第五节 Mysql性能优化整体分析和总结

Q&A

1. MVCC为什么不能解决幻读（快照读和当前读的问题）
2. Select * from user where id>1 For update ，假设id是主键索引，应该加什么锁。

   如果查询条件是二级索引，默认会有一个next key lock

   如果是主键索引， 加行锁

   针对满足条件的记录行加Record Lock
3. Next Key Lock 加锁区间的问题

   针对二级索引加锁的情况下，每个记录行都会默认存在一个next -key lock

   ```
   select * from user where class_id=2 for update;
   ```

   (-∞，1]　(1，2]　(2，8]　(8，10]，(10，+∞]

   (1，2] 、 (2，8]

   ```
   select * from user where class_id>1;
   ```

   加锁范围 (1，2]　(2，8]　(8，10]，(10，+∞]

4. ReadLock

```
select * from user where id=1 for update;
```

如果id=1这条数据不存在，就会自动加Next Key Lock

加锁 就一定会面临死锁的问题

所以，导致MySQL 出现死锁的几个要素：

1. 两个或者两个以上事务
2. 锁资源只能被同一个事务持有或者多个事务竞争的锁是不兼容的，比如排他锁和共享锁、排他锁和排他锁。
3. 每个事务都已经持有锁并且申请新的锁
4. 事务之间因为持有锁和申请锁导致彼此循环等待

死锁的演示

下面我们来通过两个案例来演示死锁的问题。

> 按照下图所示的顺序执行脚本。

Session 1	Session 2
begin; select * from user where id=3 for update;	
	begin; delete from user where id=4;
update user set name='mimi' where id=4;	
	delete from user where id=3;

在上述脚本运行后，会在第一个事务中看到下面这样的日志

```
update user set name='mimi' where id=4
> 1213 - Deadlock found when trying to get lock; try restarting transaction
> 时间: 2.475s
```

Mysql立刻检测到了死锁，并且事务1马上退出了。

为什么可以直接检测到呢？是因为死锁的发生需要满足一定的条件，所以在发生死锁时，InnoDB一般都能通过算法（wait-for graph）自动检测到。

事务竞争锁的超时时间

另外还有一个要注意的点，就是事务竞争锁的时候，如果发现当前有事务正在持有锁，那么它不会一直等下去。

默认情况下，它会等待50s的时间，如果50s还没有办法竞争到锁，则直接释放锁资源。

> 演示逻辑：

Session 1	**Session 2**
begin; select * from user where id=3 for update;	
	select * from user where id=3 for update;

上面这个的执行结果如下：

```
select * from user where id=3  for update
> 1205 - Lock wait timeout exceeded; try restarting transaction
> 时间: 51.011s
```

事务竞争锁超时的时间，是通过下面这个属性来控制的

```
show VARIABLES like 'innodb_lock_wait_timeout';
```

所以，一个事务的锁的释放，有三种情况：

1. 事务结束（commit/rollback）
2. 客户端连接断开
3. 事务竞争锁超时（50s）

查看锁的日志

如果我们需要定位当前数据库吞吐量下降的原因是否是锁竞争或者死锁导致的，应该怎么去判断呢？

在Mysql中提供了一些属性，可以看到整个数据库锁的使用情况。

```
show status like 'innodb_row_lock_%';
```

variable_name	value
Innodb_row_lock_current_waits	0
Innodb_row_lock_time	1026186
Innodb_row_lock_time_avg	15548
Innodb_row_lock_time_max	51933
Innodb_row_lock_waits	66

上述参数说明如下：

- 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的总时间长度，单位ms；
- Innodb_row_lock_time_avg ：每次等待锁的平均时间；
- 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 ：从系统启动到现在总共等待的次数。

上面只是一个总的信息，如果想看到更加详细的数据，Mysql提供了下面几个命令:

- 当前运行的所有事务 ， 还有具体的SQL语句;

```
select * from information_schema.INNODB_TRX;  -- 当前运行的所有事务 ， 还有具体的语句
```

- 查看处于锁等待的详细信息

```
select * from sys.innodb_lock_waits;   -- 锁等待的对应关系
```

- 查看正在处理中的事务

```
select * from information_schema.PROCESSLIST p where p.state<>'';

select * from performance_schema.data_locks; //锁定数据查询
```

基于这些操作命令，可以定位到死锁的源头，比如**select * from sys.innodb_lock_waits;**这个语句中的最后两列，提供了两个属性:

- sql_kill_blocking_query kill掉这个query语句的线程
- sql_kill_blocking_connection kill条这个阻塞的链接

通过提供的参考命令，可以直接把锁等待的线程终止。

也可以通过**select * from information_schema.INNODB_TRX;**这个语句中查询到的**trx_mysql_thread_id**，就是这个事务的线程id，通过**kill trx_mysql_thread_id**来终止。

当然，死锁的问题不能每次都靠kill线程来解决，这是治标不治本的行为。我们应该尽量在应用端，也就是在编码的过程中避免。

有哪些可以避免死锁的方法呢?

