SQL审核和优化(面向研发)

前言

本文是阐述研发日常工作中的SQL审核方法，供研发参考。掌握SQL审核背后的原理和方法，有助于提高研发效能，降低应用故障概率。这必将称为研发的必备能力之一。

有关研发数据库开发中的其他问题，请参考 数据库开发自助手册(面向研发) 。

SQL审核只是在iDB里点点按钮，需要几分钟或者几十分钟。做好SQL审核和优化的关键是了解背后的索引、锁、执行计划等原理。同时还要持续的关注线上应用性能和问题，及时修正问题、总结经验。一点点进步。

同时，DBA并不是简单将SQL审核这项工作前推到研发，DBA配套的系统iDB4.0目前正在试用中。一旦你看过，你就会意识到，属于研发自己的数据库优化时代，来临了！😀

目录

线上SQL问题汇总

SQL审核是基于数据库的应用开发流程中的一步，操作上很简单，点击几个“审核通过”按钮，这一步就过去了。
然而，SQL审核没做好的影响是在应用上线后一段时间才会体现。
这里总结一下线上常见的跟SQL有关的问题汇总。

DB主机load高

DB主机load高通常只有DBA通过load告警才会知道。然后DBA通过一些简单分析就可以将load高起因定位到具体的实例上。

目前DBA的系统还不能对load高原因自动定位，将来应该可以。
目前，研发只需要了解一下这个原理，平常配合DBA的请求即可。

Linux系统的load，取自 `/proc/loadavg` 的值。

```
$uptime ; cat /proc/loadavg
 21:19:03 up 48 days,  4:42,  1 user,  load average: 8.30, 8.12, 8.39
8.30 8.12 8.39 20/5331 64254
```

前三个字段分别是 1分钟，5分钟，15分钟内在CPU的run queue里状态为'R'的线程（`nr_running`）以及状态为'D'的线程（`nr_uninterruptible`,多为等待disk I/O返回）的平均数值。每隔5秒钟更新一次。
第4个字段的'/'前面是当前状态为'R'的调度实体数（进程或者线程）；'/'后面是当前系统上所有在运行的调度实体数（进程或者线程）。通常'/'前面的数值小于或者等于所有CPU的个数（包括超线程）。
有关load诊断方法请参见 tsar load模块介绍 。

所以，load高常有两个原因：**CPU很忙（CPU的 `usr+sys` 利用率很高）**或者**CPU在等待IO（IO利用率很高）**。

CPU利用率很高

通常导致DB主机CPU利用率很高的进程都是mysqld进程。这种情形在 `top` 命令中可以很明显的看出是哪个 `mysqld` 进程的CPU利用率很高。然后根据进程ID找到实例。

mysql实例耗费CPU，其原因常是实例的"总逻辑读"很高。具体有两种可能：一是突然出现某些逻辑读很高的SQL；另外一种就是SQL的并发突然增加。实际情况往往是二者都有，各自比例不一样。
"一个逻辑读"指的myslq从 `Buffer Pool` 里取一次数据的IO操作。如果数据没有命中，则会从磁盘上读取数据。后者叫"一个物理读"。然而，mysql并没有很好的指标展示一个sql的"逻辑读"是多少。这里先理解这个概念的存在即可。

虽然无法准确度量一个SQL的"逻辑读"是多少，但是知道"索引"可以降低SQL的"逻辑读"，因为"索引"影响的是数据的读取方式。后面 索引原理和实践经验 会详细介绍这个。

如果 `top` 命令中没有发现有进程的CPU利用率异常，则load高可能是因为在等待IO。

IO利用率很高

上面说到，一个"逻辑读"可能会引发一个"物理读"。"物理读"的性能取决于磁盘的响应能力。
在传统机械盘时代，最好的SAS盘，一个IO响应时间正常水平是10ms以内，IOPS可以做到150-200.在SSD出现后，SAS逐步被淘汰了。一个SSD的IO响应时间正常水平是1ms以内，IOPS能力最高可达6-7W左右，吞吐量可达1GB。这些是泛泛的说，准确的值跟IO大小、读写比例、SSD品牌有关，以实际测试为准。
有关linux下io的诊断方法请参见 tsar io介绍 。

SSD的原理跟机械盘不同，但是linux系统计算IO利用率的方法还是基于传统机械盘，所以SSD的IO利用率仅供了解SSD当前的工作负载。利用率高并不一定有问题，主要是看是否引起数据库的性能下降。这是第一个判断因素。第二，就是要看看这种高是否合理。

当IO利用率高影响到数据库的性能时，多半是IO的吞吐量达到瓶颈了，接近1GB.而其原因可能是备份进程异常（注：数据库备份是使用xtrabackup的流式备份，受带宽限制，正常情况下备份IO吞吐量不会很大。），或者就是有实例有很多SQL产生了很高的IO请求。后者说的就是有慢SQL，且并发还很高。
同理，索引也可以降低SQL的"物理读"。

总结：
无论是CPU忙还是IO忙，DB主机load很高的时候，对主机上实例都可能有一个影响就是导致SQL rt增加。这种情况需要及时定位原因，消除原因，以免引发故障。
研发同学如果碰到DBA要确认某个SQL的业务需求、使用场景等信息时，多半就跟这个问题有关，需要及时配合DBA一下。

TDDL连接池连接耗尽

TDDL连接池连接耗尽问题只有研发通过告警或者应用日志才能发现。

如果研发对应用的Druid连接池配置了监控，在性能出现问题时，很有可能会收到类似下面的连接池满的告警：

```
19:33:16 [C] et2-gamecenter[ET2] gamecenter-002.et2 Alimonitor_DruidDataSourceStat (UR
```

此时，应用的日志里也很大可能看到类似下面这种无法从连接池获取连接的错误：

```
com.taobao.tddl.common.utils.logger.slf4j.Slf4jLogger.warn(Slf4jLogger.java:62)ERR-CO
runningSqlCount 1 : /* 0ab2d68e14603564891386840e/0// */select `a1`.`sf_sku_id`,`a1`.
```

研发同学通常会误认为DB连接数满了，从而在初期将问题的分析引向错误的方向，耽误了一点时间。
实际上，电商的alisql实例对连接数的硬限制在3w。

```
root@(none) 10:23:15>show global variables like 'max_user_connections';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| max_user_connections | 30000 |
+----------------------+-------+
1 row in set (0.00 sec)
```

除非应用服务器非常多，并且tddl连接池配置的最大连接数不合理，这个连接数通常是很难达到的。

上面报错的信息都是因为TDDL连接池里的活动连接达到了连接池允许的最大连接数。这个最大连接数就是 `maxpoolsize` 。实例创建的时候会给一个默认值，DBA可以修改。

补充：
TDDL 是中间件团队的产品，有效的解决了mysql分库分表给应用读写表带来的难题。详情参见 TDDL新生入门指南 ， TDDL问题排查 。
研发人员需对TDDL有初步了解，有疑难问题请联系 TDDL接口人：梦实 。 DBA对TDDL的解释仅供参考。

连接池的连接满了，有多种可能原因：
一是应用没有正确的使用连接。如没有释放连接（把连接归还给连接池）；在循环体中申请释放连接；异常逻辑里没有及时释放连接等等。这些都是低级错误，在成熟的研发团队里已经见得很少。
二是单个连接的活跃时间变长。可能原因有：有慢SQL；或者SQL被其他会话阻塞（申请不到锁）；或者事务里有非SQL操作（同步等待其他接口返回；或者有文件IO或网络IO操作等，也属于低级错误。）导致事务时间变长（也间接导致连接活跃时间变长）。
三就是业务请求增多。如正常的业务增长或者促销活动，或者业务被攻击等等。

总结：
通常这类问题，首先看TDDL连接池配置的最大连接数是否合理。这个要考虑应用服务器的数量，然后再计算每个应用实例可以拿到的最多连接数。对于分库分表的实例，想必应用服务器很多，这个要认真计算。对于单库的实例，iDB默认生产的实例的TDDL配置里的最大连接数通常显得过小。研发同学在碰到这类问题时，可以首先请DBA提高这个值。
然后，再查是否有慢SQL导致连接平均活跃时间变长了。除去那些低级错误外，就是跟 索引、锁有关的了，后面会详细介绍。

