数据库原理

相关问题

一条SQL的执行过程?

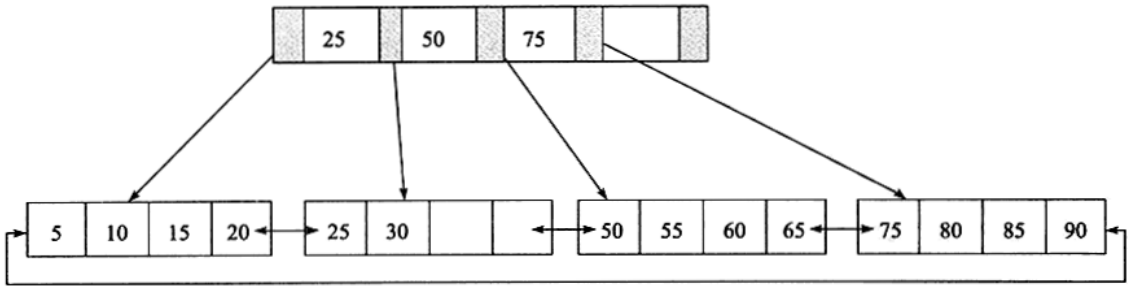
索引

索引优点

- 大大减少服务器需要扫描的数据量
- 可以帮助服务器避免排序和临时表
- 将随机IO变为顺序IO

B+树

定义



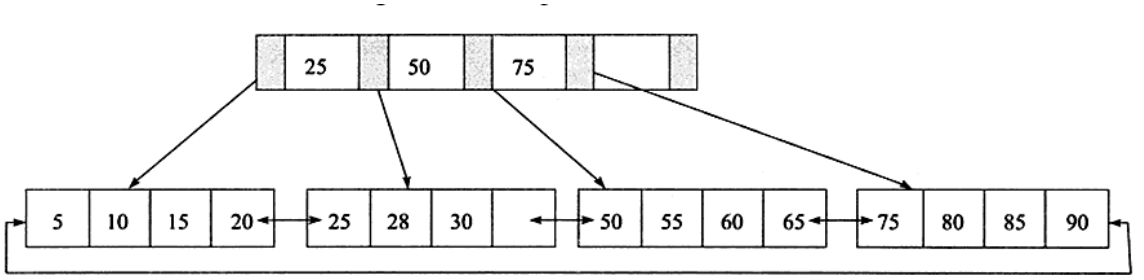
插入操作

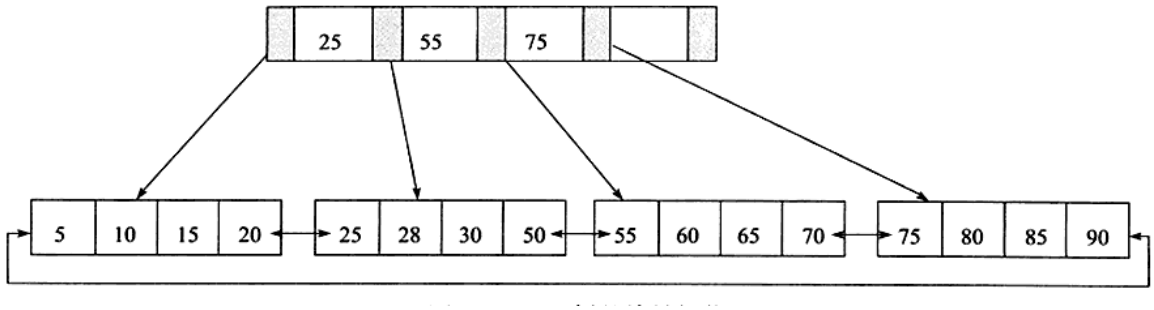
Leaf Page 满	Index Page 满	操作
No	No	直接将记录插入到叶子节点
Yes	No	1）拆分 Leaf Page 2）将中间的节点放入到 Index Page 中 3）小于中间节点的记录放左边 4）大于或等于中间节点的记录放右边
Yes	Yes	1）拆分 Leaf Page 2）小于中间节点的记录放左边 3）大于或等于中间节点的记录放右边 4）拆分 Index Page 5）小于中间节点的记录放左边 6）大于中间节点的记录放右边 7）中间节点放入上一层 Index Page

