Mysql高级-day03

1. 应用优化

前面章节，我们介绍了很多数据库的优化措施。但是在实际生产环境中，由于数据库本身的性能局限，就必须要对前台的应用进行一些优化，来降低数据库的访问压力。

1.1 使用连接池

对于访问数据库来说，建立连接的代价是比较昂贵的，因为我们频繁的创建关闭连接，是比较耗费资源的，我们有必要建立 数据库连接池，以提高访问的性能。

1.2 减少对MySQL的访问

1.2.1 避免对数据进行重复检索

在编写应用代码时，需要能够理清对数据库的访问逻辑。能够一次连接就获取到结果的，就不用两次连接，这样可以大大减少对数据库无用的重复请求。

比如 ，需要获取书籍的id 和name字段 ， 则查询如下：

```
 select id , name from tb_book;
```

之后，在业务逻辑中有需要获取到书籍状态信息， 则查询如下：

```
select id , status from tb_book;
```

这样，就需要向数据库提交两次请求，数据库就要做两次查询操作。其实完全可以用一条SQL语句得到想要的结果。

```
select id, name , status from tb_book;
```

1.2.2 增加cache层

在应用中，我们可以在应用中增加 缓存 层来达到减轻数据库负担的目的。缓存层有很多种，也有很多实现方式，只要能达到降低数据库的负担又能满足应用需求就可以。

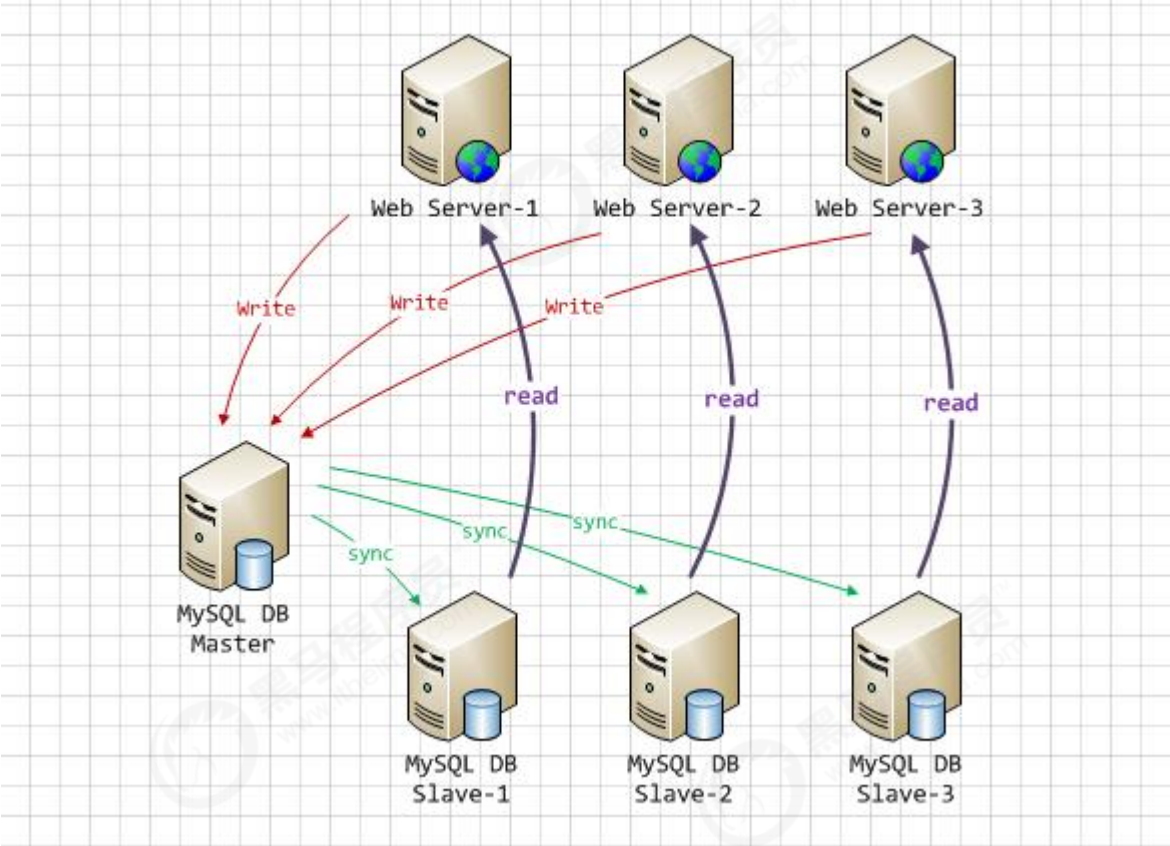
因此可以部分数据从数据库中抽取出来放到应用端以文本方式存储， 或者使用框架(Mybatis, Hibernate)提供的一级缓存/二级缓存，或者使用redis数据库来缓存数据 。