SQL变慢

具体SQL变慢这种问题通常是研发通过“鹰眼”或者应用日志发现，DBA首先只能看到mysql实例整体SQL rt平均水平的抖动，然后才可能去定位具体的慢SQL。

应用的db请求时间分为网络上来回消耗的时间加上db内部请求处理时间。同机房内部或者同城机房之间，网络延时都非常低，出问题概率很低。但是突袭演练时故意触发网络交换机丢包，可能会增加这个延时。另外，跨城域的机房之间的网络延时相对还是有点高，出问题的概率也相对高一些。
网络运营团队对于点对点之间的网络延时没有好的监控手段，导致往往落后于业务应用发现请求延时增加。而应用发现跨单元的请求时间增加的时候往往首先就怀疑是DB问题，个别时刻这种判断也会导致走了弯路。不过大部分时候，这种怀疑都是正确的。这里只说DB导致SQL变慢的原因。（网络的延时使用mtr命令即可确认。）

前面load高问题，以及TDDL连接池满的问题也都跟SQL变慢有关。首先是找出变慢的SQL。研发可以通过鹰眼反馈信息，DBA可以通过慢查日志找线索。二者都不能保证一定能发现问题，经常需要互相结合。
分析慢SQL问题总是关注两个方面，一是SQL的并发数；二是SQL的执行计划。并发越高，慢SQL的负面影响越严重,被发现的概率也越高。并发高的原因只有研发才能定位原因，DBA只能采取限流措施降低并发。而SQL的执行计划分析则是DBA和研发都可以分析干预的。后面会重点介绍执行计划。

慢SQL还有一种可能性不是SQL性能变差，而是跟阻塞有关系。见下文。

阻塞（`blocking`）、锁超时（`lock wait timeout`） 或 死锁（`deadlock detected`）

锁的问题通常是研发通过应用日志发现。
“锁超时”异常类似下面这种

```
java.sql.SQLException: Lock wait timeout exceeded; try restarting transaction
```

”死锁问题“通常应用日志或者db的alert.log里都会有 `deadlock` 字眼。

```
transactions deadlock detected
```

“阻塞问题”就是SQL在申请一个锁过程中出现等待，但是还没有等待超时，所以应用日志很难发现，表现为该SQL变慢，通过执行时间可以发现线索。在DB端，除非及时的查看mysql会话（`show full processlist`），事后是难以确认过去某个时间点是否有阻塞现象。如果被阻塞的时间不长（毫秒到秒级别，相对于人的反应时间而言是很短，想对于应用的需求时间很长），DBA也是很难发现阻塞问题。

排查锁的问题，如果能提供发生阻塞或者死锁的sql话，可以缩小排查的范围。
然后就排查所有更新该表的业务场景，查看事务里是否有非数据库操作导致的等待，事务里更新表的顺序是否一致，以及所有的更新操作是否是根据主键或者唯一键操作的。后面 锁的原理和锁问题总结 会介绍分析方法。

>> 回到目录

索引原理和实践经验

首先，在上线之后才添加索引，是一种错误的开发模式。这会导致事后要花费DBA和研发更多的时间去弥补这个错误。
其次，没有充分思考就为多个列建立单列索引，以及为所有查询场景建立很多联合索引，也是不负责任的。在事后处理上这个比上面那个场景更为棘手。因为删除索引的风险高于增加一个索引，没有人敢为删除索引拍板。

索引原理

索引结构

- 聚集索引和非聚集索引

  mysql的表是聚集索引组织表，聚集规则是：有主键则定义主键索引为聚集索引；没有主键则选第一个不允许为NULL的唯一索引；还没有就使用innodb的内置rowid为聚集索引。
  非聚集索引也称为二级索引，或者辅助索引。

  mysql的索引无论是聚集索引还是非聚集索引，都是B+树结构。聚集索引的叶子节点存放的是数据，非聚集索引的叶子节点存放的是非聚集索引的key和主键值。B+树的高度为索引的高度。

  强烈建议表都要有主键且以数字型为主键。

- 索引的高度

  聚集索引的高度决定了根据主键取数据的理论IO次数。根据非聚集索引读取数据的理论IO次数还要加上访问聚集索引的IO次数总和。实际上可能要不了这么多IO。因为索引的分支节点所在的Page因为多次读取会在mysql内存里cache住。
  mysql的一个block大小默认是16K，可以根据索引列的长度粗略估算索引的高度。

  以下表为例：

  ```
  1 | create table tab(id int primary  key,c1 int,index(c1),c2 varchar(128))
  ```

  聚集索引长度：4字节；非聚集索引c1长度:4字节；指针长度：8字节。平均行长以200字节计算。Page使用最大比例70%。
  则：每个Page包含聚集索引平均行数：16384 * 70% / 200 ≈50

每个Page包含非聚集索引平均行数：16384 * 70% / (4+8) ≈1000

则聚集索引的高度和行数关系粗略为：

索引高度	聚集索引最大记录数	非聚集索引最大记录数
2	1000*50 = 50000	1000*1000 = 100w
3	(1000)2*50 = 3500w	(1000)2*1000 = 10亿

总结：

1. 索引的高度控制在3以内（含3）最合适，所以单表的记录数不要特别大。
2. 索引列的总长度越长，索引的高度可能越大。SQL的性能就越差。

索引使用成本

- 空间成本

  索引也占空间，大表的索引包含的列很多时候，这个空间也不可忽略。

- 时间成本

  索引用的好是可以节省时间。但是用的不好，也会浪费时间。

  这里要注意两个概念：**扫描行数**和**返回行数**。
  在innodb层通过索引检索到的记录数称之为“扫描行数”。但是不是所有的记录都是查询想要的，扫描的数据返回到mysql server层时还要经过 `filter条件` 过滤掉一部分，然后才返回给客户端。最后的记录数称为“返回行数”。
  很多时候就是扫描的行数很高导致走索引的时间成本很高，而返回的行数低，就是让人迷惑的地方。

- 更新成本

  二级索引很多时，insert/update/delete的内容只要跟索引列有关系，在更新记录的同时还需要更新记录相关的二级索引，这可能会增加很多额外的IO。

  所以，只有利大于弊的时候，才会使用索引。

`ICP` (Index Condition Pushdown) 介绍

ICP是mysql 5.6新增的一个特性。先看看在没有这个特性之前，mysql的单表查询是什么过程。

这个图描述的是根据一组条件查询一个表的过程，其中不是所有where条件都在索引里。mmysql首先在

innodb层通过索引扫描定位到具体的记录，然后第4步**回表**获取全部字段的值，在第6步返回给Server层。Server层在第8步时通过where种的其他条件过滤结果集，然后返还给客户端。（这个图也说明“扫描行数”和“返回行数”不同的过程）。

mysql 5.6对这个过程做了一个优化，就是在第4步的时候提前将一些where条件在这里对扫描返回的结果集进行过滤，从而减少**回表**的行数。如下图。

ICP的意义首先是减少了**回表**次数，降低了时间成本，对buffer pool的使用也有帮助。
同时这一优化也减少了在DML走二级索引时，lock的持有范围和时间，降低阻塞和死锁的概率。

不过，ICP的使用条件很苛刻：
1. 只能用于二级索引。
2. 只能用于单表的索引作为filter条件下推，不适用多表连接的连接条件下推。
3. 索引操作类型是 `EQ_REF` / `REF_OR_NULL` / `REF` / `SYSTEM` / `CONST` 才可以使用ICP（后面介绍索引操作类型）。
4. 只能是Range操作， `ALL` / `FT` / `INDEX_MERGE` / `INDEX_SCAN` 不行。

前缀索引

前缀索引指索引的列有varchar字段且只索引了其开头部分长度的值。
在mysql里，前缀索引的最大长度在不同字符集下是不一样的。gbk下是383，utf8下是255，utf8mb4下是191.