旋转操作

旋转发生在Leaf Page已经满，但是其左右兄弟节点没有满得情况下。这时，B+树不会急于去做拆分页得操作，而是将记录移到所在页得兄弟节点上。

如下图所示，插入键值70，此时B+树不会急于去做拆分页的操作，而是将记录移到所在页的兄弟节点上。旋转操作使B+树减少了一次页的拆分操作，同时树的高度保持不变。通常情况下，**左兄弟**会被首先检查用来做旋转操作。





删除操作

B+树使用填充因子（fill factor)来控制树的删除变化，50%是填充因子可设的最小值。

叶子节点小于填充因子	中间节点小于填充因子	操作
No	No	直接将记录从叶子节点删除，如果该节点还是 Index Page 的节点，用该节点的右节点代替
Yes	No	合并叶子节点和它的兄弟节点，同时更新 Index Page
Yes	Yes	1）合并叶子节点和它的兄弟节点 2）更新 Index Page 3）合并 Index Page 和它的兄弟节点

B+树和B树的比较

1. B+ 树的磁盘 IO 更低

B+ 树的内部节点并没有指向关键字具体信息的指针。因此其内部节点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

2. B+ 树的查询效率更加稳定

由于非叶子结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

3. B+ 树元素遍历效率高

B 树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而 B 树不支持这样的操作（或者说效率太低）。

B+树和红黑树的比较

1. 磁盘I/O次数

红黑树的出度为 2，树高较高，B+树出度一般都大，层数为1-3，所以查找次数少

2. 利用磁盘预读特性和局部性原理

为了减少磁盘 I/O 操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道。每次会读取页的整数倍。

操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点。

各种索引

唯一索引

字段值必须唯一，但是可以为空

全文索引

只能在文本类型CHAR,VARCHAR,TEXT类型字段上创建全文索引。字段长度比较大时，如果创建普通索引，在进行like模糊查询时效率比较低，这时可以创建全文索引。MyISAM和InnoDB中都可以使用全文索引。

哈希索引

哈希索引能以 O(1) 时间进行查找，但是失去了有序性：

- 无法用于排序与分组；
- 只支持精确查找，无法用于部分查找和范围查找。

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

聚簇索引

叶子节点存储完整数据。

聚簇索引并不是一种单独的索引类型，而是一种数据的存储方式。具体细节依赖于其实现方式。Innodb中，索引和数据存放在同一个idb文件。InnoDB通过主键聚集数据。

因为无法把数据同时存放在两个地方，所以**一个表只能有一个聚簇索引。**如果没有定义主键，InnoDB会选择一个**唯一**的非空索引代替。 如果没有这样的索引， InnoDB会隐式定义一个主键来作为聚簇索引。

非聚簇索引

叶子节点不包含行记录的全部数据。

对于非聚簇索引的查询需要回到聚集索引得到整行数据，这个过程称之为回表，

联合索引

对多个字段同时建立的索引。

组合索引的使用，需要遵循**最左前缀匹配原则（最左匹配原则）**。一般情况下在条件允许的情况下使用组合索引替代多个单列索引使用。

索引覆盖

需要查询的值已经在索引树上，可以直接提供查询结果，不需要回表，称为覆盖索引。

覆盖索引可以减少树的搜索次数，显著提升查询功能，所以使用覆盖索引是一个常用性能优化手段。

使用覆盖索引的一个好处是辅助索引不包含整行记录的所有信息，所以其大小远小于聚集索引，因此可以减少大量的IO操作。

实现方法：将被查询字段建立到联合索引去。

前缀索引

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。

前缀长度的选取需要根据索引选择性来确定。

使用前缀索引，定义好长度，就可以做到既节省空间，又不用额外增加太多的查询成本。

怎么确定应该使用多长的前缀？我们在建立索引时关注的是区分度，区分度越高越好。因为区分度越高，意味着重复的键值越少。

使用下面语句，算出这个列上有多少个不同的值

```
mysql> select count(distinct email) as L from SUser;
```