1.3 负载均衡

负载均衡是应用中使用非常普遍的一种优化方法，它的机制就是利用某种均衡算法，将固定的负载量分布到不同的服务器上， 以此来降低单台服务器的负载，达到优化的效果。

1.3.1 利用MySQL复制分流查询

通过MySQL的主从复制，实现读写分离，使增删改操作走主节点，查询操作走从节点，从而可以降低单台服务器的读写压力。



1.3.2 采用分布式数据库架构

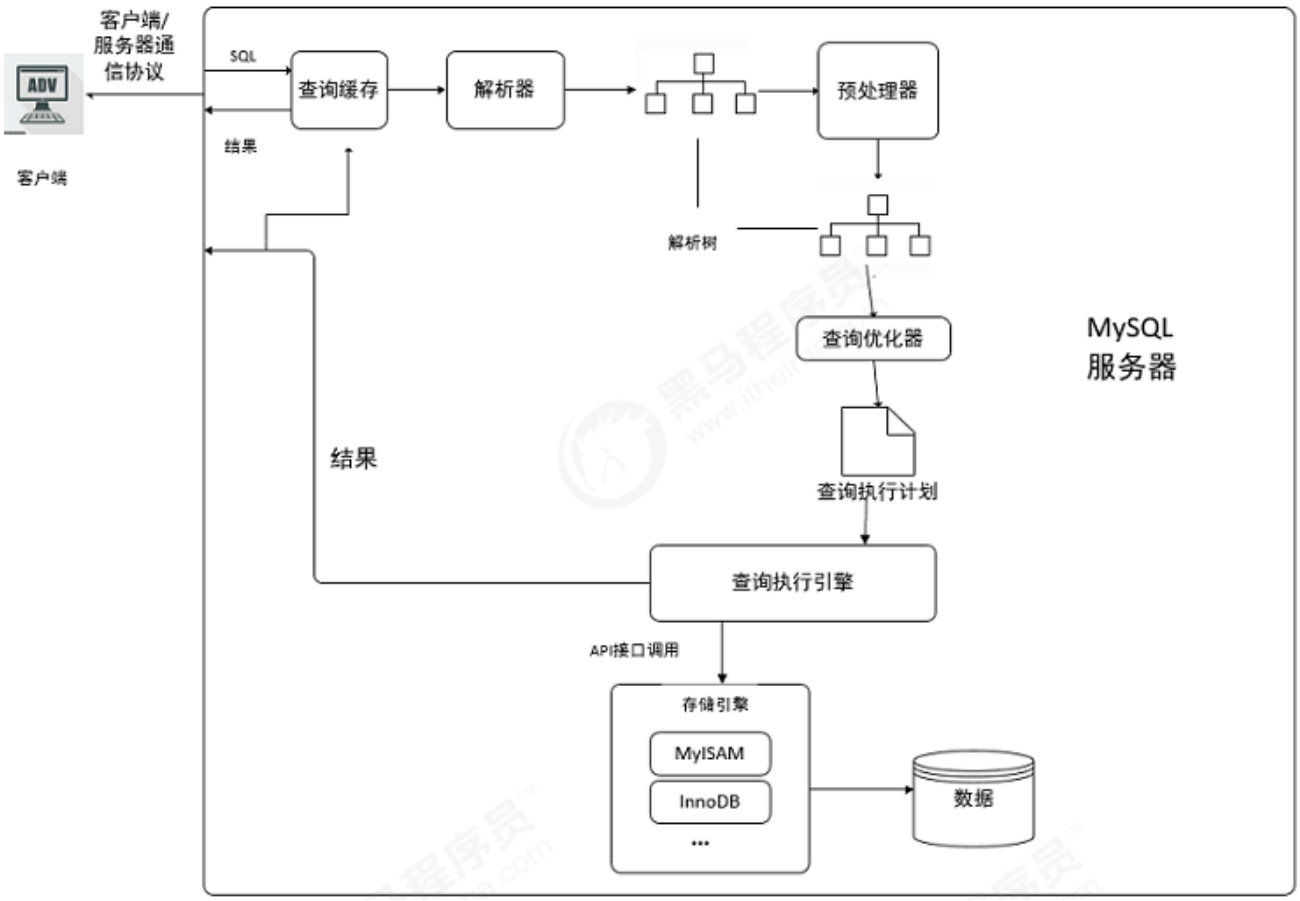
分布式数据库架构适合大数据量、负载高的情况，它有良好的拓展性和高可用性。通过在多台服务器之间分布数据，可以实现在多台服务器之间的负载均衡，提高访问效率。

2. Mysql中查询缓存优化

2.1 概述

开启Mysql的查询缓存，当执行完全相同的SQL语句的时候，服务器就会直接从缓存中读取结果，当数据被修改，之前的缓存会失效，修改比较频繁的表不适合做查询缓存。

2.2 操作流程



1. 客户端发送一条查询给服务器；
2. 服务器先会检查查询缓存，如果命中了缓存，则立即返回存储在缓存中的结果。否则进入下一阶段；
3. 服务器端进行SQL解析、预处理，再由优化器生成对应的执行计划；
4. MySQL根据优化器生成的执行计划，调用存储引擎的API来执行查询；
5. 将结果返回给客户端。

2.3 查询缓存配置

1. 查看当前的MySQL数据库是否支持查询缓存：

```
SHOW VARIABLES LIKE 'have_query_cache';
```

```
mysql> SHOW VARIABLES LIKE 'have_query_cache';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| have_query_cache | YES   |
+------------------+-------+
1 row in set (0.00 sec)
```

2. 查看当前MySQL是否开启了查询缓存 ：

```
SHOW VARIABLES LIKE 'query_cache_type';
```

```
mysql> SHOW VARIABLES LIKE 'query_cache_type';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | OFF   |
+------------------+-------+
1 row in set (0.00 sec)
```

3. 查看查询缓存的占用大小 ：

```
SHOW VARIABLES LIKE 'query_cache_size';
```

```
mysql> SHOW VARIABLES LIKE 'query_cache_size';
+------------------+---------+
| Variable_name    | Value   |
+------------------+---------+
| query_cache_size | 1048576 |
+------------------+---------+
1 row in set (0.00 sec)
```

4. 查看查询缓存的状态变量：

```
SHOW STATUS LIKE 'Qcache%';
```

```
mysql> SHOW STATUS LIKE 'Qcache%';
+-------------------------+---------+
| Variable_name           | Value   |
+-------------------------+---------+
| Qcache_free_blocks      | 1       |
| Qcache_free_memory      | 1039880 |
| Qcache_hits             | 0       |
| Qcache_inserts          | 0       |
| Qcache_lowmem_prunes    | 0       |
| Qcache_not_cached       | 72      |
| Qcache_queries_in_cache | 0       |
| Qcache_total_blocks     | 1       |
+-------------------------+---------+
8 rows in set (0.00 sec)
```