如何防止死锁

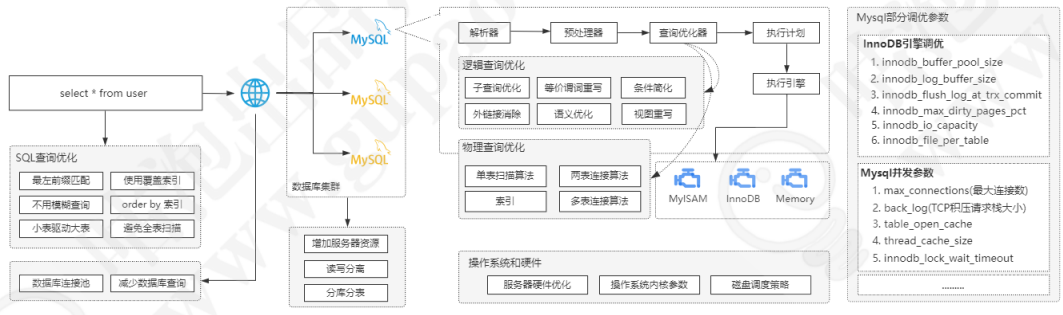
- 在程序中，操作多张表时，尽量以相同的顺序来访问（避免形成等待环路）；
- 尽量避免大事务，占有的资源锁越多，越容易出现死锁。建议拆成小事务。
- 降低隔离级别。如果业务允许，将隔离级别调低也是较好的选择，比如将隔离级别从RR调整为RC，可以避免掉很多因为gap锁造成的死锁。
- 为表添加合理的索引。防止没有索引出现表锁，出现的死锁的概率会突增

Mysql做优化

物化视图（mysql， 保存统计结果的表）

https://gper.club/articles/7e7e7f7ff4g5egc2g63

https://gper.club/articles/7e7e7f7ff3g5agc3g6b



SQL优化

SQL分析主要有两个切入点：

1. EXPLAIN 执行计划分析
2. 数据库慢SQL查询

EXPLAIN 执行计划分析

MySQL 提供了一个 EXPLAIN 命令, 它可以对 **SELECT** 语句进行分析, 并输出 **SELECT** 执行计划的详细信息, 这些信息给到开发人员参考并作出优化的方向。

为了更好的展示Explain的效果，首先来初始化一个SQL脚本

```
DROP TABLE IF EXISTS course;
CREATE TABLE `course` (
 `cid` int(3) DEFAULT NULL,
 `cname` varchar(20) DEFAULT NULL,
 `tid` int(3) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS teacher;
CREATE TABLE `teacher` (
 `tid` int(3) DEFAULT NULL,
 `tname` varchar(20) DEFAULT NULL,
 `tcid` int(3) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS teacher_contact;
CREATE TABLE `teacher_contact` (
 `tcid` int(3) DEFAULT NULL,
 `phone` varchar(200) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

INSERT INTO `course` VALUES ('1', 'mysql', '1');
INSERT INTO `course` VALUES ('2', 'jvm', '1');
INSERT INTO `course` VALUES ('3', 'juc', '2');
INSERT INTO `course` VALUES ('4', 'spring', '3');

INSERT INTO `teacher` VALUES ('1', 'qingshan', '1');
INSERT INTO `teacher` VALUES ('2', 'huihui', '2');
```

