# MySQL

MySQL里经常说到的WAL技术，WAL的全称是Write-Ahead Logging，它的关键点就是先写日志，再写磁盘。具体来说，当有一条记录需要更新的时候，InnoDB引擎就会先把记录写到redo log（粉板）里面，并更新内存，这个时候更新就算完成了。同时，InnoDB引擎会在适当的时候，将这个操作记录更新到磁盘里面，而这个更新往往是在系统比较空闲的时候做。

需要注意的是，操作的内存中的记录所在的页都是从磁盘复制过来的缓冲页。这个页被修改就成为脏页，因为与磁盘对应的页的内容不一样。

1. redo log是InnoDB引擎特有的；binlog是MySQL的Server层实现的，所有引擎都可以使用。
2. redo log是物理日志，记录的是“在某个数据页上做了什么修改”；binlog是逻辑日志，记录的是这个语句的原始逻辑，比如“给ID=2这一行的c字段加1 ”。
3. redo log是循环写的，空间固定会用完；binlog是可以追加写入的。“追加写”是指binlog文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

二阶段提交:将redo log的写入拆成了两个步骤：prepare和commit，这就是"两阶段提交"。

在redo log处于prepare阶段之后，然后写入binlog，之后redo log再提交。

**先写redo log后写binlog**。则redo log写完之后，系统崩溃，由于binlog没写完就crash了，这时候binlog里面就没有记录这个语句。导致这个语句的binlog丢失

**先写binlog后写redo log**。则可能redo log并没有提交导致binlog中存储的数据是无效的。

MVCC通过undo log日志实现。

而MySQL 5.6 引入的索引下推优化（index condition pushdown)， 可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

可重复读下，对于一个事务视图来说，除了自己的更新总是可见以外，有三种情况：

1. 版本未提交，不可见；
2. 版本已提交，但是是在视图创建后提交的，不可见；
3. 版本已提交，而且是在视图创建前提交的，可见。
4. **更新数据都是先读后写的，而这个读，只能读当前的值，称为“当前读”（current read）。**

可重复读隔离级别下，事务A,B,C依次创建且TrxA<TrxB<TrxC，并且都访问同一行记录C中的记录，其中事务C执行更新操作并且处于已提交状态，事务A和事务B尚未提交，那么事务A在访问过程中(非当前读)，对于事务B的更新的记录内容不可见，因为事务B尚未提交且TrxA<TrxB，对于事务C的更新也是不可见的，即使C已经提交了，因为TrxA<TrxC即事务A创建的时候事务C尚未提交，所以看不到事务C更新的记录。