各个变量的含义如下：

参数	含义
Qcache_free_blocks	查询缓存中的可用内存块数
Qcache_free_memory	查询缓存的可用内存量
Qcache_hits	查询缓存命中数
Qcache_inserts	添加到查询缓存的查询数
Qcache_lowmen_prunes	由于内存不足而从查询缓存中删除的查询数
Qcache_not_cached	非缓存查询的数量（由于 query_cache_type 设置而无法缓存或未缓存）
Qcache_queries_in_cache	查询缓存中注册的查询数
Qcache_total_blocks	查询缓存中的块总数

2.4 开启查询缓存

MySQL的查询缓存默认是关闭的，需要手动配置参数 query_cache_type ， 来开启查询缓存。query_cache_type 该参数的可取值有三个 ：

值	含义
OFF 或 0	查询缓存功能关闭
ON 或 1	查询缓存功能打开，SELECT的结果符合缓存条件即会缓存，否则，不予缓存，显式指定 SQL_NO_CACHE，不予缓存
DEMAND 或 2	查询缓存功能按需进行，显式指定 SQL_CACHE 的SELECT语句才会缓存；其它均不予缓存

在 /usr/my.cnf 配置中，增加以下配置 ：

```
#开启mysql的查询缓存
query_cache_type=1
```

配置完毕之后，重启服务既可生效 ；

然后就可以在命令行执行SQL语句进行验证 ，执行一条比较耗时的SQL语句，然后再多执行几次，查看后面几次的执行时间；获取通过查看查询缓存的缓存命中数，来判定是否走查询缓存。

2.5 查询缓存SELECT选项

可以在SELECT语句中指定两个与查询缓存相关的选项 ：

SQL_CACHE : 如果查询结果是可缓存的，并且 query_cache_type 系统变量的值为ON或 DEMAND ，则缓存查询结果 。

SQL_NO_CACHE : 服务器不使用查询缓存。它既不检查查询缓存，也不检查结果是否已缓存，也不缓存查询结果。

例子：

```
SELECT SQL_CACHE id, name FROM customer;
SELECT SQL_NO_CACHE id, name FROM customer;
```

2.6 查询缓存失效的情况

1 ）SQL 语句不一致的情况， 要想命中查询缓存，查询的SQL语句必须一致。

```
SQL1 : select count(*) from tb_item;
SQL2 : Select count(*) from tb_item;
```

2 ）当查询语句中有一些不确定的时，则不会缓存。如 ： now() , current_date() , curdate() , curtime() , rand() , uuid() , user() , database() 。

```
SQL1 : select * from tb_item where updatetime < now() limit 1;
SQL2 : select user();
SQL3 : select database();
```

3 ）不使用任何表查询语句。

```
select 'A';
```

4）查询 mysql， information_schema或 performance_schema 数据库中的表时，不会走查询缓存。

```
select * from information_schema.engines;
```

5）在存储的函数，触发器或事件的主体内执行的查询。

6）如果表更改，则使用该表的所有高速缓存查询都将变为无效并从高速缓存中删除。这包括使用 `MERGE` 映射到已更改表的表的查询。一个表可以被许多类型的语句，如被改变 INSERT， UPDATE， DELETE， TRUNCATE TABLE， ALTER TABLE， DROP TABLE，或 DROP DATABASE 。

3. Mysql内存管理及优化

3.1 内存优化原则

1）将尽量多的内存分配给MySQL做缓存，但要给操作系统和其他程序预留足够内存。

2）MyISAM 存储引擎的数据文件读取依赖于操作系统自身的IO缓存，因此，如果有MyISAM表，就要预留更多的内存给操作系统做IO缓存。

3）排序区、连接区等缓存是分配给每个数据库会话（session）专用的，其默认值的设置要根据最大连接数合理分配，如果设置太大，不但浪费资源，而且在并发连接较高时会导致物理内存耗尽。

3.2 MyISAM 内存优化

myisam存储引擎使用 key_buffer 缓存索引块，加速myisam索引的读写速度。对于myisam表的数据块，mysql没有特别的缓存机制，完全依赖于操作系统的IO缓存。

key_buffer_size

key_buffer_size决定MyISAM索引块缓存区的大小，直接影响到MyISAM表的存取效率。可以在MySQL参数文件中设置key_buffer_size的值，对于一般MyISAM数据库，建议至少将1/4可用内存分配给key_buffer_size。

在/usr/my.cnf 中做如下配置：

```
key_buffer_size=512M
```

read_buffer_size

如果需要经常顺序扫描myisam表，可以通过增大read_buffer_size的值来改善性能。但需要注意的是read_buffer_size是每个session独占的，如果默认值设置太大，就会造成内存浪费。

read_rnd_buffer_size

对于需要做排序的myisam表的查询，如带有order by子句的sql，适当增加 read_rnd_buffer_size 的值，可以改善此类的sql性能。但需要注意的是 read_rnd_buffer_size 是每个session独占的，如果默认值设置太大，就会造成内存浪费。

3.3 InnoDB 内存优化

innodb用一块内存区做IO缓存池，该缓存池不仅用来缓存innodb的索引块，而且也用来缓存innodb的数据块。

innodb_buffer_pool_size

该变量决定了 innodb 存储引擎表数据和索引数据的最大缓存区大小。在保证操作系统及其他程序有足够内存可用的情况下，innodb_buffer_pool_size 的值越大，缓存命中率越高，访问InnoDB表需要的磁盘I/O 就越少，性能也就越高。

```
innodb_buffer_pool_size=512M
```

innodb_log_buffer_size

决定了innodb重做日志缓存的大小，对于可能产生大量更新记录的大事务，增加innodb_log_buffer_size的大小，可以避免innodb在事务提交前就执行不必要的日志写入磁盘操作。

```
innodb_log_buffer_size=10M
```