然后，依次选取不同长度的前缀来看这个值，比如我们要看一下 4~7 个字节的前缀索引，可以用这个语句:

```
select   count(distinct left(email,4)) as L4,
        count(distinct left(email,5)) as L5,
        count(distinct left(email,6)) as L6,
        count(distinct left(email,7)) as L7,
from SUser;
```

当然，使用前缀索引很可能会损失区分度，所以你需要预先设定一个可以接受的损失比例，比如 5%。然后，在返回的 L4~L7 中，找出不小于 L * 95% 的值，假设这里 L6、L7 都满足，你就可以选择前缀长度为 6。

前缀索引缺点:

- 可能会增加扫描行数，影响性能
- 使用前缀索引用不上覆盖索引对查询的优化，因为一定会回表确认完整信息。

适合索引的场景

1.**主键**

主键一般为id等具有唯一性标识的字段，需要频繁查找、连接。InnoDB中会自动为主键建立聚集索引，即使没有定义主键，也会自动生成一个隐藏主键建立索引；MyISAM中不会自动生成主键。建议给每张表指定主键。

2.**频繁作为查询条件的字段**

索引是以空间换时间的，某字段如果频繁作为查询条件，建议建立索引

使用前缀索引后，可能会导致查询语句读数据的次数变多。

不适合索引的场景

1. **数据重复且分布平均的字段（比如性别、年龄）**
2. **很少作为查询条件的字段**
3. **频繁更新的字段、表**

如果字段添加了索引，在更新时不仅要更新数据本身，还要维护其索引，如果频繁更新会带来很多额外开销。再者，如果一个表频繁进行增删改操作，也不适合索引

4. **数据量大的字段（比如长字符串）**
5. **表的记录不多**

一般数据量达到300万-500万时考虑建立索引。

索引失效

不符合最左前缀原则。联合索引a,b,c，b+ 树是按照从左到右的顺序比较的，a，b，c

1. where条件有or
2. 复合索引未用左列字段;
3. like以%开头;
4. .需要类型转换; （varchar类型传了数字）
5. where里索引列使用了函数;

如何为表加索引?

1.添加PRIMARY KEY（主键索引）

```
ALTER TABLE `table_name` ADD PRIMARY KEY ( `column` )
```

2.添加UNIQUE(唯一索引)

```
ALTER TABLE `table_name` ADD UNIQUE ( `column` )
```

3.添加INDEX(普通索引)

```
ALTER TABLE `table_name` ADD INDEX index_name ( `column` )
```

4.添加FULLTEXT(全文索引)

```
ALTER TABLE `table_name` ADD FULLTEXT ( `column`)
```

5.添加多列索引

```
ALTER TABLE `table_name` ADD INDEX index_name ( `column1`, `column2`, `column3`
)
```