前缀索引的优点是节省索引空间，缺点是不支持ORDER BY 和GROUP BY。

索引使用场景介绍

适合返回很小比例的数据

主键索引的唯一键索引的查询场景都很简单，适合根据主键列和唯一键列做等值查询或者INLIST 查询（个数不要特别多即可）。
如果不是等值查询而是范围查询，则mysql会评估扫描的行数占总数据量的多少来决定是否走PK或者UK。不建议这么用，查询很频繁的话，建议最好是加limit限制以下记录数。

如果一组条件从大表里扫描的数据量占比“很小”，则这组条件列适合建联合索引。 这个“很小”通常指5%以下尚可，1%以下更好。千万级的表则要0.1%以下等等。

要扫描很小量的数据，就要求条件列组合的“NDV”（ `Number Distinct Value` ）很高。mysql也会对索引列的这个NDV给出一个估算值，称为“Cardinality”。
索引的“选择性”等于“Cardinality”值除以估算总记录数。选择性越高，索引的作用越大。

如上例子，mysql估算的Cardinality值跟实际的NDV值可能有些出入，但总体是对的。像这个索引 `idx_st_bt(status,biz_type)` 通常就不适合做索引。**除非**查询sql所查的数据是固定的查那种占比很小的数据。否则，一旦查询要扫描的数据比例很大，走了这个索引，当表特别大的适合，那时间成本会增加很多。

单列索引和联合索引

两个单列索引 `idx_c1(c1)` 、 `idx_c2(c2)` 不等同于联合索引 `idx_c1_c2(c1,c2)` 。同时，联合索引里的列的顺序变动，索引适用的场景也会不同。这是常见的适用误区。

由B+树索引结构知，联合索引(c1,c2,c3)能够覆盖的查询条件有：
1. (c1 = ? and c2 = ? and c3 = ?) (备注：全列匹配)
2. (c1 = ? and c2 = ? and c3 in (?,?,?)) （备注：最左前缀匹配）
3. (c1 = ? and c2 in (?, ?, ?)) （备注：最左前缀匹配，索引覆盖）
4. (c1 = ?) （备注：最左前缀匹配，索引覆盖）
5. (c1 in (?,?,?))

但是联合索引(c1,c2,c3)不能适用于不带c1的查询条件组合，如：
1. (c2=? and c3=?) 或者 (c2=? and c3 in (?,?,?))

对于这种查询，如果次数很多或业务重要，需要单独建联合索引 `idx_c2_c3(c2,c3)` .

补充：
1.对于联合索引(c1,c2,c3)适合的5种查询场景举例，越往后，其时间成本越高。因为扫描的行数会越多。当表很大的适合，其索引空间也很大，这种扫描成本也不可忽略。当然，mysql 5.6的ICP特性对此有些优化。
2.最好的优化建议是让条件尽可能的具体，让“扫描行数”和“返回行数”尽可能的贴近。
3.联合索引最好是以NDV最大的列打头，或者结合查询频率最高的条件综合决定。

排序场景

从索引结构还可见索引的记录是按某种顺序排序的，所以，有时候使用索引可以减少排序IO。
利用索引避免排序，这个原则必须是在前面所有场景都不合适的情况下才选用，不能跟其他原则相冲突。这是一个很容易被滥用的原则，导致最后得不偿失。

对于一些分页查询场景，如果没有其他更好的索引，在不引起其他查询异常的情况下，可以将排序字段加入到索引列中。

如索引idxc1c2(c1,c2) 可以应对这种场景：(c1=? order by c2 limit ?,?), (order by c1, c2)

索引使用案例

分页排序走错索引

前面说的索引可以规避排序，就有一个误用的例子。

- 天猫无线后台服务中的评论分页查询走错索引

表结构：

```
CREATE TABLE `fun_comment` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `gmt_create` datetime NOT NULL COMMENT '创建时间',
  `gmt_modified` datetime NOT NULL COMMENT '修改时间',
  `text` varchar(500) DEFAULT NULL COMMENT '评论内容',
  `subject_id` bigint(20) unsigned NOT NULL COMMENT '主题id',
  `images` varchar(4096) DEFAULT NULL COMMENT '图片列表',
  `valid` tinyint(3) unsigned NOT NULL DEFAULT '1' COMMENT '是否有效0表示无效，1表示有效'
  `user_id` bigint(20) unsigned NOT NULL COMMENT '评论人用户id',
  `parent_id` bigint(20) unsigned NOT NULL DEFAULT '0' COMMENT '被回复的评论id',
  `source` varchar(20) NOT NULL COMMENT '评论的来源',
  `url` varchar(1024) DEFAULT NULL COMMENT '评论整块区域的跳转地址',
  `device_info` varchar(1024) DEFAULT NULL COMMENT '设备信息',
  `app_name` varchar(20) NOT NULL COMMENT '应用名',
  `source_id` varchar(100) NOT NULL COMMENT '关联的外部id',
  `status` tinyint(4) NOT NULL DEFAULT '1' COMMENT '状态，1-审核通过，0-未审核，-1-审核不
  `type` tinyint(3) unsigned NOT NULL DEFAULT '0' COMMENT '0表示普通评论，1表示弹幕',
  `floor` int(10) unsigned DEFAULT '0' COMMENT '楼层，从1开始增长',
  `audit_time` datetime NOT NULL COMMENT '评论审核时间',
  PRIMARY KEY (`id`),
  KEY `idx_gmtcreate` (`gmt_create`),
  KEY `idx_audit_time(`audit_time`),
  KEY `idx_subject_id` (`subject_id`),
  KEY `idx_search` (`subject_id`,`status`,`source`,`user_id`),
  KEY `idx_app_sourceid` (`app_name`,`source_id`,`type`,`status`,`user_id`,`audit_time
) ENGINE=InnoDB AUTO_INCREMENT=1477611 DEFAULT CHARSET=utf8mb4 COMMENT='评论'
```

SQL为：

```
SELECT *
  FROM fun_comment
 WHERE app_name= 'tlive'
   and source_id= '3a9fb3f1-5c90-4a95-9fdd-ea88f9af9435'
   and status= 1
   and type in(0, 1, 2)
   and user_id in(840538579, 2360533880, 2621625676)
   and audit_time<= now()
order by audit_time desc limit 50
```

因为 `audit_time` 列上有个索引，SQL里根据 `audit_time` 分页排序，结果MySQL选择了索引 `idx_audit_time`，导致SQL很慢。

解决办法：
研发确认业务没有根据 `audit_time` 列作为条件的查询，于是按建议删除了索引 `idx_audit_time`。
该SQL改走了索引 `idx_app_sourceid`。
（当然，走丢索引后，该SQL查询仍然要1~8ms左右。其原因是NDV最高的 `user_id` 条件可有可无，且还是in条件，不适合做联合索引的最左列。业务只能接受这个慢的结果。）

索引减少排序案例

索引结构：

SQL例子：

where 条件发生隐式类型转换导致使用不了索引

有些列名字叫ID类的，实际类型却是个字符型，存储的又是数字型。粗心的研发很容易以为就是数字型，SQL里直接用数字给该列赋值，导致不能使用列上的索引。

解决办法：
同一业务含义的字段其定义（类型、长度）一定要保持一致。
如果做不到一致，只能将等号的一边用函数做显示的类型转换，以期使用另外一列上的索引。

where 条件的列上有表达式导致使用不了索引

这种错误场景有：
1. datetime列，条件里想查某一天的数据，使用了函数 `date_format(gmt_create, '%Y-%m-%d') = '2016-05-08'`
2. varchar列，条件里做了大小写转换，如 upper(status) = ‘ACTIVE’

解决办法：
1. 去掉函数，改为 大于当天0时0分0秒，小于次日0时0分0秒。
2. 表结构设计上用规范保持统一，如统一为小写或者大写，SQL里就不需要这么纠结了…

本节总结：
索引就两个要点：一是理解索引的B+树结构，二就是理解聚集索引、非聚集索引、组合索引、前缀索引在取值方法上的相同点和不同点。
此外，索引不仅影响的是数据的读取方法，同时还间接影响了数据的加锁方式。

>> 回到目录

锁的原理和锁问题总结

MySQL数据库并发读写数据时，借助“锁”来达到这样的目的：
1. 每个会话都能以一致的方式读取和修改共享数据
2. 尽可能的提高共享数据的并发访问数

在不同的事务隔离级别下，这个目标会有些许差异。电商MySQL数据库默认的事务隔离级别是RC(`Read-Committed`)。这个隔离级别下，每个会话只能读取其他会话的已提交的数据（最新的数据）。因此，在RC隔离级别下，一个事务在开始和结束期间，可能会读到其他事务新增加的数据（幻想读）或者发现某笔数据被修改（不可重复读）。
本文不讨论其他事务隔离级别下的锁行为。 有兴趣的请参见圭多写的MySQL 加锁处理分析