4. Mysql并发参数调整

从实现上来说，MySQL Server 是多线程结构，包括后台线程和客户服务线程。多线程可以有效利用服务器资源，提高数据库的并发性能。在Mysql中，控制并发连接和线程的主要参数包括 max_connections、back_log、thread_cache_size、table_open_cahce。

4.1 max_connections

采用max_connections 控制允许连接到MySQL数据库的最大数量，默认值是 151。如果状态变量 connection_errors_max_connections 不为零，并且一直增长，则说明不断有连接请求因数据库连接数已达到允许最大值而失败，这是可以考虑增大max_connections 的值。

Mysql 最大可支持的连接数，取决于很多因素，包括给定操作系统平台的线程库的质量、内存大小、每个连接的负荷、CPU的处理速度，期望的响应时间等。在Linux 平台下，性能好的服务器，支持 500-1000 个连接不是难事，需要根据服务器性能进行评估设定。

4.2 back_log

back_log 参数控制MySQL监听TCP端口时设置的积压请求栈大小。如果MySql的连接数达到max_connections时，新来的请求将会被存在堆栈中，以等待某一连接释放资源，该堆栈的数量即back_log，如果等待连接的数量超过back_log，将不被授予连接资源，将会报错。5.6.6 版本之前默认值为 50 ， 之后的版本默认为 50 + （ max_connections / 5 ） ， 但最大不超过900。

如果需要数据库在较短的时间内处理大量连接请求， 可以考虑适当增大back_log 的值。

4.3 table_open_cache

该参数用来控制所有SQL语句执行线程可打开表缓存的数量， 而在执行SQL语句时，每一个SQL执行线程至少要打开 1 个表缓存。该参数的值应该根据设置的最大连接数 max_connections 以及每个连接执行关联查询中涉及的表的最大数量来设定 ：

max_connections x N ；

4.4 thread_cache_size

为了加快连接数据库的速度，MySQL 会缓存一定数量的客户服务线程以备重用，通过参数 thread_cache_size 可控制 MySQL 缓存客户服务线程的数量。

4.5 innodb_lock_wait_timeout

该参数是用来设置InnoDB 事务等待行锁的时间，默认值是50ms ， 可以根据需要进行动态设置。对于需要快速反馈的业务系统来说，可以将行锁的等待时间调小，以避免事务长时间挂起； 对于后台运行的批量处理程序来说，可以将行锁的等待时间调大， 以避免发生大的回滚操作。

5. Mysql锁问题

5.1 锁概述

锁是计算机协调多个进程或线程并发访问某一资源的机制（避免争抢）。

在数据库中，除传统的计算资源（如 CPU、RAM、I/O 等）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。

5.2 锁分类

从对数据操作的粒度分 ：

1 ）表锁：操作时，会锁定整个表。

2 ）行锁：操作时，会锁定当前操作行。

从对数据操作的类型分：

1 ）读锁（共享锁）：针对同一份数据，多个读操作可以同时进行而不会互相影响。

2）写锁（排它锁）：当前操作没有完成之前，它会阻断其他写锁和读锁。

5.3 Mysql 锁

相对其他数据库而言，MySQL的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。下表中罗列出了各存储引擎对锁的支持情况：

存储引擎	表级锁	行级锁	页面锁
MyISAM	支持	不支持	不支持
InnoDB	支持	支持	不支持
MEMORY	支持	不支持	不支持
BDB	支持	不支持	支持

MySQL这3种锁的特性可大致归纳如下 ：

锁类型	特点
表级锁	偏向MyISAM 存储引擎，开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。
行级锁	偏向InnoDB 存储引擎，开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。
页面锁	开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

从上述特点可见，很难笼统地说哪种锁更好，只能就具体应用的特点来说哪种锁更合适！仅从锁的角度来说：表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web 应用；而行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有并查询的应用，如一些在线事务处理（OLTP）系统。

5.2 MyISAM 表锁

MyISAM 存储引擎只支持表锁，这也是MySQL开始几个版本中唯一支持的锁类型。

5.2.1 如何加表锁

MyISAM 在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（UPDATE、DELETE、INSERT 等）前，会自动给涉及的表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用 LOCK TABLE 命令给 MyISAM 表显式加锁。

显示加表锁语法：

```
加读锁 ： lock table table_name read;

加写锁 ： lock table table_name write ;
```

5.2.2 读锁案例

准备环境

```
create database demo_03 default charset=utf8mb4;

use demo_03;

CREATE TABLE `tb_book` (
  `id` INT(11) auto_increment,
  `name` VARCHAR(50) DEFAULT NULL,
  `publish_time` DATE DEFAULT NULL,
  `status` CHAR(1) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=myisam DEFAULT CHARSET=utf8 ;

INSERT INTO tb_book (id, name, publish_time, status) VALUES(NULL,'java编程思想','2088-08-01','1');
INSERT INTO tb_book (id, name, publish_time, status) VALUES(NULL,'solr编程思想','2088-08-08','0');



CREATE TABLE `tb_user` (
  `id` INT(11) auto_increment,
  `name` VARCHAR(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=myisam DEFAULT CHARSET=utf8 ;

INSERT INTO tb_user (id, name) VALUES(NULL,'令狐冲');
INSERT INTO tb_user (id, name) VALUES(NULL,'田伯光');

```

客户端 一 ：

1) 获得tb_book 表的读锁

```
lock table tb_book read;
```

2) 执行查询操作

```
select * from tb_book;
```

```
mysql> select * from tb_book;
+----+------------------+--------------+--------+
| id | name             | publish_time | status |
+----+------------------+--------------+--------+
|  1 | java编程思想     | 2088-08-01   | 1      |
|  2 | solr编程思想     | 2088-08-08   | 0      |
+----+------------------+--------------+--------+
2 rows in set (0.00 sec)
```

