链滴

一条更新 SQL 是如何执行的?

作者：xhaoxiong

原文链接：https://ld246.com/article/1635734840295

来源网站：链滴

<h3 id="前言">前言</h3>
比如说这个表有一个主键 ID 和一个整型字段 c：
<pre><code class="highlight-chroma">mysql> create table T(ID int primary key, c int);
</code></pre>
如果要将 ID=2 这一行的值加 1，SQL 语句就会这么写：
<pre><code class="highlight-chroma">mysql> update T set c=c+1 where ID=2;
</code></pre>
首先，可以确定的说，查询语句的那一套流程，更新语句也是同样会走一遍。

你执行语句前要先连接数据库，这是连接器的工作。
在一个表上有更新的时候，跟这个表有关的查询缓存会失效，所以这条语句就会把表 T 上所有缓结果都清空。这也就是我们一般不建议使用查询缓存的原因。
接下来，分析器会通过词法和语法解析知道这是一条更新语句。优化器决定要使用 ID 这个索引然后，执行器负责具体执行，找到这一行，然后更新。
与查询流程不一样的是，更新流程还涉及两个重要的日志模块，它们正是我们今天要讨论的主角redo log（重做日志）和 binlog（归档日志）。
<h3 id="重要的日志模块-redo-log">重要的日志模块：redo log</h3>
例：酒店掌柜有一个粉板，专门用来记录客人的赊账记录。如果赊账的人不多，那么他可以把顾名和账目写在板上。但如果赊账的人多了，粉板总会有记不下的时候，这个时候掌柜一定还有一个专记录赊账的账本。
如果有人要赊账或者还账的话，掌柜一般有两种做法：

一种做法是直接把账本翻出来，把这次赊的账加上去或者扣除掉；
另一种做法是先在粉板上记下这次的账，等打烊以后再把账本翻出来核算。

在生意红火柜台很忙时，掌柜一定会选择后者，因为前者操作实在是太麻烦了。首先，你得找到个人的赊账总额那条记录。你想想，密密麻麻几十页，掌柜要找到那个名字，可能还得带上老花镜慢找，找到之后再拿出算盘计算，最后再将结果写回到账本上。
这整个过程想想都麻烦。相比之下，还是先在粉板上记一下方便。你想想，如果掌柜没有粉板的助，每次记账都得翻账本，效率是不是低得让人难以忍受？
同样，在 MySQL 里也有这个问题，如果每一次的更新操作都需要写进磁盘，然后磁盘也要找到应的那条记录，然后再更新，整个过程 IO 成本、查找成本都很高。为了解决这个问题，MySQL 的计者就用了类似酒店掌柜粉板的思路来提升更新效率。
而粉板和账本配合的整个过程，其实就是 MySQL 里经常说到的 WAL 技术，WAL 的全称是 Wrie-Ahead Logging，它的关键点就是先写日志，再写磁盘，也就是先写粉板，等不忙的时候再写账本
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，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，将这个操作记录更到磁盘里面，而这个更新往往是在系统比较空闲的时候做，这就像打烊以后掌柜做的事。
如果今天赊账的不多，掌柜可以等打烊后再整理。但如果某天赊账的特别多，粉板写满了，又怎办呢？这个时候掌柜只好放下手中的活儿，把粉板中的一部分赊账记录更新到账本中，然后把这些记从粉板上擦掉，为记新账腾出空间。
与此类似，InnoDB 的 redo log 是固定大小的，比如可以配置为一组 4 个文件，每个文件的大是 1GB，那么这块“粉板”总共就可以记录 4GB 的操作。从头开始写，写到末尾就又回到开头循环，如下面这个图所示。

write pos 是当前记录的位置，一边写一边后移，写到第 3 号文件末尾后就回到 0 号文件开头。ceckpoint 是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到数据文件。</