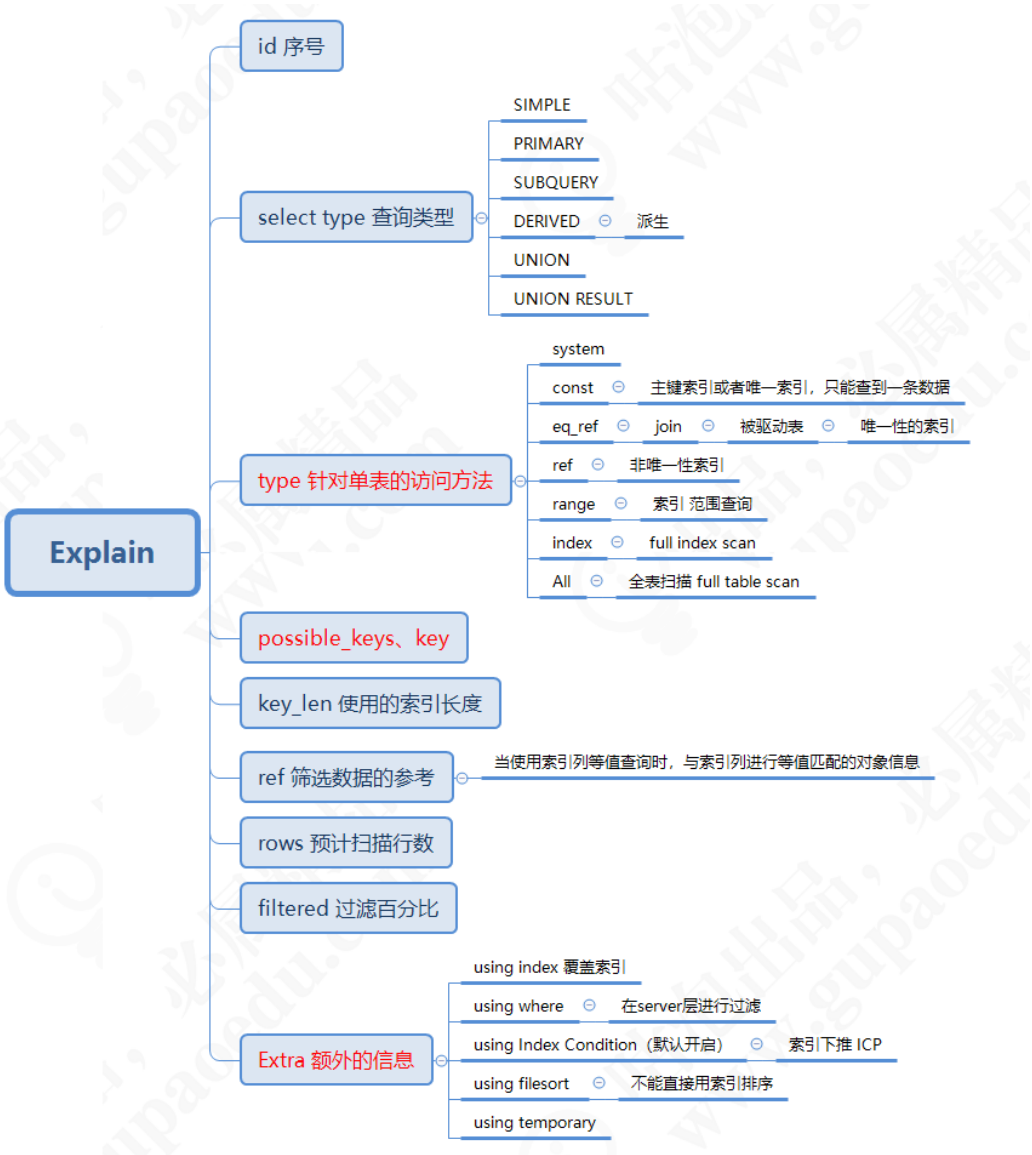
```
INSERT INTO `teacher` VALUES ('3', 'mic', '3');

INSERT INTO `teacher_contact` VALUES ('1', '13688888888');
INSERT INTO `teacher_contact` VALUES ('2', '18166669999');
INSERT INTO `teacher_contact` VALUES ('3', '17722225555');
```

接着我们执行下面一个语句

```
explain select * from course;
```

在Explain中可以看到很多的数据列，下面分别来说明一下每个数据列的含义。



> 官方文档： https://dev.mysql.com/doc/refman/8.0/en/explain-output.html

id

id是查询序列编号，每张表都是单独访问的，一个SELECT就会有一个序号，比如下面这样一个sql。

```
select tc.phone from teacher_contact tc where tcid=
(select tcid from teacher t where t.tid=
(select c.tid from course c where c.cname='mysql'));
```

对应的执行计划如下。

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	tc	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where
2	SUBQUERY	t	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where
3	SUBQUERY	c	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25.00	Using where

查询顺序：course c——teacher t——teacher_contact tc。

先查课程表，再查老师表，最后查老师联系方式表。子查询只能以这种方式进行，只有拿到内层的结果之后才能进行外层的查询。

还有一种查询的情况，id的值相同，查询语句如下。

```
-- 查询课程ID为2，或者联系表ID为3的老师
EXPLAIN
SELECT t.tname,c.cname,tc.phone
FROM teacher t, course c, teacher_contact tc
WHERE t.tid = c.tid
AND t.tcid = tc.tcid
AND (c.cid = 2
OR tc.tcid = 3);
```

执行计划如下图所示。

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100.00	(Null)
1	SIMPLE	c	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25.00	Using where; Using
1	SIMPLE	tc	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where; Using