可以正常执行 ， 查询出数据。

客户端 二 ：

3) 执行查询操作

```
select * from tb_book;
```

```
mysql> select * from tb_book;
+----+------------------+--------------+--------+
| id | name             | publish_time | status |
+----+------------------+--------------+--------+
|  1 | java编程思想     | 2088-08-01   | 1      |
|  2 | solr编程思想     | 2088-08-08   | 0      |
+----+------------------+--------------+--------+
2 rows in set (0.00 sec)
```

客户端 一 ：

4) 查询未锁定的表

```
select name from tb_seller;
```

```
mysql> select name from tb_seller;
ERROR 1100 (HY000): Table 'tb_seller' was not locked with LOCK TABLES
mysql>
```

客户端 二 ：

5) 查询未锁定的表

```
select name from tb_seller;
```

```
mysql> select name from tb_seller;
+------------------------------------------------------------+
| name                                                       |
+------------------------------------------------------------+
| OPPO科技有限公司                                           |
| 传智播客教育科技有限公司                                   |
| 传智播客教育科技有限公司旗下黑马程序员                     |
| 千度科技                                                   |
| 华为科技有限公司                                           |
| 宜家家居                                                   |
| 小米科技                                                   |
| 掌趣科技股份有限公司                                       |
| 新浪科技有限公司                                           |
| 百度科技有限公司                                           |
| 罗技科技有限公司                                           |
| 阿里巴巴                                                   |
+------------------------------------------------------------+
12 rows in set (0.00 sec)
```

可以正常查询出未锁定的表；

客户端 一 ：

6 ） 执行插入操作

```
insert into tb_book values(null,'Mysql高级','2088-01-01','1');
```

```
mysql>
mysql> insert into tb_book values(null,'Mysql高级','2088-01-01','1');
ERROR 1099 (HY000): Table 'tb_book' was locked with a READ lock and can't be updated
mysql>
```

执行插入， 直接报错 ， 由于当前tb_book 获得的是 读锁， 不能执行更新操作。

客户端 二 ：

7 ） 执行插入操作

```
insert into tb_book values(null,'Mysql高级','2088-01-01','1');
```

```
mysql> insert into tb_book values(null,'Mysql高级','2088-01-01','1');
       等待中。。。
```

当在客户端一中释放锁指令 unlock tables 后 ， 客户端二中的 inesrt 语句 ， 立即执行 ；

5.2.3 写锁案例

客户端 一 :

1) 获得tb_book 表的写锁

```
lock table tb_book write ;
```

2) 执行查询操作

```
select * from tb_book ;
```

```
mysql> select * from tb_book ;
+----+------------------+--------------+--------+
| id | name             | publish_time | status |
+----+------------------+--------------+--------+
|  1 | java编程思想     | 2088-08-01   | 1      |
|  2 | solr编程思想     | 2088-08-08   | 0      |
|  3 | Mysql高级        | 2088-01-01   | 1      |
+----+------------------+--------------+--------+
3 rows in set (0.00 sec)
```

查询操作执行成功 ;

3) 执行更新操作

```
update tb_book set name = 'java编程思想 ( 第二版 ) ' where id = 1;
```

```
mysql> update tb_book set name = 'java编程思想（第二版）' where id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

更新操作执行成功 ；

客户端 二：

4) 执行查询操作

```
select * from tb_book ;
```

```
mysql> select * from tb_book;
       等待中 。 。 。
```

当在客户端一中释放锁指令 unlock tables 后 ， 客户端二中的 select 语句 ， 立即执行 ；

```
mysql> select * from tb_book;

+----+---------------------------------+--------------+--------+
| id | name                            | publish_time | status |
+----+---------------------------------+--------------+--------+
|  1 | java编程思想（第二版）          | 2088-08-01   | 1      |
|  2 | solr编程思想                    | 2088-08-08   | 0      |
|  3 | Mysql高级                       | 2088-01-01   | 1      |
+----+---------------------------------+--------------+--------+
3 rows in set (1 min 55.60 sec)
```

5.2.4 结论

锁模式的相互兼容性如表中所示：

当前锁模式 \ 请求锁模式	None	读锁	写锁
读锁	是	是	否
写锁	是	否	否

由上表可见：

1 ）对MyISAM 表的读操作，不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求；

2 ）对MyISAM 表的写操作，则会阻塞其他用户对同一表的读和写操作；

简而言之，就是读锁会阻塞写，但是不会阻塞读。而写锁，则既会阻塞读，又会阻塞写。

此外，MyISAM 的读写锁调度是写优先，这也是MyISAM不适合做写为主的表的存储引擎的原因。因为写锁后，其他线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永远阻塞。

5.2.5 查看锁的争用情况

```
show open tables ;
```

```
mysql> show open tables;
+--------------------+-----------------------------------------------------+--------+-------------+
| Database           | Table                                               | In_use | Name_locked |
+--------------------+-----------------------------------------------------+--------+-------------+
| performance_schema | events_waits_history                                |      0 |           0 |
| performance_schema | events_waits_summary_global_by_event_name           |      0 |           0 |
| performance_schema | setup_timers                                        |      0 |           0 |
| performance_schema | events_waits_history_long                           |      0 |           0 |
| demo_02            | user_role                                           |      0 |           0 |
| performance_schema | mutex_instances                                     |      0 |           0 |
| performance_schema | events_waits_summary_by_instance                    |      0 |           0 |
| performance_schema | events_stages_history                               |      0 |           0 |
| performance_schema | events_stages_summary_by_thread_by_event_name       |      0 |           0 |
| performance_schema | events_waits_summary_by_thread_by_event_name        |      0 |           0 |
| mysql              | user                                                |      0 |           0 |
| performance_schema | events_statements_summary_by_user_by_event_name     |      0 |           0 |
| performance_schema | events_statements_history_long                      |      0 |           0 |
| performance_schema | performance_timers                                  |      0 |           0 |
| performance_schema | file_instances                                      |      0 |           0 |
| performance_schema | events_stages_summary_by_host_by_event_name         |      0 |           0 |
| performance_schema | events_stages_history_long                          |      0 |           0 |
| demo_02            | goods_myisam                                        |      0 |           0 |
```

In_user : 表当前被查询使用的次数。如果该数为零，则表是打开的，但是当前没有被使用。

Name_locked：表名称是否被锁定。名称锁定用于取消表或对表进行重命名等操作。