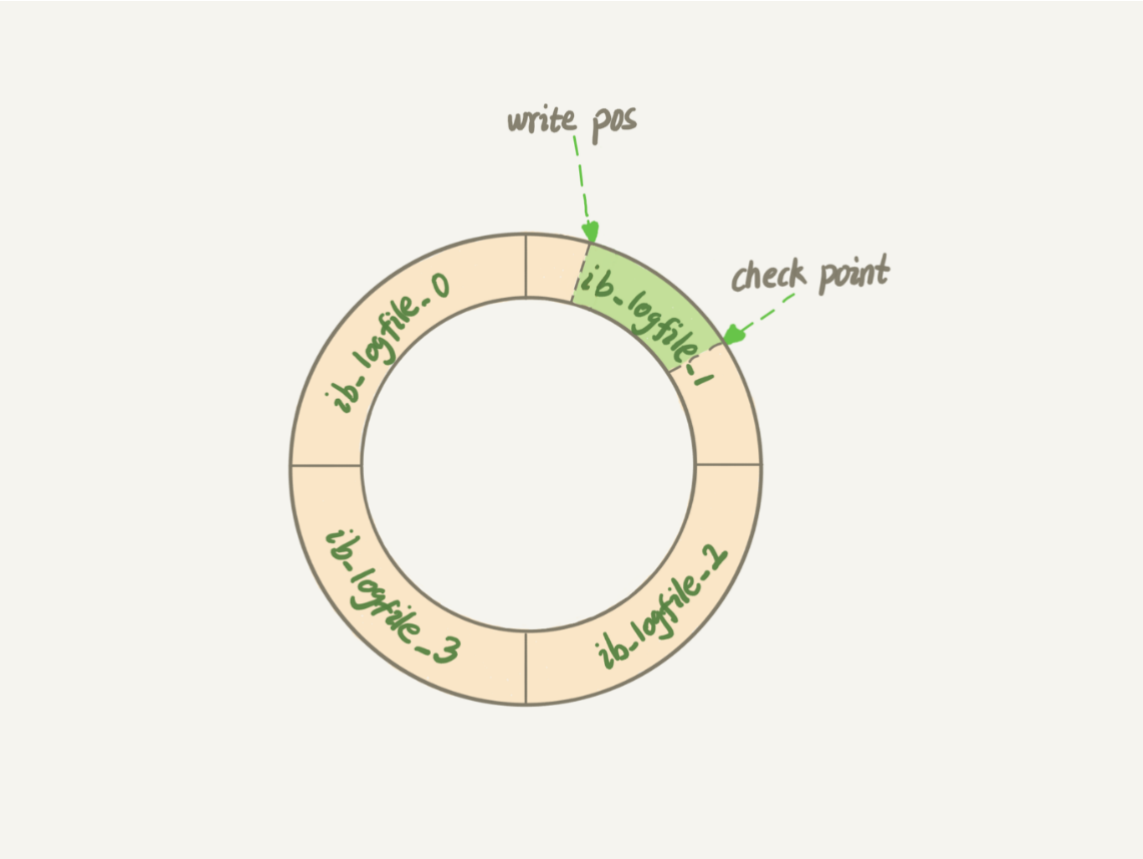
日志

redo log

是物理格式的日志，记录数据库每个页的修改信息。插入数据库会先把更新写到redolog，等空闲时再写入磁盘。

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。

InnoDB 的 redo log 是固定大小的，比如可以配置为一组 4 个文件，每个文件的大小是 1GB，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写，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。checkpoint 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 write 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，这个能力称为`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或写在了账本上，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数据明确赊账账目。

undo log

undo是**逻辑日志**！！！只是将数据**逻辑**的恢复到原来的样子。

undo的作用，支持回滚和MVCC。

binlog

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。上面提到的 redo log 是 InnoDB 引擎特有的日志，而 Server 层也有自己的日志，称为 binlog（归档日志）。

为什么会有两份日志呢?

因为最开始 MySQL 里并没有 InnoDB 引擎。MySQL 自带的引擎是 MyISAM，但是 MyISAM 没有 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 redo log 来实现 crash-safe 能力。

binlog是语句格式的日志，事务提交的时候，一次性将事务中的所有sql语句和反向信息记录到binlog中。

比如delete操作的话，就对应着delete本身和其反向的insert；update操作的话，就对应着update执行前后的版本的信息；insert操作则对应着delete和insert本身的信息。

用途：

1. 用于复制，在主从复制中，从库利用主库上的binlog进行重播，实现主从同步。
2. 用于数据库的基于时间点的还原。

redo log 和 binlog 区别

redo log是InnoDB引擎生成的。binlog是在mysql的服务层产生的，MySQL数据库中的任何存储引擎对于数据库的更改都会产生二进制日志。

redo log是物理日志，其记录的是对于每个页的修改，而binlog是一种逻辑日志，其记录的是对应的SQL语句。

redo log 是循环写，写到末尾是要回到开头继续写的。这样历史日志没法保留，redo log 也就起不到归档的作用。

此外，两种日志记录写入磁盘的时间点不同。binlog只在事务提交完成后进行一次写入。而Innodb存储引擎的redo log在事务进行中不断的被写入，这表现为日志并不是随事务提交的顺序进行写入的。

redo log、binlog两阶段提交

将redo log的写入拆成了两个步骤：prepare和commit，即两阶段提交。

为什么要两阶段提交?