这种连接查询的情况下，id值相同，表的查询顺序是从上往下执行。

查询的顺序是teacher t（3条）——course c（4条）——teacher_contact tc（3条）。

在连接查询中，先查询的叫做驱动表，后查询的叫做被驱动表。

应该先查小表（得到结果少的表）还是大表（得到结果多的表）？我们肯定要把小表放在前面查询，因为它的中间结果最少。 （小表驱动大表的思想）

因此，总结来看： **ID不同的先大后小，ID相同的从上往下**

select_type

select_type表示查询类型，它的常用取值类型如下：

- SIMPLE, 表示此查询不包含 UNION 查询或子查询
- PRIMARY, 表示此查询是最外层的查询
- UNION, 表示此查询是 UNION 的第二或随后的查询
- DEPENDENT UNION, UNION 中的第二个或后面的查询语句, 取决于外面的查询
- UNION RESULT, UNION 的结果
- SUBQUERY, 子查询中的第一个 SELECT
- DEPENDENT SUBQUERY: 子查询中的第一个 SELECT, 取决于外面的查询. 即子查询依赖于外层查询的结果.

下面对常规类型做一个简单的说明。

SIMPLE类型

简单查询，不包含子查询，不包含关联查询union。

```
EXPLAIN SELECT * FROM teacher;
```

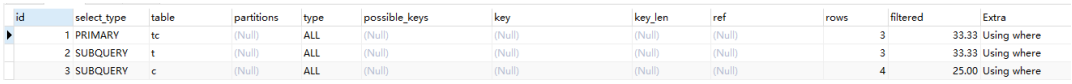
执行计划如下图所示：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	teacher	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100.00	(Null)

如果再执行下面这种包含多个子查询的语句，

```
select tc.phone from teacher_contact tc where tcid=
(select tcid from teacher t where t.tid=
(select c.tid from course c where c.cname='mysql'));
```

执行计划如下：



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	tc	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where
2	SUBQUERY	t	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where
3	SUBQUERY	c	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25.00	Using where

从上图的执行计划中可以看到，它有两种类型：

1. primary， 表示SQL语句中的主查询，也就是最外层的查询。
2. subquery， 子查询中的所有内查询，都是subquery类型。

Derived类型

DERIVED类型，衍生查询，表示的到最终查询结果之前会用到临时表，比如

```
-- 查询ID为1或2的老师教授的课程
EXPLAIN SELECT cr.cname
FROM (
SELECT * FROM course WHERE tid = 1
UNION
SELECT * FROM course WHERE tid = 2
) cr;
```

执行计划如下图所示：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	100.00	(Null)
2	DERIVED	course	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25.00	Using where
3	UNION	course	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25.00	Using where
(Null)	UNION RESULT	<union2,3>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Using temporary

注意看，id=2的查询类型是DERIVED，id=3的查询类型是UNION。在这种查询中。

1. 先执行UNION右边的表，也就是（SELECT * FROM course WHERE tid=2），再执行左边的表**SELECT * FROM course WHERE tid = 1**，此时左边表的类型是DERIVED。

2. UNION RESULT，显示那些表之间存在UNION查询，<union2,3> 表示id=2和id=3的查询存在union。

type连接类型

type 字段比较重要, 它提供了判断查询是否高效的重要依据依据，通过 type 字段, 我们判断此次查询是 全表扫描 还是 索引扫描 等。

type常见的取值类型有很多，比如 system/const/eq_ref/ref/range/index/all等等，下面详细展开说明一下

const

const: 针对主键或唯一索引的等值查询扫描, 最多只返回一行数据. const 查询速度非常快, 因为它仅仅读取一次即可。

比如下面这个sql，根据主键id=4查询user表的数据。

```
explain select * from user where id=4;
```

执行计划如下：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	const	PRIMARY	PRIMARY	4	const	1	100.00	(Null)

system

system: 表中只有一条数据. 这个类型是特殊的 const 类型。

对于MyISAM、Memory的表，只查询到一条记录，也是system。

```
CREATE TABLE `table1` (
 `id` int NOT NULL,
 `a` int DEFAULT NULL,
 `b` int DEFAULT NULL,
 `c` int DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
 -- 比如在上面的表中，创建一条记录，这个时候执行下面的
explain，就会的到system类型
 explain select * from table1
```

eq_ref

在多表连接查询中，**被驱动表**通过**唯一索引**(UNIQUE或PRIMARY KEY)进行访问的时候，**被驱动表**的访问方式就是**eq_ref**

通过下面这条sql进行演示

```
-- ALTER TABLE teacher_contact DROP PRIMARY KEY;
ALTER TABLE teacher_contact ADD PRIMARY KEY(tcid);  --增加主
键索引

explain select t.tcid from teacher t,teacher_contact tc where t.tcid
= tc.tcid;
```