```
1 show status like 'Table_locks%';
```

```
mysql> show status like 'Table_locks%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Table_locks_immediate | 70    |
| Table_locks_waited    | 0     |
+-----------------------+-------+
2 rows in set (0.00 sec)
```

Table_locks_immediate ： 指的是能够立即获得表级锁的次数，每立即获取锁，值加1。

Table_locks_waited ： 指的是不能立即获取表级锁而需要等待的次数，每等待一次，该值加1，此值高说明存在着较为严重的表级锁争用情况。

5.3 InnoDB 行锁

5.3.1 行锁介绍

行锁特点 ：偏向InnoDB 存储引擎，开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

InnoDB 与 MyISAM 的最大不同有两点：一是支持事务；二是 采用了行级锁。

5.3.2 背景知识

事务及其ACID属性

事务是由一组SQL语句组成的逻辑处理单元。

事务具有以下4个特性，简称为事务ACID属性。

ACID属性	含义
原子性（Atomicity）	事务是一个原子操作单元，其对数据的修改，要么全部成功，要么全部失败。
一致性（Consistent）	在事务开始和完成时，数据都必须保持一致状态。
隔离性（Isolation）	数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的 “独立” 环境下运行。
持久性（Durable）	事务完成之后，对于数据的修改是永久的。

并发事务处理带来的问题

问题	含义
丢失更新（Lost Update）	当两个或多个事务选择同一行，最初的事务修改的值，会被后面的事务修改的值覆盖。
脏读（Dirty Reads）	当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。
不可重复读（Non-Repeatable Reads）	一个事务在读取某些数据后的某个时间，再次读取以前读过的数据，却发现和以前读出的数据不一致。
幻读（Phantom Reads）	一个事务按照相同的查询条件重新读取以前查询过的数据，却发现其他事务插入了满足其查询条件的新数据。

事务隔离级别

为了解决上述提到的事务并发问题，数据库提供一定的事务隔离机制来解决这个问题。数据库的事务隔离越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使用事务在一定程度上“串行化” 进行，这显然与“并发” 是矛盾的。

数据库的隔离级别有4个，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏写、脏读、不可重复读、幻读这几类问题。

隔离级别	丢失更新	脏读	不可重复读	幻读
Read uncommitted	×	√	√	√
Read committed	×	×	√	√
Repeatable read（默认）	×	×	×	√
Serializable	×	×	×	×

备注 ： √代表可能出现 ， × 代表不会出现 。

Mysql 的数据库的默认隔离级别为 Repeatable read ， 查看方式：

```
show variables like 'tx_isolation';
```

```
mysql> show variables like 'tx_isolation';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)
```

5.3.3 InnoDB 的行锁模式

InnoDB 实现了以下两种类型的行锁。

- 共享锁（S）：又称为读锁，简称S锁，共享锁就是多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改。
- 排他锁（X）：又称为写锁，简称X锁，排他锁就是不能与其他锁并存，如一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁，包括共享锁和排他锁，但是获取排他锁的事务是可以对数据就行读取和修改。

对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X)；

对于普通SELECT语句，InnoDB不会加任何锁；

可以通过以下语句显示给记录集加共享锁或排他锁 。

```
共享锁（S）：SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE

排他锁（X） ：SELECT * FROM table_name WHERE ... FOR UPDATE
```

5.3.4 案例准备工作

```
create table test_innodb_lock(
    id int(11),
    name varchar(16),
    sex varchar(1)
)engine = innodb default charset=utf8;

insert into test_innodb_lock values(1,'100','1');
insert into test_innodb_lock values(3,'3','1');
insert into test_innodb_lock values(4,'400','0');
insert into test_innodb_lock values(5,'500','1');
insert into test_innodb_lock values(6,'600','0');
insert into test_innodb_lock values(7,'700','0');
insert into test_innodb_lock values(8,'800','1');
insert into test_innodb_lock values(9,'900','1');
insert into test_innodb_lock values(1,'200','0');
```