快照读(普通select)的情况下，读到的数据是事务刚创建的时候生成的视图，该视图在创建的时候只能看到已提交的记录；

**当前读(select ...for update,update/delete)的情况下，读到的数据是当前所有事务中最新的处于已提交状态的记录。**

普通查询语句是一致性读，一致性读会根据row trx_id和一致性视图确定数据版本的可见性。

- **对于可重复读，查询只承认在事务启动前就已经提交完成的数据；**
- **对于读提交，查询只承认在语句启动前就已经提交完成的数据**；

而当前读，总是读取已经提交完成的最新版本。

**在InnoDB事务中，行锁是在需要的时候才加上的(不是在事务开始的时候而是读取到该行的时候)，但并不是不需要了就立刻释放，而是要等到事务结束时才释放。这个就是两阶段锁协议。**

Mysql死锁处理策略：1.直接进入等待，直到超时，通过参数innodb_lock_wait_timeout来设置，默认50s

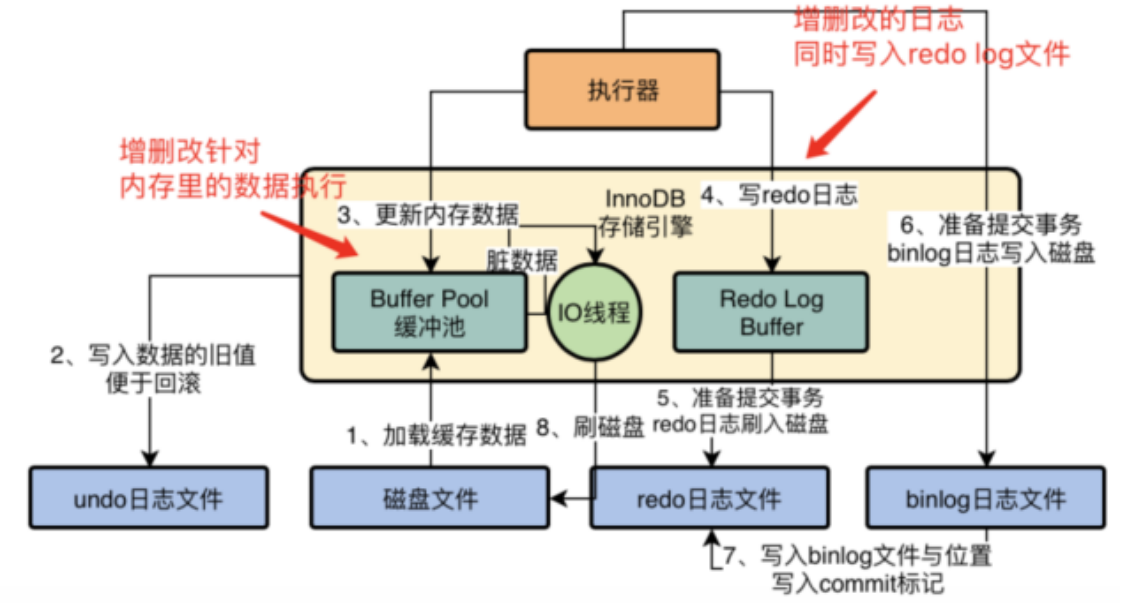
2.发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务，让其他事务得以继续执行。将参数innodb_deadlock_detect设置为on（默认是on），表示开启这个逻辑。

当需要更新一个数据页时，如果数据页在内存中就直接更新，而如果这个数据页还没有在内存中的话，在不影响数据一致性的前提下，InooDB会将这些更新操作缓存在change buffer中，这样就不需要从磁盘中读入这个数据页了。在下次查询需要访问这个数据页的时候，将数据页读入内存，然后执行change buffer中与这个页有关的操作。通过这种方式就能保证这个数据逻辑的正确性。