执行计划如下：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100.00	Using where
1	SIMPLE	tc	(Null)	eq_ref	PRIMARY	PRIMARY	4	mysql-example.t.tcid	1	100.00	Using index

ref

查询用到了**非唯一性索引**，或者关联操作只使用了索引的**最左前缀**，则连接类型是**ref**。

下面通过sql演示一下这种类型：

```
-- ALTER TABLE teacher DROP INDEX idx_tcid;
ALTER TABLE teacher ADD INDEX idx_tcid (tcid);  --增加普通索引
explain SELECT * FROM teacher where tcid = 3;
```

执行计划如下

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	teacher	(Null)	ref	idx_tcid	idx_tcid	5	const	1	100.00	(Null)

range

索引范围扫描。

如果where后面是 between and 或 <或 > 或 >= 或 <=或in这些，type类型就为range。

下面通过sql演示一下range范围检索

```
-- ALTER TABLE teacher DROP INDEX idx_tid;
ALTER TABLE teacher ADD INDEX idx_tid (tid);
```

执行范围查询（字段上有普通索引）：

```
EXPLAIN SELECT * FROM teacher t WHERE t.tid <3;
-- 或
EXPLAIN SELECT * FROM teacher t WHERE tid BETWEEN 1 AND 2;
```

IN查询也是range（字段有主键索引）

```
EXPLAIN SELECT * FROM teacher_contact t WHERE tcid in (1,2,3);
```

> mysql中in中的参数个数不受限制，但是sql本身的长度会受到限制，默认是64M
>
> https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_max_allowed_packet

> in通常是走索引的，当in后面的参数个数较多的情况下，就不会再走索引，直接走全表扫描。

index

Full Index Scan，全索引扫描(Full Index Scan)。

index 和ALL最大的区别是，index类型只扫描索引树即可，所以它要比ALL的查询效率要快。

比如在teacher表中，tid是主键，当我们只查询主键列tid时。

```
EXPLAIN SELECT tid FROM teacher;
```

执行计划如下：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	teacher	(Null)	index	(Null)	idx_tid	5	(Null)	3	100.00	Using index

all

Full Table Scan，如果没有索引或者没有用到索引，type就是ALL。代表全表扫描，比如下面这种情况必然是全表扫描。

```
EXPLAIN SELECT * FROM teacher;
```

null

不用访问表或者索引就能得到结果，例如：

```
EXPLAIN select 1 from dual;
```

possible_key、key

possible_key表示可能用到的索引，key表示实际用到的索引。如果这一列为空，表示没有用到索引。

possible_key可以有一个或者多个，当然，可能用到索引不代表一定用到索引。

下面演示一下这种索引的使用。

```
--首先需要创建一张表，并且建立一个联合索引
CREATE TABLE `user_innodb` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `gender` tinyint(1) DEFAULT NULL,
  `phone` varchar(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `name_key` (`name`,`phone`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=3000002 DEFAULT
CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

执行下面这个sql

```
explain select phone from user_innodb where phone='126';
```

执行计划如下：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user_innodb	(Null)	index	name_key	name_key	1070	(Null)	2991533	10.00	Using where; Using ir

这里用到了覆盖索引，也就是是直接在二级索引树上就获得了数据，所以possible_keys和key对应的是联合索引的名字(name_key)。

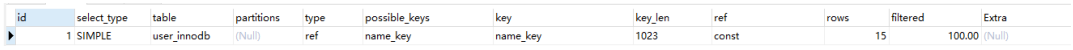
key_len

索引的长度（使用的字节数）。跟索引字段的类型、长度有关。

比如我们执行下面这个sql

```
explain select * from user_innodb where name ='李翩';
```

的到的执行计划如下



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user_innodb	(Null)	ref	name_key	name_key	1023	const	15	100.00	(Null)

可以看到，key_len是1023。

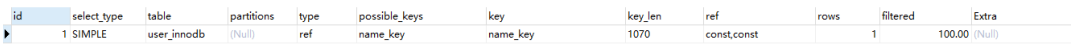
我们来看下这个1023是如何计算的，首先在name和phone上建立了联合索引，而name定义的长度是255、phone的长度是11。

这里索引只用到了name字段，所以定义长度是255，而在utf8mb4编码中，一个字符占4个字节，所以255*4=1020。

另外，使用可变长字段varchar，需要额外增加2个字节，允许NULL需要额外增加1个字节，所以一共是1023个字节。

key_len越长，表示使用的索引范围越广，比如我们再使用下面这个sql执行一次