由于 redo log 和 binlog 是两个独立的逻辑，如果不用两阶段提交，要么就是先写完 redo log 再写 binlog，或者采用反过来的顺序。我们看看这两种方式会有什么问题。

仍然用前面的 update 语句来做例子。假设当前 ID=2 的行，字段 c 的值是 0，再假设执行 update 语句过程中在写完第一个日志后，第二个日志还没有写完期间发生了 crash，会出现什么情况呢?

先写 redo log 后写 binlog

假设在 redo log 写完，binlog 还没有写完的时候，MySQL 进程异常重启。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，所以恢复后这一行 c 的值是 1。但是由于 binlog 没写完就 crash 了，这时候 binlog 里面就没有记录这个语句。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，如果需要用这个 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。

先写 binlog 后写 redo log

如果在 binlog 写完之后 crash，由于 redo log 还没写，崩溃恢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”这个日志。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的状态不一致。

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的状态不一致。

不只是误操作后需要用这个过程来恢复数据。当你需要扩容的时候，也就是需要再多搭建一些备库来增加系统的读能力的时候，现在常见的做法也是用全量备份加上应用 binlog 来实现的，这个“不一致”就会导致你的线上出现主从数据库不一致的情况。

事务

ACID特性

A（Atomic) 原子性

事务是不可分割的工作单位。一个事务要么成功，要么不成功。事务中任何一个SQL语句如果执行失败，已经成功执行的SQL语句必须撤销，数据库状态应该退回到执行事务前的状态。

C (consistency) 一致性

一致性指事务从一种状态变为下一种一致性状态。在事务开始前和事务结束后，数据库的完整性约束没有遭到破坏。

例如，在一个表中有一个字段为姓名，为唯一约束，即在表中姓名不能重复。如果某个事务对姓名字段进行了修改，但是在事务提交或事务操作发生回滚后，表中的姓名变得非唯一了，即数据库从一种状态变为了一种不一致的状态。

I (isolation) 隔离性

一个事务所做的修改在最终提交前，对其他事务是不可见的。

D (Durability)

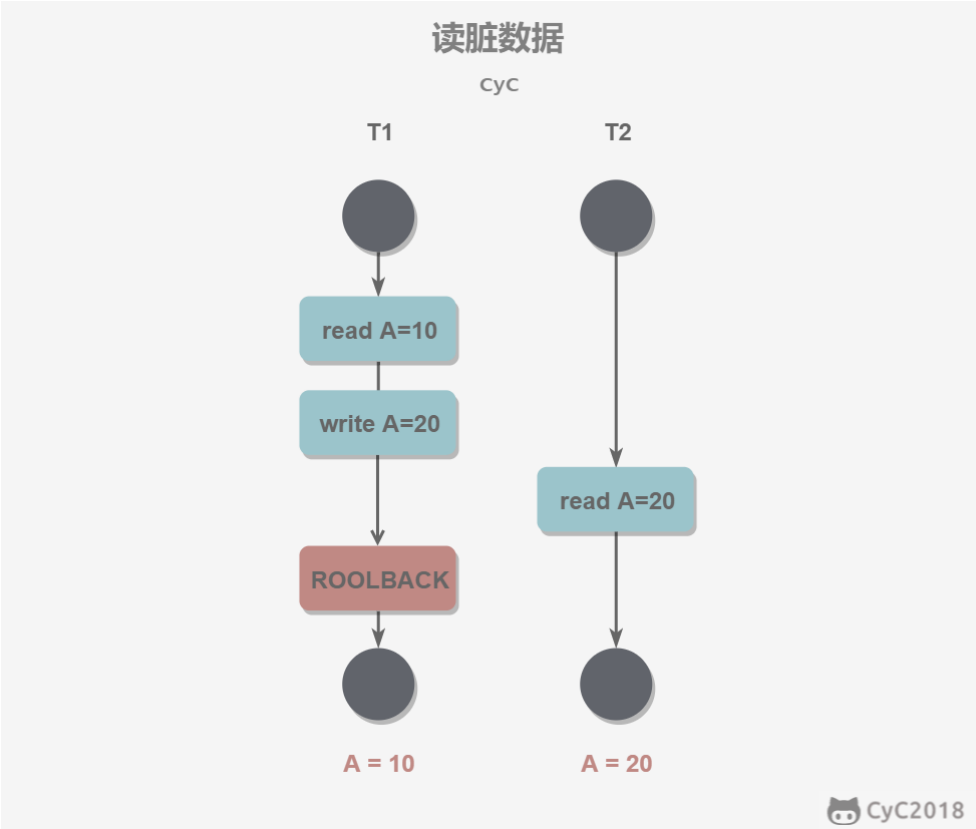
一旦事务提交，则其所做的修改将会永远保存到数据库中，即使系统发生崩溃，事务执行的结果也不能丢失。