**将change buffer中的操作应用到原数据页，得到最新结果的过程称为merge。**除了访问这个数据页会触发merge外，系统有后台线程会定期merge。在数据库正常关闭（shutdown）的过程中，也会执行merge操作。

只有普通索引会使用到change buffer，唯一索引不会使用。

Buffer Pool

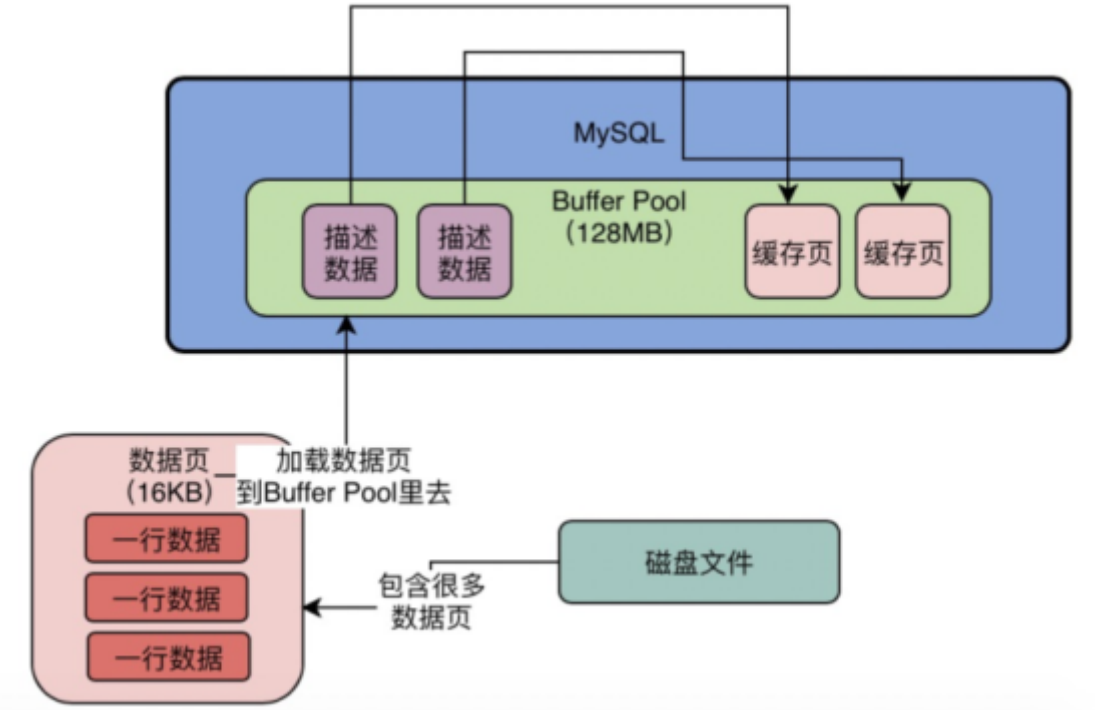



我们在对数据库执行增删改操作的时候，不可能直接更新磁盘上的数据的，因为如果你对磁盘进行随机读写操作，那速度是相当的慢，所以在对数据库执行增删改操作的时候，实际上主要都是针对内存里的Buffer Pool中的数据进行的，也就是实际上主要是对数据库的内存里的数据结构进行了增删改。但是如果Buffer pool中记录的修改操作尚未刷新到磁盘的时候，系统崩溃了怎么保证数据不丢失呢？MySQL就怕这个问题，所以引入了一个redo log机制，你在对内存里的数据进行增删改的时候，他同时会把增删改对应的日志写入redo log中。万一你的数据库突然崩溃了，没关系，只要从redo log日志文件里读取出来你之前做过哪些增删改操作，瞬间就可以重新把这些增删改操作在你的内存里执行一遍，这就可以恢复出来你之前做过哪些增删改操作了。这就是crash safe

## Buffer Pool的一句话总结

Buffer Pool是数据库中我们第一个必须要搞清楚的核心组件，因为增删改操作首先就是针对这个内存中的Buffer Pool里的数据执行的，同时配合了后续的redo log、刷磁盘等机制和操作。

所以Buffer Pool就是数据库的一个内存组件，里面缓存了磁盘上的真实数据，然后我们的系统对数据库执行的增删改操作，其实主要就是对这个内存数据结构中的缓存数据执行的。




缓冲页与磁盘中的数据页一一对应，对于每个缓存页，他实际上都会有一个描述信息，这个描述信息大体可以认为是用来描述这个缓存页的。 比如包含如下的一些东西：这个数据页所属的表空间、数据页的编号、这个缓存页在Buffer Pool中的地址以及别的一些杂七杂八的东西。 每个缓存页都会对应一个描述信息，这个描述信息本身也是一块数据，在Buffer Pool中，每个缓存页的描述数据放在最前面，然后各个缓存页放在后面。

在线上运行时，buffer pool是有多个的，每个buffer pool里多个chunk但是共用一套链表数据结构，然后执 行crud的时候，就会不停的加载磁盘上的数据页到缓存页里来，然后会查询和更新缓存页里的数据，同时维护一系列的链表结构。 然后后台线程定时根据lru链表和flush链表，去把一批缓存页刷入磁盘释放掉这些缓存页，同时更新free链表。 如果执行crud的时候发现缓存页都满了，没法加载自己需要的数据页进缓存，此时就会把lru链表冷数据区域的缓存页刷入磁盘，然后加载自己需要的数据页进来。

redo log buffer先刷新到磁盘中的redo log日志文件，等到磁盘中redo log日志文件写满了或者某个特定时机把redo log日志文件记录的修改内容应用到磁盘的数据页

### 4、磁盘上的一行数据到底如何读取出来的?

结合上面的磁盘上的数据存储格式来思考一下，一行数据到底是如何读取出来的呢？ 再看上面的磁盘数据存储格式：

```
0x09 0x04 00000101 头信息 column1=value1 column2=value2 ... columnN=valueN
```