```
explain select * from user_innodb where name ='李翮' and phone='13002811115';
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user_innodb	(Null)	ref	name_key	name_key	1070	const,const	1	100.00	(Null)

可以看到，key_len=1070， 因为phone是varchar类型长度为11，字节长度=11*4 =44 再加三个字节一共是47。

所以一共是1070个字节。

rows

MySQL认为扫描多少行才能返回请求的数据，是一个预估值，一般来说行数越少越好。

filtered

Filtered表示返回结果的行数占需读取行数的百分比 ， 它只对index和all的扫描有效。

如果比例很低，说明存储引擎层返回的数据需要经过大量过滤，这个是会消耗性能的，需要关注。

> https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#explain_filtered

ref

使用哪个列或者常数和索引一起从表中筛选数据。

```
explain select * from user_innodb where name ='李翩' and phone='13002811115';
```

的到的执行计划如下：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user_innodb	(Null)	ref	name_key	name_key	1070	const,const	1	100.00	(Null)

上面的ref，用到了两个常量进行数据的筛选。

Extra

EXplain 中的很多额外的信息会在 Extra 字段显示, 常见的有以下几种内容:

- Using filesort
  当 Extra 中有 **Using filesort** 时, 表示 MySQL 需额外的排序操作, 不能通过索引顺序达到排序效果. 一般有 **Using filesort**, 都建议优化去掉, 因为这样的查询 CPU 资源消耗大.

  ```
  --先删除user_innodb中的联合索引，再执行下面的语句。
  explain select * from user_innodb order by name;
  ```

  的到的执行计划如下：

  信息 结果 1 剖析 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user_innodb	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	2991533	100.00	Using filesort

- Using index
  "覆盖索引扫描", 表示查询在索引树中就可查找所需数据, 不用扫描表数据文件, 往往说明性能不错。

  ```
  --下面查询的tid直接可以从索引中检索到
  EXPLAIN SELECT tid FROM teacher ;
  ```

- Using temporary
  查询有使用临时表, 一般出现于排序, 分组和多表 join 的情况, 查询效率不高, 建议优化。

  下面这个 sql，就会用到 Using temporary

```
EXPLAIN select t.tid from teacher t join course c on t.tid = c.tid group by t.tid;
```

数据库慢查询日志

在一个数据库中，有很多的应用程序来执行sql，我们一般不会等到sql查询慢了才去处理，而是希望能够有一个慢查询的监控，在应用程序中，有两种方式监控。

1. Druid包提供了慢查询的监控
2. Mysql提供了慢查询日志

Mysql慢查询开启

我们可以通过下面这个sql来获得慢查询的信息：

```
show variables like 'slow_query%';
```

的到的结果如下：

Variable_name	Value
slow_query_log	OFF
slow_query_log_file	/var/lib/mysql/localhost-slow.log

show_query_log： 默认是关闭的（开启它需要代价，因为需要写文件）

show_query_log_file: 慢日志存放的目录

除了这两个参数之外，我们还需要告诉Mysql，sql执行速度达到什么阈值的时候才认为是慢查询呢?

```
show variables like '%long_query%';
```

默认是10s，也就是记录查询时间大于10s的sql，这三个参数可以在my.cnf文件中修改

```
slow_query_log = ON
long_query_time=2
slow_query_log_file =/var/lib/mysql/localhost-slow.log
```

慢查询日志分析

下面我们查询user_innodb这个表（300W数据），在没有任何索引的情况下

```
SELECT * FROM user_innodb where phone='180';
```

这个查询语句耗时比较长，于是我们在/var/lib/mysql/localhost-show.log目录下的文件，内容如下

```
# Time: 2022-03-08T19:15:54.086977Z
# User@Host: root[root] @  [192.168.8.16]  Id:    12
# Query_time: 2.800553  Lock_time: 0.000517 Rows_sent: 3000001
Rows_examined: 3000001
SET timestamp=1646766951;
select * from user_innodb;
```

如果文件内容较大，通过肉眼来分析很明显不合适，所以我们其实可以通过日志监控的方式，或者是用mysql提供的mysqldumpslow工具来分析。

> mysqldumpshow是Mysql提供的分析工具，
> 在/usr/bin/mysqldumpshow目录下
>
> https://dev.mysql.com/doc/refman/8.0/en/mysqldumpslow.html

例如：查询用时最多的10条慢SQL：

```
mysqldumpslow -s t -t 10 -g 'select' /var/lib/mysql/localhost-slow.log
```

