会加锁当前被查询的行（即读取到的数据），但每次查询完毕后，锁会被释放。因此，其他事务可以在事务A的查询过程中修改其他未加锁的数据行。

可重复读：事务中的 所有查询在事务开始后都能看到相同的数据，即使其他事务提交了数据也不受影响。 隔离级别下，为了保证同一事务在不同时间读取相同数据时不受其他事务影响，数据库会使用 行级锁（或者在某些数据库中使用更严格的锁策略）来 锁定查询的行。幻读通常发生在使用范围查询时（如查询 `age > 30` 的所有记录）。如果在事务A查询时，事务B在其后插入了一行满足查询条件的记录，那么事务A再次查询时，会发现数据集发生了变化。

锁会持续整个事务的生命周期，直到事务提交或回滚

MVCC

> *undo log*和*redo log*的区别

sql语句操作时 先去看

缓冲池 :内存的区域，先去看内存有没有 没有的话再从磁盘加载到内存中

数据页:存储引擎磁盘管理的最小单元 每个页大小默认为16kb 页中存储的是行数据

减少磁盘的IO

脏页 内存的数据没存到磁盘 同步失败 内存数据有可能消失

> *redo log*

redo log buffer存在内存 redo log file 存在磁盘

修改信息存在redo log file 用于在刷新脏页到磁盘 发生错误时，进行数据恢复使用 顺序磁盘IO 性能快

Write ahead logging： 先写日志，再写数据

当 事务提交 或 日志缓冲区满 时，系统会将日志缓冲区中的内容刷新到磁盘上的 **Redo Log** 文件。这个过程是通过顺序写入的方式来提高性能。

只有在 Redo Log 被刷新到磁盘之后，事务才会被视为提交成功。

如果在事务提交后系统崩溃，数据库启动时会扫描 Redo Log，根据日志中的记录 回放 所有已提交事务的操作，直到恢复到崩溃前的状态。 未提交的就失败回滚了

> *undo log*

记录数据被修改前的信息 提供回滚和**MVCC**

undo log 是逻辑日志 delete一条记录 就记录insert记录 记录相反的记录

> *mvcc* 也能控制隔离

维护一个数据的多个版本 使得读写操作没有冲突

依赖于隐藏字段 **undo log**日志 **readView**

隐藏字段：

DB TRX_ID 最近修改事务**ID**

DB_ROLL_PTR 回滚指针 指向这条记录的上一个版本 配合**undo log**

DR_ROW_ID 隐藏主键

undo log日志 配合指针 链表！！！！！！！！

当insert的时候，产生的undo log日志只在回滚时需要，在事务提交后，可被立即删除。

而update、delete的时候，产生的undo log日志不仅在回滚时需要，mvcc版本访问也需要，不会立即被删除。

> *readview:* 快照读*sql*执行时 *MVCC*提取数据的依据 记录和维护系统当前活跃的（未提交）事务*id*

当前读：读取的是记录的最新版本 读取时还要保证其他并发事务不能修改当前记录 会对读取的记录进行加锁 日常操作都是一种当前读

快照读：读取的是可见版本 （无加锁的select) 有可能是历史数据 是非阻塞读

可重复读的第一个**select**语句才是快照读的地方

读可提交 每次**select** 都生成一个快照读

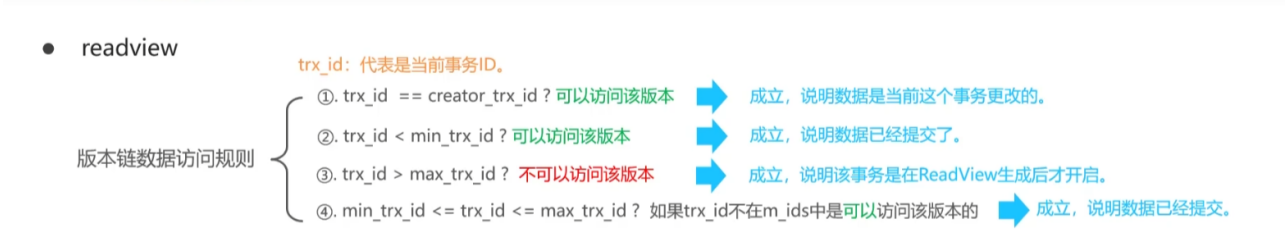
m_ids 当前活跃的事务ID集合

min_trx_id:最小的活跃集合id

max_trx_id：预分配事务ID，当前最大事务ID+1

creator_trx_id:readview创建者的事务ID

trx_id代表当前事务id （在记录的隐藏字段）



简单讲：存每个事务和数据的情况　根据事务和数据的情况来决定读哪个

比对记录的最近事务修改ID

主从同步原理

核心就是二进制日志

定义了所有 DDL和DML　　update insert 之类的 除了查询

从节点的IOTHREAD 读主节点的binlog　然后写到relay log

SQL thread 再读relay log

其实就是缓存那一套 没啥

分库分表

太大就分表呗

> 垂直分库

根据业务将不同表拆分到不同库

跟微服务有关系

> 垂直分表

以字段为依据， 把不同字段拆到不同表中

比如商品的description　跟ES有关系

冷热数据分离 减少IO过渡争抢

> 水平分库

拆成不同库

类似于分片集群　 取模找库

对应redis的 哈希槽 16384

> 水平分表

一个表拆成多个表 （可以在同一个库）

找表也是取模呗

> 新问题

分布式事务一致性问题

跨节点关联查询

跨节点分页、排序函数

主键避重

中间件Mycat 来解决以上问题 （水平分）