首先他必然要把变长字段长度列表和NULL值列表读取出来，有哪几个变长字段，哪几个变长字段是NULL，因为NULL值列表里谁是NULL谁不是NULL都一清二楚。

此时就可以从变长字段长度列表中解析出来不为NULL的变长字段的值长度，然后也知道哪几个字段是NULL的，此时根据这些信息，就可以从实际的列值存储区域里，把你每个字段的值读取出来了。

- 如果是变长字段的值，就按照他的值长度来读取，
- 如果是NULL，就知道他是个NULL，没有值存储，
- 如果是定长字段，就按照定长长度来读取，

这样就可以完美的把你一行数据的值都读取出来了！

**总结**

（1）缓冲池(buffer pool)是一种**常见的降低磁盘访问的机制**；

（2）缓冲池通常**以页(page)为单位缓存数据**；

（3）缓冲池的**常见管理算法是LRU**，memcache，OS，InnoDB都使用了这种算法；

（4）InnoDB对普通LRU进行了优化：

- 将缓冲池分为**老生代和新生代**，入缓冲池的页，优先进入老生代，页被访问，才进入新生代，以解决预读失效的问题
- 页被访问，且在老生代**停留时间超过配置阈值**的，才进入新生代，以解决批量数据访问，大量热数据淘汰的问题

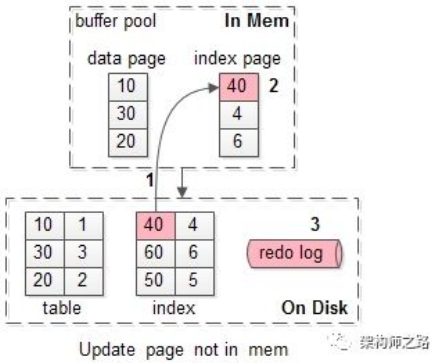


正常情况下，数据库数据都是存在磁盘文件中，以页的形式存储。但是由于磁盘的读取速度比较慢，所以mysql引入了缓冲池的概念，类似于操作系统中的缓存，把磁盘中的页放入缓冲池，这样的话要读取的数据在缓冲池，那么由于缓冲池是基本内存的，所以读取速度会大大加快。同时，对mysql数据修改的时候，也是先修改缓冲池中的页，这些被修改的页称为脏页，在合适的时机，会把脏页刷新到磁盘对应的页。显然，引入缓冲池可以大大加快读写的速度。但是，由于内存中的数据会在进程异常重启时丢失，所以需要把写操作持久化到文件，这样即使异常导致缓冲池中的页数据丢失，也可以通过已持久化的文件进行恢复。这就引入了redo log，每次涉及到对数据更新的操作的时候，会把相关的更改记录顺序写入redo log日志，这样即使异常出现，也可以保证之前的更改操作不丢失，这就是crash safe。当然，为了加快更新的速度，并没有直接将修改操作写入redo log日志文件，也是先写入redo log buffer，在合适的时机再刷回redo log日志。

为什么引入写缓存(change buffer)?

**情况二**

假如要修改页号为40的索引页，而这个页正好**不**在缓冲池内。




Update page not in mem



此时麻烦一点，如上图需要1-3：

（1）先把需要为40的索引页，从磁盘加载到缓冲池，一次磁盘随机读操作；

（2）修改缓冲池中的页，一次内存操作；

（3）写入redo log，一次磁盘顺序写操作；

没有命中缓冲池的时候，**至少产生一次磁盘IO**，对于写多读少的业务场景，**是否还有优化的空间呢?**

**什么是InnoDB的写缓冲?**

在MySQL5.5之前，叫插入缓冲(insert buffer)，只针对insert做了优化；现在对delete和update也有效，叫做写缓冲(change buffer)。

它是一种应用在**非唯一普通索引页**(non-unique secondary index page)不在缓冲池中，对页进行了写操作，并不会立刻将磁盘页加载到缓冲池，而仅仅记录缓冲变更(buffer changes)，等未来数据被读取时，再将数据合并(merge)恢复到缓冲池中的技术。写缓冲的**目的**是降低写操作的磁盘IO，提升数据库性能。

*画外音：R了狗了，这个句子，好长。*