其中-s 表示如何排序，t表示查询时间或者平均查询时间； -t 查询条数； -g 相当于grep，从目标文件中匹配的关键字。

得到的结果如下：

```
[root@localhost bin]# mysqldumpslow -s t -t 10 -g 'select' /var/lib/mysql/localhost-slow.log

Reading mysql slow query log from /var/lib/mysql/localhost-slow.log
Count: 1  Time=23.23s (23s)  Lock=0.00s (0s)  Rows=101.0 (101), root[root]@[192.168.8.16]
  use mysql-example;
  SET timestamp=N;
  SELECT
  TIMESTAMPDIFF(YEAR, birth, CURDATE()) as age,
  COUNT(N) as num
  FROM
  employee
  GROUP BY
  age
```

参数说明如下：

- Count代表这个SQL执行了多少次；
- Time代表执行的时间，括号里面是累计时间；
- Lock表示锁定的时间，括号是累计；
- Rows表示返回的记录数，括号是累计。

当然，有的时候查询慢，不一定是SQL语句的问题，也有可能是服务器状态的问题。所以我们也要掌握一些查看服务器和存储引擎状态的命令。

BinLog和RedoLog

binlog和RedoLog

binlog以事件的形式记录了所有的DDL和DML语句（因为它记录的是操作而不是数据值，属于逻辑日志），可以用来做主从复制和数据恢复。

跟redo log不一样，它的文件内容是可以追加的，没有固定大小限制。

开启binlog

修改/etc/my.cnf文件，增加下面的配置。

```
-- binlog的文件名称
log-bin=mysql-bin
-- binlog内容格式，可设置日志三种格式：STATEMENT、ROW、MIXED 。
binlog-format=Row
```

binlog-format有三种格式：

- STATMENT模式(默认)：基于SQL语句的复制(statement-based replication, SBR)，每一条会修改数据的sql语句会记录到binlog中。
  优点：不需要记录每一条SQL语句与每行的数据变化，这样子binlog的日志也会比较少，减少了磁盘IO，提高性能。
- ROW:``基于行的复制(row-based replication, RBR)格式：不记录每一条SQL语句的上下文信息，仅需记录哪条数据被修改了，修改成了什么样子了。
  优点：不会出现某些特定情况下的存储过程、或function、或trigger的调

用和触发无法被正确复制的问题。
缺点：会产生大量的日志，尤其是alter table的时候会让日志暴涨。

- MIXED:``混合模式复制(mixed-based replication, MBR)：以上两种模式的混合使用，Mysql根据执行的SQL选择日志保存方式。

通过

```
show variables like 'log_%'
```

可以获得如下信息

Variable_name	Value
log_bin	ON
log_bin_basename	/var/lib/mysql/mysql-bin
log_bin_index	/var/lib/mysql/mysql-bin.index
log_bin_trust_function_creators	OFF
log_bin_use_v1_row_events	OFF
log_error	/var/log/mysqld.log
log_error_services	log_filter_internal; log_sink_internal
log_error_suppression_list	
log_error_verbosity	2

```
--执行一条更新语句
update user_innodb set name='mic' where name='蒋茶';
--查看binlog日志，
show binary logs
```

此时，在var/lib/mysql/目录下，会生成一个mysql-bin.000001的binlog文件。

Log_name	File_size	Encrypted
mysql-bin.000001	1012	No

接着，我们在/usr/bin目录下，执行下面这个命令，就可以打开binlog文件的内容。

```
mysqlbinlog --base64-output=decode-rows -v
/var/lib/mysql/mysql-bin.000001
```