事务并发可能出现的情况

脏读

一个事务读到了另一个未提交事务修改过的数据

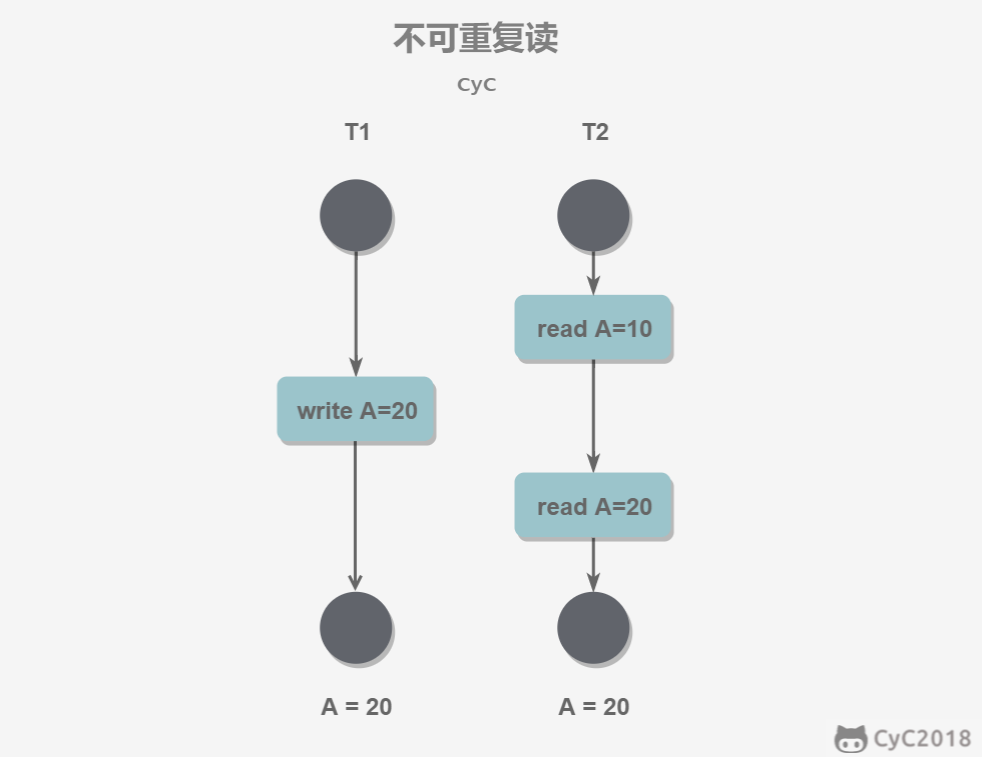
例如：T1 修改一个数据但未提交，T2 随后读取这个数据。如果 T1 撤销了这次修改，那么 T2 读取的数据是脏数据。



不可重复读

一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值。（不可重复读在读未提交和读已提交隔离级别都可能会出现）