锁的原理和加锁场景

MySQL锁的理论是在遵循SQL标准上逐步发展出自己的特点的。虽然跟Oracle的“锁”技术相比还是LOW了点，不过也有不少创新，值得学习。

注意：以下所介绍的原理都是在 `Read-Committed` 这个事务隔离级别下的，其他隔离级别下，个别锁的行为可能会不一样！

共享锁和排他锁

InnoDB支持两种标准的行级锁：共享锁（S Lock，允许读取行数据）和排他锁（X Lock，允许修改行数据）。
当该行上面已经被一个事务加了一个锁时，另外一个事务想要加锁能否成功取决于前后两个锁的类型是否“兼容”。实际上只有（S Lock，S Lock）是兼容的，其他组合（ `S Lock` ， `X Lock` ），（ `X Lock` ， `S Lock` ），（ `X Lock` ， `X Lock` ）都是不兼容的。此时后者必须等待，直到超时（即报： `lock wait timeout` ）。

兼容性	`S`	`IS`	`X`	`IX`
`S`	YES	YES	NO	NO
`IS`	YES	YES	NO	YES
`X`	NO	NO	NO	NO
`IX`	NO	YES	NO	YES

意向共享锁和意向排他锁

Innodb支持多粒度锁定，允许行级锁和表级锁同时存在。意向锁就是表级锁。
意向共享锁（IS Lock，事务想要获取一个表中某几行的共享锁），意向排他锁（IX Lock，事务想要获得一

个表中某几行的排他锁）。
意向锁不会阻塞除全表扫描外的其他请求。

记录锁细分

Innodb针对行锁的算法，还细分为：
间隙锁（`Gap Lock`）：不锁记录，仅仅锁记录前面的Gap
记录锁（`Record Lock`）：只锁记录，不锁记录前面的Gap，记录形式为：`lock_mode X(或S) locks rec but not gap`。
`Next Key锁`：锁住记录，同时锁住记录前面的Gap，记录形式为：`lock mode S(或X)`
插入意向锁(`Insert Intention锁`): insert的时候，如果发现位置下一项上面有Gap锁，则获取这个锁。可以简单理解为`Gap 锁`上打了个`Insert Intention的标记`。

兼容性	RK	GK	IK	NK
RK	NO	YES	YES	NO
GK	YES	YES	YES	YES
IK	YES	NO	YES	NO
NK	NO	YES	YES	NO

锁的实现方式、隐式锁和显示锁

InnoDB的行锁具体是指B-Tree的记录锁（`Record Lock`），也可以理解为是索引的锁。
在`Read-Committed`隔离级别下，没有间隙锁（`Next-Key Lock`或`Gap Lock`）。update的时候如果涉及到二级索引，也会对二级索引加锁。

如果一条记录没有多个事务并发更新的场景，当一个事务修改记录的时候也要对涉及到的索引行加锁，但却没有什么意义。MySQL对此做了一些优化。在事务修改记录时，只是把事务ID（`trx id`）记录在聚集索引的B-Tree叶子的行记录上，而不真正的对该行记录加锁（`Record Lock`），这个叫“隐式锁”，大大降低了锁的数量，节省了CPU资源。
此时当有另外一个事务要对该行记录加锁时，发现该B-Tree叶子节点的行上已经有一个事务ID（`trx_id`）并且其状态是活动的（未提交），则为该已知事务(`trx_id`记录的事务）加锁（`X Lock`），即“隐式锁”变为”显示锁”。这是一种延迟加锁技术，大大降低了锁的数量和锁冲突的概率。
二级索引的“隐式锁”技术细节稍微复杂点，就不说了。

MVCC，一致读和当前读

上面共享锁和排他锁是不兼容的，因此读写会互相阻塞。MySQL为了降低这个影响，支持MVCC（`Multi-Version Concurrency Control`）协议。
在MVCC中，读操作分为“快照读”（`Snapshop Read`）和“当前读”（`current read`）。

“快照读”也称为“一致性非锁定读”（`consistent nonblocking read`）.其原理如下图：

这个读操作碰到当前行被加锁（`X Lock`）时，不受兼容性影响，不需要等待`X Lock`释放，而选择从Undo里读取该行最近的已提交版本的数据。
简单的select查询（不带`for update`或者`lock in share mode`语句）都是“快照读”。这种方式极大的提升了数据读取的并发性。

“当前读”也称“锁定读”，读取的是记录的最新版本,并且,当前读返回的记录,都会加上锁（`S Lock`或`X Lock`）,保证其他事务不会再并发修改这条记录.
使用“当前读”的场景有：
“select * from table where ? lock in share mode”，加的是`S Lock`。
“select * from table where ? for update”，加的是`X Lock`。
上面2个SQL必须在事务里，当事务提交时，锁也就释放了。所以，必须显示开启事务（`set autocommit=0`或`begin;`）
“insert into table values (…)”、“update table set ? where ?”、“delete from table where ?” 都会在更新数据之前先触发一次“当前读”，并加`X Lock`。原理如下图：

所以，MVCC只对SELECT有效。对于UPDATE/INSERT/DELETE无效。

半一致读（`semi-consistent`）

上面说MVCC只对select有效，UPDATE的时候读取的当前数据，如果数据在其他事务没有提交，该UPDATE还是会被阻塞。于是MySQL又对这个做了个优化。这就是`semi-consistent`技术。
前提是事务隔离级别是`Read-Committed`或者`innodb_locks_unsafe_for_binlog`为True(注意：电商这个参数为OFF，不会修改的)

UPDATE的时候，MySQL会判断一下是否可以用`semi-consistent`，判断方法：UPDATE需要加锁，且UPDATE的走的是聚集索引的`Range Scan`或者`ALL`（全表扫描）。如果不满足，则加锁等待；如果满足，则用“一致读”读取该行数据的最新提交版本，然后根据是否符合where条件决定是否加锁。

优点：减少了更新同一行记录的冲突，减少锁冲突；可以提前释放锁，减少进一步锁冲突。
缺点：对binlog不安全（执行顺序和提交顺序不同，影响备库结果，不是完全的冲突串行化结果）

更深入的分析请参见圭多的MySQL+InnoDB semi-consitent read原理及实现分析 。

这个优化的理解有点难，了解即可。只需要谨记，业务的UPDATE操作尽量根据聚集索引去做。

阻塞（`Blocking`）分析

阻塞原理

首先要澄清的是“阻塞”，表示事务在申请锁的过程中等待其他事务释放锁，表现为“等待”，SQL的rt会增加。而研发有时候会说成是“死锁”。

阻塞的原因分析首先是排查事务使用方法是否正确。每个事务时间要在满足业务需求的情况下尽可能的短小，以减少事务之间锁冲突的概率。常犯的错误是在事务中有IO等待或者网络等待，或者夹杂着查询SQL，而该SQL性能还不是很好。
其次，看看事务更新表的条件走聚集索引还是二级索引。如果是根据二级索引更新记录，则会对二级索引中的行记录，以及扫描到的记录在聚集索引中的行记录加锁。如果有ICP，则会在返回记录给Server层之前，释放不符合filter条件的记录上的锁。

lock查看

由于InnoDB的延时加锁技术，只有在发生阻塞的时候，才可以通过 `information_schema.innodb_locks` 和 `innodb_lock_waits` 查看阻塞相关的锁信息。而lock wait的最长等待时间是5秒（有参数 `innodb_lock_wait_timeout` 决定），所以，给阻塞问题排查增加了些不便。