>
write pos 和 checkpoint 之间的是“粉板”上还空着的部分，可以用来记录新的操作。如果 writ pos 追上 checkpoint，表示“粉板”满了，这时候不能再执行新的更新，得停下来先擦掉一些记录把 checkpoint 推进一下。
有了 redo log，InnoDB 就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这能力称为 crash-safe。
要理解 crash-safe 这个概念，可以想想我们前面赊账记录的例子。只要赊账记录记在了粉板上写在了账本上，之后即使掌柜忘记了，比如突然停业几天，恢复生意后依然可以通过账本和粉板上的据明确赊账账目。
<h3 id="重要的日志模块-binlog">重要的日志模块：binlog</h3>
MySQL 整体来看，其实就有两块：一块是 Server 层，它主要做的是 MySQL 功能层面的事情还有一块是引擎层，负责存储相关的具体事宜。 redo log 是 InnoDB 引擎特有的日志，而 Server 也有自己的日志，称为 binlog（归档日志）。
为什么会有两份日志？
因为最开始 MySQL 里并没有 InnoDB 引擎。MySQL 自带的引擎是 MyISAM，但是 MyISAM 有 crash-safe 的能力，binlog 日志只能用于归档。而 InnoDB 是另一个公司以插件形式引入 MySQL 的，既然只依靠 binlog 是没有 crash-safe 能力的，所以 InnoDB 使用另外一套日志系统——也就是 edo log 来实现 crash-safe 能力。
这两种日志有以下三点不同：
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，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。
有了对这两个日志的概念性理解，我们再来看执行器和 InnoDB 引擎在执行这个简单的 update 句时的内部流程。
1、执行器先找引擎取 ID=2 这一行。ID 是主键，引擎直接用树搜索找到这一行。如果 ID=2 这行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回
2、执行器拿到引擎给的行数据，把这个值加上 1，比如原来是 N，现在就是 N+1，得到新的一数据，再调用引擎接口写入这行新数据。
3、引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时 redo log 处于 prepare 状态。然后告知执行器执行完成了，随时可以提交事务。
4、执行器生成这个操作的 binlog，并把 binlog 写入磁盘。
5、执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 改成提交（commit）状态，新完成。

最后三步将 redo log 的写入拆成了两个步骤：prepare 和 commit，这就是"两阶段提交"。</p

<h3 id="两阶段提交">两阶段提交</h3>
1、先写 redo log 后写 binlog。假设在 redo log 写完，binlog 还没有写完的时候，MySQL 程异常重启。由于我们前面说过的，redo log 写完之后，系统即使崩溃，仍然能够把数据恢复回来，以恢复后这一行 c 的值是 1。但是由于 binlog 没写完就 crash 了，这时候 binlog 里面就没有记录个语句。因此，之后备份日志的时候，存起来的 binlog 里面就没有这条语句。然后你会发现，如果要用这个 binlog 来恢复临时库的话，由于这个语句的 binlog 丢失，这个临时库就会少了这一次更新恢复出来的这一行 c 的值就是 0，与原库的值不同。
2、先写 binlog 后写 redo log。如果在 binlog 写完之后 crash，由于 redo log 还没写，崩溃复以后这个事务无效，所以这一行 c 的值是 0。但是 binlog 里面已经记录了“把 c 从 0 改成 1”这日志。所以，在之后用 binlog 来恢复的时候就多了一个事务出来，恢复出来的这一行 c 的值就是 1 与原库的值不同。
可以看到，如果不使用“两阶段提交”，那么数据库的状态就有可能和用它的日志恢复出来的库

状态不一致。
简单说，redo log 和 binlog 都可以用于表示事务的提交状态，而两阶段提交就是让这两个状态持逻辑上的一致。
redo log 用于保证 crash-safe 能力。innodb_flush_log_at_trx_commit 这个参数设置成 1 的候，表示每次事务的 redo log 都直接持久化到磁盘。这个参数我建议你设置成 1，这样可以保证 MyQL 异常重启之后数据不丢失。
sync_binlog 这个参数设置成 1 的时候，表示每次事务的 binlog 都持久化到磁盘。这个参数我建议你设置成 1，这样可以保证 MySQL 异常重启之后 binlog 不丢失。