例如：T2 读取一个数据，T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。

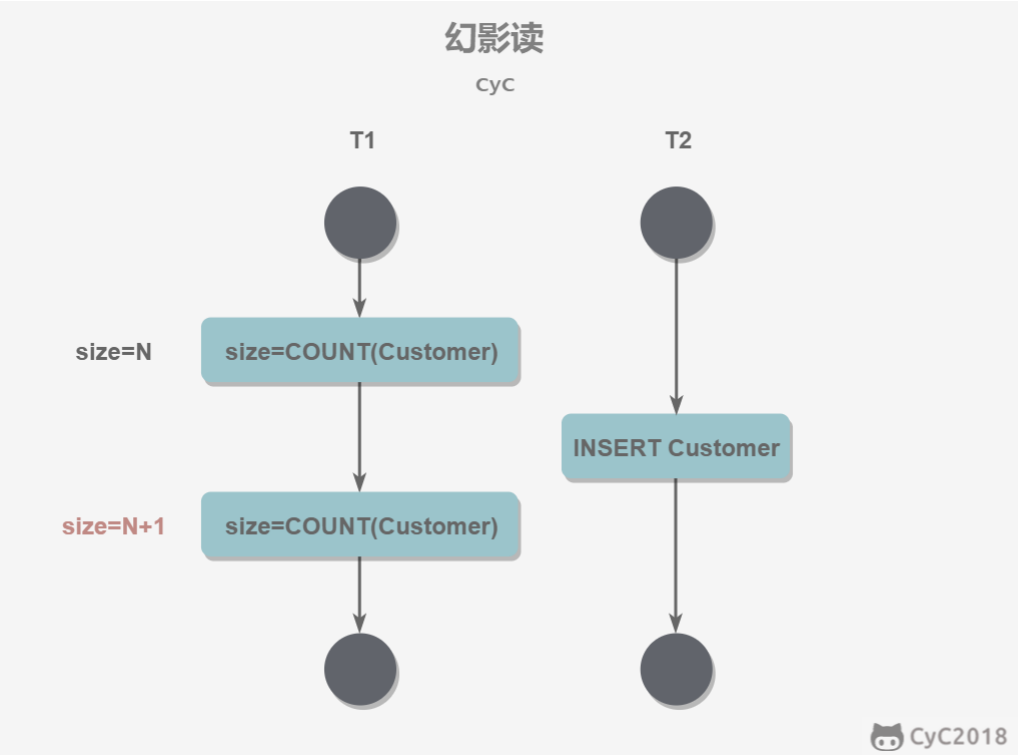


幻读

一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来。（幻读在读未提交、读已提交、可重复读隔离级别都可能会出现）

在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的。因此，幻读在“当前读”下才会出现。幻读仅专指“新插入的行”。MySQL使用行锁和间隙锁配合防止幻读现象。

幻读本质上也属于不可重复读的情况，T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。



产生并发不一致性问题的主要原因是破坏了事务的隔离性，解决方法是通过并发控制来保证隔离性。并发控制可以通过封锁来实现，但是封锁操作需要用户自己控制，相当复杂。数据库管理系统提供了事务的隔离级别，让用户以一种更轻松的方式处理并发一致性问题。

事务的四种隔离级别

四种隔离级别：

- 读未提交
- 读提交
- 可重复读
- 串行化

隔离级别比较：可串行化>可重复读>读已提交>读未提交

隔离级别对性能的影响比较：可串行化>可重复读>读已提交>读未提交

由此看出，隔离级别越高，所需要消耗的MySQL性能越大（如事务并发严重性），为了平衡二者，一般建议设置的隔离级别为可重复读，MySQL默认的隔离级别也是可重复读。

1. 读未提交

该隔离级别的事务会读到其它未提交事务的数据，此现象也称之为**脏读**。

2. 读提交（不可重复读）

一个事务可以读取另一个已提交的事务，多次读取会造成不一样的结果，此现象称为**不可重复读问题**，Oracle 和 SQL Server 的默认隔离级别。

3. 可重复读

该隔离级别是 MySQL 默认的隔离级别，在同一个事务里， `select` 的结果是事务开始时时间点的状态，因此，同样的 `select` 操作读到的结果会是一致的，但是，会有 **幻读** 现象。