另外，通过 `SHOW ENGINE INNODB STATUS` 可以查看事务持有的锁信息。这个输出内容很多，需要耐心解读。这里就不展开了。

详细诊断经验参见SHOW INNODB STATUS walk through

阻塞示例

示例1:

```
create table t01(id bigint not null primary key, c1 int not null, t1 datetime );
insert into t01 values(1,100,now()),(2,200,now()),(3,200,now()),(4,400,now()); commit;

#1#2: SET tx_isolation='READ-COMMITTED'; SET innodb_lock_wait_timeout=60; SET autocomm
#1: update t01 set t1=now() where id=2; /* 持有隐式锁: X RK(PRIMARY, 2) */
#2: select * from t01 where c1=200 order by id desc limit 1 for update; /* 持有隐式锁: 
#2: select * from t01 where c1=200 order by id limit 1 for update; /* 请求: x RK(PRIMAR
```

lock report:

```
------- TRX HAS BEEN WAITING 3 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1799730 page no 3 n bits 72 index `PRIMARY` of table `test`.`t01
Record lock, heap no 3 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
 0: len 8; hex 8000000000000002; asc         ;;
 1: len 6; hex 0000264b39d0; asc   &K9 ;;
 2: len 7; hex 61000001f829f3; asc a    ) ;;
 3: len 4; hex 800000c8; asc     ;;
 4: len 5; hex 9999794567; asc   yEg;;

```

示例2:

```
create table t01(id bigint not null primary key, c1 int not null, t1 datetime,key idx_
insert into t01 values(1,100,now()),(2,200,now()),(3,300,now()),(4,400,now()); commit;

#12: SET tx_isolation='READ-COMMITTED'; SET innodb_lock_wait_timeout=60; SET autocommi
#1: insert into t01 values(5,500,now()); /* 持有：X RK(idx_c1_t1, (500, '2016-05-28 20:
#2: insert into t01 values(6,500,now()); /* 持有: X RK(idx_c1_c1, (500, '2016-05-28 20:
#1: update t01 force index(idx_c1_t1) set t1=now() where c1=500 and id in (5); /* 持有:
#2: update t01 force index(idx_c1_t1) set t1=now() where c1=500 and id in (6); /* 持有:
#1: /*success */
```

deadlock report:

```
2016-05-28 20:28:49 2ba7ddd01700
*** (1) TRANSACTION: #1
TRANSACTION 642464267, ACTIVE 35 sec fetching rows
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1184, 3 row lock(s), undo log entries 1
MySQL thread id 40683, OS thread handle 0x2ba7ddc80700, query id 164213 127.0.0.1 root
update t01 force index(idx_c1_t1) set t1=now() where c1=500 and id in (5)
*** (1) HOLDS THE LOCK(S):
RECORD LOCKS space id 1799731 page no 4 n bits 80 index `idx_c1_t1` of table `test`.`t
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 800001f4; asc     ;;
 1: len 5; hex 999979470e; asc   yG ;;
 2: len 8; hex 8000000000000005; asc         ;;

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1799731 page no 4 n bits 80 index `idx_c1_t1` of table `test`.`t
Record lock, heap no 7 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 800001f4; asc     ;;
 1: len 5; hex 9999794714; asc   yG ;;
 2: len 8; hex 8000000000000006; asc         ;;

*** (2) TRANSACTION: #2
TRANSACTION 642464268, ACTIVE 29 sec starting index read, thread declared inside InnoD
mysql tables in use 1, locked 1
3 lock struct(s), heap size 360, 2 row lock(s), undo log entries 1
MySQL thread id 40695, OS thread handle 0x2ba7ddd01700, query id 164235 127.0.0.1 root
update t01 force index(idx_c1_t1) set t1=now() where c1=500 and id in (6)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 1799731 page no 4 n bits 80 index `idx_c1_t1` of table `test`.`t
Record lock, heap no 7 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 800001f4; asc     ;;
 1: len 5; hex 9999794714; asc   yG ;;
 2: len 8; hex 8000000000000006; asc         ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1799731 page no 4 n bits 80 index `idx_c1_t1` of table `test`.`t
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 800001f4; asc     ;;
 1: len 5; hex 999979470e; asc   yG ;;
 2: len 8; hex 8000000000000005; asc         ;;

*** WE ROLL BACK TRANSACTION (2)
```

死锁（`Deadlock`）分析

死锁原理

“死锁”就是两个或两个以上事务发生互相“阻塞”时，MySQL自动Kill掉其中一个最小的事务（事务会回滚，释放锁），以让其他事务不再被阻塞下去。

下列场景常常发生死锁：
1. 不同事务会话，以相反的顺序对多个表的记录进行加锁
2. 同一表的同一索引上，不同事务会话以相反的顺序加锁多行记录
3. 同一表，一个会话先通过聚集索引找到记录加锁，并更新二级索引列，对二级索引加锁，另外一个会话通过二级索引更新记录对二级索引和聚集索引行加锁。
4. 同一表，不同会话的UPDATE或DELETE根据不同的二级索引更新多笔记录，导致对聚集索引行记录加锁顺序不同

死锁分析

“死锁”是一个瞬时的事件，当发生“死锁”时，它就已经结束了。唯一的线索就是MySQL的alert.log里关于 `deadlock` 的相关告警内容（是死锁发生时的 `SHOW ENGINE INNODB STATUS` 的输出结果）。另外，应用的日志里也会有报错信息，含字眼 `deadlock` 。

死锁示例

示例3:

```
create table t01(id bigint not null primary key, c1 int not null, t1 datetime,key idx_
insert into t01 values(1,100,now()),(2,200,now()),(3,200,now()),(4,400,now()); commit;

#1#2: SET tx_isolation='READ-COMMITTED'; SET innodb_lock_wait_timeout=60; SET autocomm
#1: update t01 set t1=now() where id=2;   /* 持有: X RK(PRIMARY, 2) */
#2: update t01 set t1=now() where c1=200 order by id desc;  /* 持有: X RK(PRIMARY, 3),
#1: update t01 set t1=now() where id=3; /* 持有: X RK(PRIMARY, 2) 请求: X RK(PRIMARY, 3
```

deadlock report:

```
2016-05-28 20:42:46 2ba7ddc80700
*** (1) TRANSACTION: #2
TRANSACTION 642464409, ACTIVE 12 sec fetching rows
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1184, 4 row lock(s)
MySQL thread id 40695, OS thread handle 0x2ba7ddd01700, query id 164887 127.0.0.1 root
update t01 set t1=now() where c1=200 order by id desc
*** (1) HOLDS THE LOCK(S):
RECORD LOCKS space id 1799733 page no 3 n bits 72 index `PRIMARY` of table `test`.`t01
Record lock, heap no 4 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
 0: len 8; hex 8000000000000003; asc         ;;
 1: len 6; hex 0000264b3a77; asc    &K:w;;
```

```
 2: len 7; hex c7000033050134; asc    3  4;;
 3: len 4; hex 800000c8; asc     ;;
 4: len 5; hex 9999794a96; asc   yJ ;;

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1799733 page no 3 n bits 72 index `PRIMARY` of table `test`.`t01
Record lock, heap no 3 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
 0: len 8; hex 8000000000000002; asc         ;;
 1: len 6; hex 0000264b3a88; asc   &K: ;;
 2: len 7; hex 5000001bb71553; asc P     S;;
 3: len 4; hex 800000c8; asc     ;;
 4: len 5; hex 9999794a9b; asc   yJ ;;

*** (2) TRANSACTION: #1
TRANSACTION 642464392, ACTIVE 19 sec starting index read, thread declared inside InnoD
mysql tables in use 1, locked 1
3 lock struct(s), heap size 360, 2 row lock(s), undo log entries 1
MySQL thread id 40683, OS thread handle 0x2ba7ddc80700, query id 164893 127.0.0.1 root
update t01 set t1=now() where id=3
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 1799733 page no 3 n bits 72 index `PRIMARY` of table `test`.`t01
Record lock, heap no 3 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
 0: len 8; hex 8000000000000002; asc         ;;
 1: len 6; hex 0000264b3a88; asc   &K: ;;
 2: len 7; hex 5000001bb71553; asc P     S;;
 3: len 4; hex 800000c8; asc     ;;
 4: len 5; hex 9999794a9b; asc   yJ ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1799733 page no 3 n bits 72 index `PRIMARY` of table `test`.`t01
Record lock, heap no 4 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
 0: len 8; hex 8000000000000003; asc         ;;
 1: len 6; hex 0000264b3a77; asc   &K:w;;
 2: len 7; hex c7000033050134; asc    3  4;;
 3: len 4; hex 800000c8; asc     ;;
 4: len 5; hex 9999794a96; asc   yJ ;;

*** WE ROLL BACK TRANSACTION (2)
```