```
16
17 create index idx_test_innodb_lock_id on test_innodb_lock(id);
18 create index idx_test_innodb_lock_name on test_innodb_lock(name);
```

5.3.5 行锁基本演示

Session-1	Session-2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) 关闭自动提交功能	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) 关闭自动提交功能
mysql> select * from test_innodb_lock; +------+-------+------+ | id | name | sex | +------+-------+------+ | 1 | name1 | 1 | | 3 | 3 | 1 | | 4 | 400 | 0 | | 5 | 500 | 1 | | 6 | 600 | 0 | | 7 | 700 | 0 | | 8 | 800 | 1 | | 9 | 900 | 1 | | 1 | name2 | 0 | +------+-------+------+ 9 rows in set (0.00 sec) 可以正常的查询出全部的数据	mysql> select * from test_innodb_lock; +------+-------+------+ | id | name | sex | +------+-------+------+ | 1 | name1 | 1 | | 3 | 3 | 1 | | 4 | 400 | 0 | | 5 | 500 | 1 | | 6 | 600 | 0 | | 7 | 700 | 0 | | 8 | 800 | 1 | | 9 | 900 | 1 | | 1 | name2 | 0 | +------+-------+------+ 9 rows in set (0.00 sec) 可以正常的查询出全部的数据
mysql> select * from test_innodb_lock where id =3; +------+------+------+ | id | name | sex | +------+------+------+ | 3 | 3 | 1 | +------+------+------+ 1 row in set (0.00 sec) 查询id 为3的数据 ；	mysql> select * from test_innodb_lock where id =3; +------+------+------+ | id | name | sex | +------+------+------+ | 3 | 3 | 1 | +------+------+------+ 1 row in set (0.00 sec) 获取id为3的数据 ；
mysql> update test_innodb_lock set name = 'A1' where id = 3; Query OK, 1 row affected (0.01 sec) Rows matched: 1 Changed: 1 Warnings: 0 更新id为3的数据，但是不提交；	mysql> update test_innodb_lock set name = 'A2' where id = 3; 更新id为3 的数据， 出于等待状态
mysql> commit; Query OK, 0 rows affected (0.03 sec) 通过commit， 提交事务	mysql> update test_innodb_lock set name = 'A2' where id = 3; Query OK, 1 row affected (2.52 sec) Rows matched: 1 Changed: 1 Warnings: 0 解除阻塞，更新正常进行
以上， 操作的都是同一行的数据，接下来，演示不同行的数据 ：	
mysql> update test_innodb_lock set name = 'B1' where id = 3; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0 更新id为3数据，正常的获取到行锁 ， 执行更新 ；	mysql> update test_innodb_lock set name = 'C1' where id = 5; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0 由于与Session-1 操作不是同一行，获取当前行锁，执行更新；

5.3.6 无索引行锁升级为表锁

如果不通过索引条件检索数据，那么InnoDB将对表中的所有记录加锁，实际效果跟表锁一样。

查看当前表的索引 ： show index from test_innodb_lock ;

```
mysql> show index from test_innodb_lock\G;
*************************** 1. row ***************************
        Table: test_innodb_lock
   Non_unique: 1
     Key_name: idx_test_innodb_lock_id
 Seq_in_index: 1
  Column_name: id
    Collation: A
  Cardinality: 9
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
*************************** 2. row ***************************
        Table: test_innodb_lock
   Non_unique: 1
     Key_name: idx_test_innodb_lock_name
 Seq_in_index: 1
  Column_name: name
    Collation: A
  Cardinality: 9
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
2 rows in set (0.00 sec)
```

Session-1	Session-2

关闭事务的自动提交
`mysql> set autocommit = 0;`
`Query OK, 0 rows affected (0.00 sec)`
	关闭事务的自动提交
	`mysql> set autocommit = 0;`
	`Query OK, 0 rows affected (0.00 sec)`

执行更新语句 ：
`mysql> update test_innodb_lock set sex='2' where name = 400;`
`Query OK, 1 row affected (0.00 sec)`
`Rows matched: 1  Changed: 1  Warnings: 0`
	执行更新语句， 但处于阻塞状态：
	`mysql> update test_innodb_lock set sex='2' where id = 9;`

提交事务：
`mysql> commit ;`
`Query OK, 0 rows affected (0.03 sec)`
	解除阻塞，执行更新成功 ：
	`mysql> update test_innodb_lock set sex='2' where id = 9;`
	`Query OK, 1 row affected (8.27 sec)`
	`Rows matched: 1  Changed: 1  Warnings: 0`

	执行提交操作 ：
	`mysql> commit;`
	`Query OK, 0 rows affected (0.00 sec)`

由于 执行更新时 ， name字段本来为varchar类型， 我们是作为数组类型使用，存在类型转换，索引失效，最终行锁变为表锁 ；

5.3.7 间隙锁危害

当我们用范围条件，而不是使用相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据进行加锁； 对于键值在条件范围内但并不存在的记录，叫做 "间隙（ GAP ）" ， InnoDB也会对这个 "间隙" 加锁，这种锁机制就是所谓的 间隙锁（ Next-Key锁 ）。

示例 ：

Session-1	Session-2

关闭事务自动提交
`mysql> set autocommit = 0;`
`Query OK, 0 rows affected (0.00 sec)`
	关闭事务自动提交
	`mysql> set autocommit = 0;`
	`Query OK, 0 rows affected (0.00 sec)`

根据id范围更新数据
`mysql> update test_innodb_lock set name = '8888' where id <4 ;`
`Query OK, 3 rows affected (0.00 sec)`
`Rows matched: 3  Changed: 3  Warnings: 0`	

	插入id为2的记录， 出于阻塞状态 `mysql> insert into test_innodb_lock values(2,'1001','1');`

提交事务 ；
`mysql> commit;`
`Query OK, 0 rows affected (0.00 sec)`	

	解除阻塞 ， 执行插入操作 ：
	`mysql> insert into test_innodb_lock values(2,'1001','1');`
	`Query OK, 1 row affected (3.44 sec)`

	提交事务 ：

5.3.8 InnoDB 行锁争用情况

```
show  status like 'innodb_row_lock%';
```

```
mysql> show  status like 'innodb_row_lock%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| Innodb_row_lock_current_waits | 0     |
| Innodb_row_lock_time          | 21538 |
| Innodb_row_lock_time_avg      | 10769 |
| Innodb_row_lock_time_max      | 19536 |
| Innodb_row_lock_waits         | 2     |
+-------------------------------+-------+
5 rows in set (0.00 sec)
```