开启BinLog之后的影响

前面说过，binlog主要是用来做主从数据同步的，意味着当我们对数据进行更新操作的时候，必须要把这个更新的动作写入到binlog中，这样才能实现数据同步。

开启binlog之后，在事务提交时，意味着innoDB就有两份数据要写。

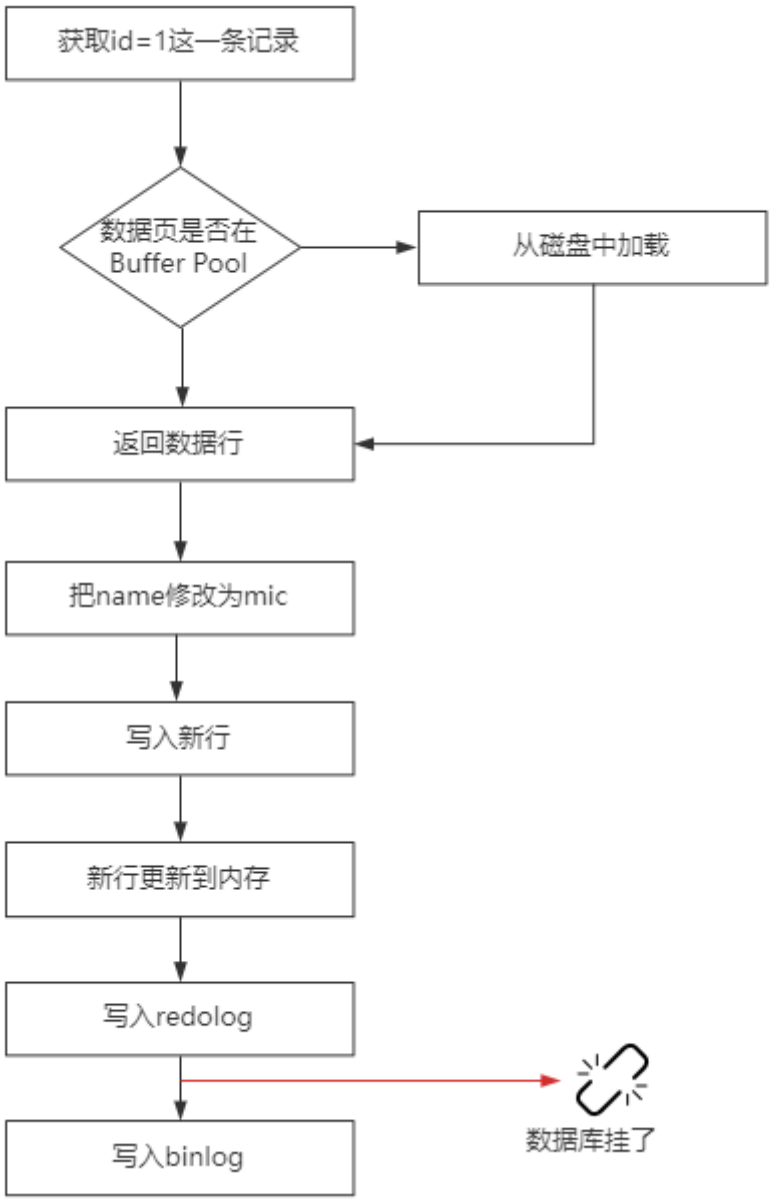
1. 写Redo Log
2. 写BinLog

在主从集群中，主库是通过Redo log来进行数据恢复，从库是通过binlog来实现数据同步。这里就存在一个问题，我们怎么保证redo log和bin log的数据一致性呢?

因为有可能出现redo log写入成功，但是在写binlog的时候数据库挂了，那么这个时候主从数据库就存在数据不一致了。

```
update teacher set name='Mic' where id=1;
```

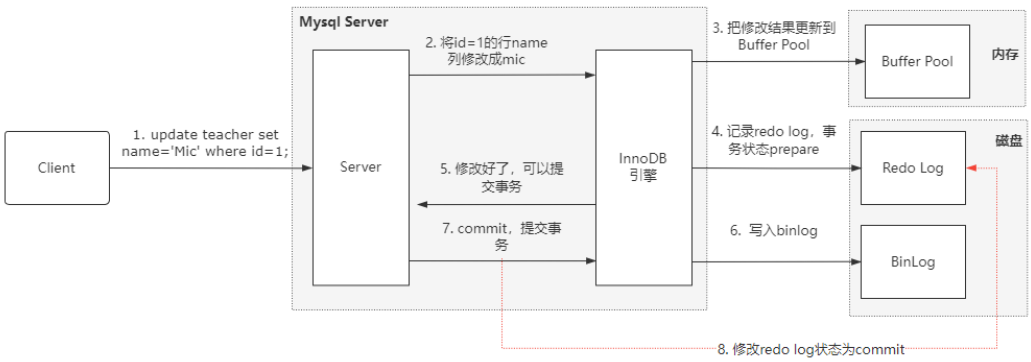
这个语句的执行流程如下。



为了解决这个问题，innoDB设计了两阶段提交。

二阶段提交大家已经不陌生了，在Mysql中支持xa协议，就是一种二阶段提交来实现数据一致性的方法。

binlog和redolog的二阶段提交流程如下：



两阶段提交的流程是：

- prepare阶段，写redo log；
- commit阶段，写binlog并且将redo log的状态改成commit状态；

mysql发生崩溃恢复的过程中，会根据redo log日志，结合 binlog 记录来做事务回滚：

- 如果redo log 和 binlog都存在，逻辑上一致，那么提交事务；
- 如果redo log存在而binlog不存在，逻辑上不一致，那么回滚事务；

所以在写两个日志的情况下，binlog就充当了一个事务的协调者。通知InnoDB来执行prepare或者commit或者rollback。

如果第⑥步写入binlog失败，二阶段事务就不会提交，那么此时这个数据就会被回滚！

简单地来说，这里有两个写日志的操作，类似于分布式事务，不用两阶段提交，就不能保证都成功或者都失败。