示例4:

```
create table t01(id bigint not null primary key, c1 int not null, t1 datetime,unique u
insert into t01 values(1,100,now()),(2,200,now()),(3,300,now()),(4,400,now()); commit;

#123: SET tx_isolation='READ-COMMITTED'; SET innodb_lock_wait_timeout=60; SET autocomm
#1: delete from t01 where c1=400;  /* 持有: X RK(uk_c1,400) */
#2: insert into t01 values(5,400,now()); /* 请求: S NK(uk_c1, ((300,400],5))  blocking
#3: insert into t01 values(6,400,now()); /* 请求: S NK(uk_c1, ((300,400],6))  blocking
#1: commit;    /* 释放 X RK(uk_c1,400) */
#3: /* 持有: S NK(uk_c1, ((300,400], 5)), 请求 : X IK(uk_c1, ((300,400),6))  blocking b
#2: /* 持有: S NK(uk_c1, ((300,400], 6)),请求 : X IK(uk_c1, ((300,400),5))   blocking b
```

deadlock report：

```
2016-05-28 20:13:09 2ba7e0080700
*** (1) TRANSACTION: #3
TRANSACTION 642464162, ACTIVE 50 sec inserting, thread declared inside InnoDB 1
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 360, 3 row lock(s), undo log entries 1
MySQL thread id 40695, OS thread handle 0x2ba7ddd01700, query id 163486 127.0.0.1 root
insert into t01 values(5,400,now())
*** (1) HOLDS THE LOCK(S):
RECORD LOCKS space id 1799729 page no 4 n bits 72 index `uk_c1` of table `test`.`t01`
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;

Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 80000190; asc     ;;
 1: len 8; hex 8000000000000004; asc         ;;

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1799729 page no 4 n bits 72 index `uk_c1` of table `test`.`t01`
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;

*** (2) TRANSACTION: #2
TRANSACTION 642464163, ACTIVE 40 sec inserting, thread declared inside InnoDB 1
mysql tables in use 1, locked 1
3 lock struct(s), heap size 360, 3 row lock(s), undo log entries 1
MySQL thread id 40699, OS thread handle 0x2ba7e0080700, query id 163491 127.0.0.1 root
insert into t01 values(6,400,now())
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 1799729 page no 4 n bits 72 index `uk_c1` of table `test`.`t01`
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;

Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 80000190; asc     ;;
 1: len 8; hex 8000000000000004; asc         ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1799729 page no 4 n bits 72 index `uk_c1` of table `test`.`t01`
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;

*** WE ROLL BACK TRANSACTION (2)
```

分析：

1. 这个例子列c1上有唯一索引，使用3个并发会话。
2. 会话1删除 `c1=400` 的记录，对聚集索引记录和唯一索引记录加 `X RK`

3. 会话2插入 `c1=400` 的记录，申请对唯一索引记录加 `S RK` （此处不同mysql版本行为不一样，大部分版本是加 `S NK` ），被会话1阻塞。
4. 会话3插入 `c1=400` 的记录，申请对唯一索引记录加 `S RK` ,被会话1阻塞。
5. 会话1，提交释放 `X RK` ，会话2和会话3抢占锁都成功了。会话2和会话3拿到 `S RK` ，然后都申请 `X IK` ，但被对方的 `S RK` 阻塞，死锁发生。

案例分析

阻塞案例分析

- 天猫快速活动业务锁等待超时

  天猫快速活动业务的一个投放规则表 `delta_rule` 上，有16个二级索引。光看这么多索引就知道该表的阻塞和死锁概率会很高。

  详细案例参见俞月总结的tmall_delta的锁等待超时排查 。
  这个案例中阻塞的原因是使用了事务（ `set autocommit=0;` ）,并且把dml sql和select sql放到一个事务里，导致事务持续时间没有做到足够短。

  备注：

  1. 该业务表可能还有其他阻塞问题，需要应用研发排查。DBA没有办法从DB端发现。
  2. 该业务表有时候凌晨0点以后还会发生死锁。可能跟大量数据更新（回流？）有关。还需要确认。

- 天猫库存业务减库存逻辑优化

  双11交易额能做到多大，很大程度取决于天猫库存业务单品减库存的效率。所以，减库存业务的逻辑从业务上和DB上都做了不少优化，其目的就是降低阻塞的概率，提升TPS值。
  详情参见凌洛总结的库存优化文章：

  1. 库存中心DB优化–语法优化 这是通过调整事务里写表的顺序，来降低某个lock的持有时间，达到降低阻塞冲突的概率，提高了并发。
  2. 库存中心DB优化–MySQL patch 这是AliSQL使用热点更新补丁，采用了特殊的语法和技术，降低了事务活跃的时间，达到降低阻塞冲突的概率，提高了并发。

死锁案例分析

- 天猫库存业务更新死锁

  天猫库存业务的库存更新业务存在死锁场景，详情请参见俞月总结的库存死锁分析和解决方案。
  这个案例死锁产生的可能原因是事务里有UPDATE SQL是强制根据二级索引去更新记录的，还有就是更新的执行计划有INDEX MERGE。在高并发时候出现死锁。
  目前库存死锁问题还在解决中。

- 蚂蚁芝麻信用业务UK列导致死锁

案例请参见江疑总结的 由一个死锁引发的对InnoDB行锁的探索。这个例子描述的是UK加锁逻辑引起的冲突。
在MySQL5.6.12-5.6.20期间，insert涉及到UK的时候，使用的是记录锁，而其他的mysql版本里，使用的是间隙锁（这跟上面说Read-Committed级别下不使用间隙锁矛盾）。
死锁的解决是通过调整Unique Key里索引列的顺序，将NDV高的列放在前面，这是一个很好的启发。

本节总结：
阻塞和死锁的成因都是一样的，通常都是不合理的加锁顺序、或者加锁的时间过长等等。这种不合理可能跟应用编码有关，也可能跟索引的使用有关。了解这些原理后，再分析几个经典案例，很容易就可以掌握同类问题的处理技巧。

>> 回到目录

SQL写法优化

以上介绍了索引、锁的知识。这节单说SQL的写法。 SQL的写法不同，也会导致使用的索引不同，产生的锁行为不同。这节就索引和锁的细节就不再重复介绍。

分页排序优化

单表自连接优化

通常分页排序SQL的写法就是where里写上条件，然后加上 `order by xxx limit m,n` 就完成功能了。但这个SQL的性能往往随着m的增大而不好。假设SQL走上了某个二级索引，其回表的成本很高。或者说“扫描的行数”会随着m增大而越来越大，尽管“返回的行数”是固定的。注意limit没法使用ICP技术。

至于性能是否有优化的空间取决于where里的条件列是否能全部被某个联合索引覆盖。
如果where里的条件都可以被联合索引覆盖，则单表的分页排序SQL可以写成该表自己跟自己join的SQL，其中里面的子查询的select列只有主键列，并且在子查询里面排序分页，然后外层的join列就是主键列。这样只是对二级索引进行了扫描，大大降低了回表的次数。

排序的陷阱

MySQL有两种文件排序算法，如果需要进行排序的列的总大小加上ORDER BY列的大小超过了 `max_length_for_sort_data` 定义的字节，MySQL首先根据主键排序，然后再根据主键查找结果，因此结果是有序的； 如果需要进行排序的列的总大小加上ORDER BY列的大小小于 `max_length_for_sort_data` 定义的字节，MySQL会对结果集直接在内存中排序。而这两种算法的结果集顺序可能不一致，翻页的时候会出现记录漏了或者记录重复。
解决办法： `order by` 的列后面增加主键列或者唯一索引列。

实际案例参见 查询字段不同导致排序后的结果不一致 。

连接优化

Join类型

Join分内连接和外连接。二者的结果返回形式不一样，where条件写法不一样，这些是数据库基础信息。不了解的可以网上查看一下介绍。

当sql里join的表很多的时候，有些是内连接，有些是外连接，条件众多，很容易写错，虽然结果可能碰巧对，但是SQL执行计划可能会因为写法而变化。
这属于低级错误，一定要避免。

NLJ连接原理

MySQL连接算法常用的就是“嵌套循环”（`Nested Loop Join`）。
假设表t1,t2 做join，类型如下

```
Table   Join Type
t1      range
t2      ref
```