```
Innodb_row_lock_current_waits: 当前正在等待锁定的数量

Innodb_row_lock_time: 从系统启动到现在锁定总时间长度

Innodb_row_lock_time_avg:每次等待所花平均时长

Innodb_row_lock_time_max:从系统启动到现在等待最长的一次所花的时间

Innodb_row_lock_waits: 系统启动后到现在总共等待的次数


当等待的次数很高，而且每次等待的时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手制定优化计划。

```

5.3.9 总结

InnoDB存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面带来了性能损耗可能比表锁会更高一些，但是在整体并发处理能力方面要远远由于MyISAM的表锁的。当系统并发量较高的时候，InnoDB的整体性能和MyISAM相比就会有比较明显的优势。

但是，InnoDB的行级锁同样也有其脆弱的一面，当我们使用不当的时候，可能会让InnoDB的整体性能表现不仅不能比MyISAM高，甚至可能会更差。

优化建议：

- 尽可能让所有数据检索都能通过索引来完成，避免无索引行锁升级为表锁。
- 合理设计索引，尽量缩小锁的范围
- 尽可能减少索引条件，及索引范围，避免间隙锁
- 尽量控制事务大小，减少锁定资源量和时间长度
- 尽可使用低级别事务隔离（但是需要业务层面满足需求）

6. 常用SQL技巧

6.1 SQL执行顺序

编写顺序

```
SELECT DISTINCT
    <select list>
FROM
    <left_table> <join_type>
JOIN
    <right_table> ON <join_condition>
WHERE
    <where_condition>
GROUP BY
    <group_by_list>
HAVING
    <having_condition>
ORDER BY
    <order_by_condition>
LIMIT
    <limit_params>
```

执行顺序

```
FROM    <left_table>

ON      <join_condition>

<join_type>     JOIN    <right_table>

WHERE       <where_condition>

GROUP BY    <group_by_list>
```

```
10
11 HAVING      <having_condition>
12
13 SELECT DISTINCT     <select list>
14
15 ORDER BY    <order_by_condition>
16
17 LIMIT       <limit_params>
```

6.2 正则表达式使用

正则表达式（Regular Expression）是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串。

符号	含义
^	在字符串开始处进行匹配
$	在字符串末尾处进行匹配
.	匹配任意单个字符, 包括换行符
[...]	匹配出括号内的任意字符
[^...]	匹配不出括号内的任意字符
a*	匹配零个或者多个a(包括空串)
a+	匹配一个或者多个a(不包括空串)
a?	匹配零个或者一个a
a1|a2	匹配a1或a2
a(m)	匹配m个a
a(m,)	至少匹配m个a
a(m,n)	匹配m个a 到 n个a
a(,n)	匹配0到n个a
(...)	将模式元素组成单一元素

```
1 select * from emp where name regexp '^T';
2
3 select * from emp where name regexp '2$';
4
5 select * from emp where name regexp '[uvw]';
```

6.3 MySQL 常用函数

数字函数

函数名称	作 用
ABS	求绝对值
SQRT	求二次方根
MOD	求余数
CEIL 和 CEILING	两个函数功能相同，都是返回不小于参数的最小整数，即向上取整
FLOOR	向下取整，返回值转化为一个BIGINT
RAND	生成一个0~1之间的随机数，传入整数参数是，用来产生重复序列
ROUND	对所传参数进行四舍五入
SIGN	返回参数的符号
POW 和 POWER	两个函数的功能相同，都是所传参数的次方的结果值
SIN	求正弦值
ASIN	求反正弦值，与函数 SIN 互为反函数
COS	求余弦值
ACOS	求反余弦值，与函数 COS 互为反函数
TAN	求正切值
ATAN	求反正切值，与函数 TAN 互为反函数
COT	求余切值

字符串函数

函数名称	作 用
LENGTH	计算字符串长度函数，返回字符串的字节长度
CONCAT	合并字符串函数，返回结果为连接参数产生的字符串，参数可以使一个或多个
INSERT	替换字符串函数
LOWER	将字符串中的字母转换为小写
UPPER	将字符串中的字母转换为大写
LEFT	从左侧字截取符串，返回字符串左边的若干个字符
RIGHT	从右侧字截取符串，返回字符串右边的若干个字符
TRIM	删除字符串左右两侧的空格
REPLACE	字符串替换函数，返回替换后的新字符串
SUBSTRING	截取字符串，返回从指定位置开始的指定长度的字符换
REVERSE	字符串反转（逆序）函数，返回与原始字符串顺序相反的字符串

日期函数

函数名称	作 用
CURDATE 和 CURRENT_DATE	两个函数作用相同，返回当前系统的日期值
CURTIME 和 CURRENT_TIME	两个函数作用相同，返回当前系统的时间值
NOW 和 SYSDATE	两个函数作用相同，返回当前系统的日期和时间值
MONTH	获取指定日期中的月份
MONTHNAME	获取指定日期中的月份英文名称
DAYNAME	获取指定日期对应的星期几的英文名称
DAYOFWEEK	获取指定日期对应的一周的索引位置值
WEEK	获取指定日期是一年中的第几周，返回值的范围是否为 0∽52 或 1∽53
DAYOFYEAR	获取指定日期是一年中的第几天，返回值范围是1~366
DAYOFMONTH	获取指定日期是一个月中是第几天，返回值范围是1~31
YEAR	获取年份，返回值范围是 1970∽2069
TIME_TO_SEC	将时间参数转换为秒数
SEC_TO_TIME	将秒数转换为时间，与TIME_TO_SEC 互为反函数
DATE_ADD 和 ADDDATE	两个函数功能相同，都是向日期添加指定的时间间隔
DATE_SUB 和 SUBDATE	两个函数功能相同，都是向日期减去指定的时间间隔
ADDTIME	时间加法运算，在原始时间上添加指定的时间
SUBTIME	时间减法运算，在原始时间上减去指定的时间
DATEDIFF	获取两个日期之间间隔，返回参数 1 减去参数 2 的值
DATE_FORMAT	格式化指定的日期，根据参数返回指定格式的值
WEEKDAY	获取指定日期在一周内的对应的工作日索引

聚合函数

函数名称	作用
MAX	查询指定列的最大值
MIN	查询指定列的最小值
COUNT	统计查询结果的行数
SUM	求和，返回指定列的总和
AVG	求平均值，返回指定列数据的平均值