简单的NLJ算法如下：

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

所以，要想让NLJ算法性能好，优化点有：
1. 让外层循环次数比内层循环次数低，即t1扫描的行数尽可能的比t2扫描的行数小很多，尽量让t1不回表（即索引覆盖）。通常说的小的结果集放前面（外面）。这是一个range操作，通常走二级索引。
2. 让内层取数据时间尽可能的低。即t2取数据的条件能走索引（聚集索引更好，二级索引也行，返回少量的数据）。t2的二级索引里的列顺序会很重要，顺序反了可能导致无法走索引。

MySQL针对NLJ还有写优化，稍微有点复杂，基本思想不变。只要遵循上面思路优化即可。

NLJ优化案例

分析：
1. `drrmt` 根据 `drrmt.tms_page_id` 过滤后的结果集应该很小，且上面有 `unique index` 。实际上外层驱动表是 `drrm` 而不是 `drrmt` 有点意外。这是SQL性能不好的直观原因。
2. `drrmt.tms_page_id` 列类型是 `varchar` ，sql传值是数字，发生隐式类型转换，导致走不上索引，从而也误导MySQL选择了错误的Join算法。

更正后的SQL：

子查询优化

对于 `in` （）这样的子查询，能用JOIN替换就替换。对于 `exists` 子查询不一定，看情况而定。

条件优化

之所以把条件写法单独列出来，一是为了规避一些写法错误导致SQL走不了正确的索引，二个是为了说明某些条件越具体越好。

经验：
1. 不要在条件列上应用表达式，以免使用不了该列上的索引。
2. 不要在条件的赋值上发生隐式转换，以免使用不了该列上的索引。
3. 时间列如果要判断为某一天改用BETWEEN…AND…方式。
4. 时间列条件的跨度越短越好，减少返回的行数。
5. 如果是后台任务，分页查询，那么分页的大小尽可能的大一些，减少查询次数。或者加上条件 `id >` 上次分页取到的最大ID。

>> 回到目录

SQL审核

在了解常见SQL性能问题、索引和锁的原理后，SQL审核就是一些操作上的事情了。
不过实际经验表明，到SQL审核这一步才发现有性能问题还是有点晚，但好过于上线后才发现问题。这里就专说SQL审核这一步如何操作。

SQLMaps审核

sqlmaps文件将sql和代码分离，极大的方便了sql审核工作，同时也方便了sql的重用。借助于iDB，DBA可以方便查阅sqlmaps文件。但是根据sql审核其性能做起来还是很难决策，很多信息要跟研发沟通，效率很低。

动态条件

为了让某个sql可以重用，研发习惯讲where条件里都写成动态条件。实际上哪些条件一定会传就留在研发的脑子里了，也建了相应的索引。
下一次需求换了一个人，发现有现成的接口方法可以复用（复用了该sql），只不过传的是别的条件组合。后面的研发想着反正是复用，可能不会考虑索引问题；或者条件组合跟现有索引不完全匹配，于是增加了新的索引。

这里的建议是做需求的人如果从业务模型上认为某个条件很有可能会作为查询条件，那就放到动态条件里，并且为其考虑索引；如果该条件还是必传的条件，那就拿掉动态条件语法，作为sql文本的固定部分。以防止其他人调用漏传必要条件。
要规避讲一个列作为动态条件了，但是却没有为其考虑索引这种做法，宁可不写这个动态条件，让将来有这个需求的人再来改这个sqlmaps文件。

sqlmaps文件里的sql重用度越高，承载的业务场景就可能越多，尽量增加必要的注释，提醒后人要怎么正确使用，怎么规避问题之类的。有时候可以选择不重用，而写几个不同查询场景的sql，以供不同业务场景调用。当然，分多了也不好维护。就是根据业务复杂度、重要性、性能多方权衡。

iDB里绑定sqlmaps文件

在iDB里提交sql审核的时候，一定要绑定到具体sqlmpas文件，或者当前目录。如果当前目录有很多sqlmaps文件并且大部分没有修改，那还是绑定到具体的文件比较好。
否则，绑定方式不对，研发同学要考虑一下DBA看到很多sqlmaps文件会无从下手，这会严重降低审核的效率，耽搁应用发布的时间。

理清楚查询条件组合后，就是看他们的执行计划了。

执行计划审核

执行计划查看方式

- DBA查看方式 explain 命令

explain后面接SQL即可，SQL不要有语法错误。命令会展示该SQL的执行计划。绝大部分时候这个执行计划跟实际执行时的执行计划是一致的。
伴随这个命令后习惯性的会查看表的索引，以综合判断这个执行计划是否使用了正确的索引。
这个例子没有用到索引，因为没有合适的索引。

- iDB查看执行计划

在SQL查询窗口输入SQL,结果的TAB选择执行计划，然后点“执行”

同时在iDB里查看对应表结构和索引结构

注意： iDB里给的行数和表大小的信息往往都很过时，只能参考。可以通过iDB查看实际行数再跟上面值对比，估算实际大小。

执行计划解释

MySQL执行计划十一个顺序执行的过程，每行记录表示一个操作，注意其操作次序，操作对象，用到的索引（`Key`），扫描的行数（`Rows`）。

EXPLAIN列的解释：

列	描述
`table`	显示这一行的数据是关于哪张表的。
`type`	这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为const、eq_reg、ref、range、index和ALL。
`possible_keys`	显示可能应用在这张表中的索引。如果为空，没有可能的索引。可以为相关的域从WHERE语句中选择一个合适的语句。
`key`	实际使用的索引。如果为NULL，则没有使用索引。很少的情况下，MySQL会选择优化不足的索引。这种情况下，可以在SELECT语句中使用USE
`INDEX (indexname)`	来强制使用一个索引或者用IGNORE INDEX（indexname）来强制MySQL忽略索引。
`key_len`	使用的索引的长度。在不损失精确性的情况下，长度越短越好。
`ref`	显示索引的哪一列被使用了，如果可能的话，是一个常数。
`rows`	MySQL认为必须检查的用来返回请求数据的行数。
`Extra`	关于MySQL如何解析查询的额外信息。将在表4.3中讨论，但这里可以看到的坏的例子是Using temporary和Using filesort，意思MySQL根本不能使用索引，结果是检索会很慢。

备注：
通常关注每个操作选择的`key`，以及`rows`，`Extra`信息。

extra列返回的描述的意义：

值	意义
	一旦MySQL找到了与行相联合匹配的行，就不再搜索

`Distinct`	了。
`Not exists`	MySQL优化了LEFT JOIN，一旦它找到了匹配LEFT JOIN标准的行，就不再搜索了。
`Range checked for each Record`	没有找到理想的索引，因此对于从前面表中来的每一个行组合，MySQL检查使用哪个索引，并用它来从表中返回行。这是使用索引的最慢的连接之一。
`Using filesort`	看到这个的时候，查询就需要优化了。MySQL需要进行额外的步骤来发现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行。
`Using index`	列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表的全部的请求列都是同一个索引的部分的时候。
`Using temporary`	看到这个的时候，查询需要优化了。这里，MySQL需要创建一个临时表来存储结果，这通常发生在对不同的列集进行ORDER BY上，而不是GROUP BY上。
`Where used`	使用了WHERE从句来限制哪些行将与下一张表匹配或者是返回给用户。如果不想返回表中的全部行，并且连接类型ALL或index，这就会发生，或者是查询有问题不同连接类型的解释（按照效率高低的顺序排序）。
`system`	表只有一行 system 表。这是const连接类型的特殊情况。
`const`	表中的一个记录的最大值能够匹配这个查询（索引可以是主键或惟一索引）。因为只有一行，这个值实际就是常数，因为MySQL先读这个值然后把它当做常数来对待。
`eq_ref`	在连接中，MySQL在查询时，从前面的表中，对每一个记录的联合都从表中读取一个记录，它在查询使用了索引为主键或惟一键的全部时使用。
`ref`	这个连接类型只有在查询使用了不是惟一或主键的键或者是这些类型的部分（比如，利用最左边前缀）时发生。对于之前的表的每一个行联合，全部记录都将从表中读出。这个类型严重依赖于根据索引匹配的记录多少—越少越好。
`range`	这个连接类型使用索引返回一个范围中的行，比如使用>或<查找东西时发生的情况。

`index`	这个连接类型对前面的表中的每一个记录联合进行完全扫描（比ALL更好，因为索引一般小于表数据）。
`ALL`	这个连接类型对于前面的每一个记录联合进行完全扫描，这一般比较糟糕，应该尽量避免。

备注：
通常 extra显示为 `const` 、 `eq_ref` 最好， `ref` 、 `Using index` , `range` , `index` 尚可， `ALL` 可能就最糟。

执行计划就多看看例子，实际工作中多看看自己SQL的执行计划，很容易掌握。

执行计划示例

- 天猫俪人购业务

分析：
1. 用了子查询，应当改为表连接，虽然子查询的执行计划通常也会转换为表连接。但不要依赖db的优化器去做。
2. `idx_isdelete` 这个从 `Cardinality` 上看根本不适合建索引，而这个SQL就是误走了这个索引（当然，因为它没有更合适的索引）。所以，第一行执行计划扫描的行数（ `Rows` ）很高。
3. `sf_item_shadow_0000` 直观原因是缺索引，可是这个表上已经有很多索引了，从 `sf_category_id` 和 `sf_category_no` 都作为索引头列，可知业务需求混乱。查询场景太多了，索引也很多。
4. `.*` 是我改写的，节省图片篇幅。应用不要写 `.*` ,必须具体的写明每个字段。

- 创新供应链/3w大家电业务

分析：
1. 这个例子展示了一个“子查询”的执行计划会是什么样子，看图中 `<derived3>` 信息。
2. 滥用子查询，多此一举。
3. 滥用 `LEFT JOIN` 。 "AND B.deleted= 0 AND B.scitem_id= 522071700058" 这个条件导致 `LEFT JOIN` 偏离本意，结果跟内连接相同。
4. 两个表都没有用到索引，因为前面的2个低级错误。

去掉子查询、改掉 `LEFT JOIN` ，执行计划如下：

- 商超业务

分析：
1. 没有精确匹配的索引，所以选了一个最接近的索引。扫描的行数（ `Rows` ）相对很高。

分析：
1. 单列索引太多，导致选择了2个单列索引做 `intersect` 。其实一个组合索引就可以搞定。
2. 单列索引总是容易被误用，可以拉上几个经常作为条件的状态或类型列或者时间一起建个联合索引（当然，也要是实际业务查询场景）。

分析：
1. 索引走对了。
2. 后台定时任务，分页大小为50，太小，导致分页取的次数太多。可以一次性取1W,5W都没关系（根据应用程序内存而定）。

使用场景沟通

举两个极端例子。
一个慢SQL，全表扫描，一月或一周或一天跑一次，业务低峰期跑也没问题。特殊大促节日，停掉其运行（降级）。执行计划再烂，这样的SQL也可以上。如各种报表业务。
一个很快的SQL，线上跑一下只要5ms，但是请求量非常高，虽然前端用了tair，但是落到DB上的请求还是不少，尤其是大促的时候。这样的SQL，即使执行计划最优，也要想办法进一步优化，降低rt。如库存业务的减库存SQL.
绝大部分的业务SQL都介入这二者之间。

跑得慢的SQL，并发不能很多；并发高的SQL,必须跑得快！
所以，一个SQL能不能上线，除了看其执行计划是否做到最优了，还要看其使用场景。
1. 该表多大数据量？数据增量？是否要分库分表？
2. 该sql多大调用量（QPS）？业务对DB的rt要求多少？
3. 该表有多少种查询场景？多少个索引？

当查询场景很多的时候，不可能为每种查询场景都建立索引。这时候就要对业务查询场景进行分级。优先级高的优先满足，优先级低的，要么改需求（改SQL加条件），要么换时间执行，要么就接受慢且还不能影响其他业务。

DBA需要跟研发沟通，研发需要跟PD沟通。沟通的越早，修改的成本越小，修改的可能性越大。

>> 回到目录

观察线上性能

应用发上线后，还需要关注一下应用。通常研发同学会关注功能是否正确、系统是否文档、性能是否有严重问题。
这里看的性能问题是从应用层面粗略的看。如果问题隐藏的很深，是很难及时发现。

本节就介绍一些具体的跟SQL有关的性能观察手段。经常在发布后关注一段时间，及时发现问题，及时修复，总结经验，逐步提升SQL优化和审核技能。

观察内容

重点观察SQL的QPS、RT等。

SQL调用次数（`QPS`）

一个SQL的性能跟它执行的QPS是有密切关系（但不是线性关系）。
如果一个SQL性能不好，执行时间在几十ms或者几秒以上，QPS只要稍微上去一点（如并发超过10），MySQL实例主机的CPU或者IO压力就非常大，进而该SQL的rt会下降。
如果一个SQL性能很好，执行时间在0.5ms~1ms之间，它的QPS可以很高，比如说达到1K时可能都没有问题，但是如果再高，MySQL实例主机的CPU或者IO同样有可能压力增加，也会导致该SQL部分请求的延时增加到几ms甚至几十ms。

线上有很多慢SQL，跑着也没有导致什么问题。一方面是DB主机的性能很好，能抗得住，另外一方面，这类SQL的并发通常也不高，所以相安无事。
但是，一旦有特别的事件。比如说有流量攻击，或者促销活动，或者前端tair异常，导致DB上的流量突增，平时没关注的慢SQL一下子把DB拖垮了，导致DB主备切换。但没过多久，新的主库又挂了。
所以，当前无事并不表示将来无事。有性能问题的SQL出事的概率总是要高于没问题的SQL。提前消灭问题是负责任的做法。

关注QPS一般是关注高峰期时的值，同时也可以观察一下24小时内每个时段的QPS，促进对业务的理解。反过来，理解业务了都不用看指标就可以估计出各个时段的调用量。第一次还是要看看，设计是否符合预期。
通过QPS的调用异常，可能还会发现应用中隐藏很深的BUG。如类似循环调用问题、连接不释放问题等等。

SQL的RT

从应用层面观察到SQL的响应时间是包含应用服务器到DB主机网络上来回2次的时间 加上 SQL在DB内部的执行时间。
每个应用的SQL的rt都有个期望值，一定要找到这个期望值，尽可能的描述清楚，然后观察实际的DB的rt跟期望值差别大不大。核心应用经常观察，留意性能的变化。其原因可能是QPS增加了，数据量增加了，或者DB性能出了问题等等。

观察方法

可以从应用和DB两个层面观察性能。

iDB查看数据库性能

iDB里看的性能是这个实例级别的，关注指标如下。每个指标的意义在iDB里都有提示。

iDB里展示的都是每分钟各个指标的“平均值”。平均值有个问题就是个别数据点除非有很大的变化，否则，平均值很难看到变化。这里就要善于从平均值的变化中预测实际发生了什么事情。当然，要多个指标一起观察才可以缩小范围。

- 天猫俪人购业务库（`tmall_linlang`）

分析：
1. 这只是个例子。并不一定是个问题，是否构成问题要看对DB，对业务是否产生或者有可能产生负面影响。
2. 这个例子里，实例在`10:19`左右，平均RT抖动了一下。主机的LOAD 也抖动了一下，结合QPS和TPS的变化，初步可以推断是有大量INSERT导致的。（INSERT的rt通常高于select的rt，对平均值影响的能力更大）。
3. 大量的INSERT，就很可能是业务表回流数据。那么，就要查一下是什么业务表。以及在这个高峰期回流是否妥当。等等。

鹰眼链路

依然以`tmall_linlang`举例：

`tmall_linlang`鹰眼调用链路

查询明细：

即席故障多维分析

链路明细

查看SQL明细：

iDB4的CloudDBA剧透

idb4目前还在试用阶段，设计完全面向研发的。其中CloudDBA诊断系统是个创新，能有效帮助研发发现问题SQL。

（CloudDBA是将DBA的诊断经验，一些原理沉淀在系统里，功能还在逐步完善。他可以帮助你做决定，但不能代替你做决定。所以，了解SQL优化背后的原理还是很有必要的）。

CloudDBA入口：

性能概览：

诊断结果：

慢SQL建议：

其他：

>> 回到目录

后记

信息量有点多，还需日常多多练习，总结，交流。有问题欢迎随时找我、俞月、海缤。😀

本周四下午DBA将会给天猫技术部研发培训SQL审核和优化。考虑到大家很忙，不一定有空。本文是培训内容文字稿。如果大家看了还有兴趣，也可以报名培训沟通交流。培训链接后面给出。