InnoDB 引擎的默认级别为可重复读，可以通过间隙锁（ `next-key locking` ）机制来避免幻读。间隙锁使得InnoDB不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定，以防止幻影行的插入

4. 串行化

对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

两个事务A和B, 读读操作不会阻塞，读写，写读，写写会阻塞。

并发一致性对比

	脏读	不可重复读	幻读
读未提交	✕	✕	✕
读提交	✔	✕	✕
可重复读	✔	✔	✕
串行化	✔	✔	✔

需要注意的是，MySQL默认隔离级别为**可重复读**，采用Next-Key算法，能够避免幻读。

锁

共享锁和排他锁

- 共享锁（**S**hare Locks，记为S锁），读取数据时加S锁
- 排他锁（e**X**clusive Locks，记为X锁），修改数据时加X锁

表 6-3　排他锁和共享锁的兼容性

	X	S
X	不兼容	不兼容
S	不兼容	兼容

用法：

- 共享锁之间不互斥，简记为：**读读可以并行**
- 排他锁与任何锁互斥，简记为：**写读，写写不可以并行**

共享/排它锁的潜在问题是，不能充分的并行，解决思路是**数据多版本**

意向锁（Intention Lock)

InnoDB支持多粒度锁(multiple granularity locking)，它允许行级锁与表级锁共存，实际应用中，InnoDB使用的是意向锁。

意向锁有这样一些特点：

(1) 首先，**意向锁，是一个表级别的锁**(table-level locking)

(2) 意向锁分为

- 意向共享锁(IS Lock)，事务想要获得一张表中某几行的共享锁
- 意向排他锁(IX Lock)，事务想要获得一张表中某几行的排他锁

(3) 意向锁协议(intention locking protocol)并不复杂：

- 事务要获得某些行的S锁，必须先获得表的IS锁
- 事务要获得某些行的X锁，必须先获得表的IX锁

(4) 由于意向锁仅仅表明意向，它其实是比较弱的锁，**意向锁之间并不相互互斥**，而是可以并行

	IS	**IX**
IS	兼容	兼容
IX	兼容	兼容

(5) 它会与共享锁/排它锁互斥，其**兼容互斥表**如下：

	S	**X**
IS	兼容	互斥
IX	互斥	互斥

表锁

记录锁、行锁（Record Lock)

在 InnoDB 事务中，行锁是在需要的时候才加上的，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。如果你的事务中需要锁多个行，要把最可能造成锁冲突、最可能影响并发度的锁尽量往后放，这样能够减少锁持有的时间。

InnoDB 行锁是通过给索引上的索引项加锁来实现的。通过索引条件检索数据，InnoDB 才使用行级锁，否则，InnoDB 将使用表锁！（可以通过 explain 检查 SQL 的执行计划）

varchar类型没加引号，会导致索引失效，行锁变表锁

间隙锁（Gap Lock)

锁住一个范围，但是不包含记录本身

临键锁（Next-key Lock)

Next-key Lock是结合了Gap Lock和Record Lock的一种锁定算法。在Next-key Lock算法下，InnoDB对于行的查询都是采用这种锁定算法。临键锁的范围是左开右闭。

自增锁（Auto-inc Locks）

自增锁是一种特殊的表级别锁（table-level lock），专门针对事务插入`AUTO_INCREMENT`类型的列。最简单的情况，如果一个事务正在往表中插入记录，所有其他事务的插入必须等待，以便第一个事务插入的行，是连续的主键值。

如果没有自增锁，默认RR下，举例假设有数据表：

```
t(id AUTO_INCREMENT, name);
```

数据表中有数据：

1, shenjian

2, zhangsan

3, lisi

事务A	事务B
begin;	
insert into t(name) values(xxx);	begin;
	insert into t(name) values(ooo);
insert into t(name) values(xxoo);	
select * from t where id>3;	

假设B不会阻塞，A会看到`4,xxx`和`6，xxoo`不连续的两条记录，明明是自增的这就很魔幻了。

悲观锁与乐观锁

死锁解决

1. 进入等待，超时释放资源

这个超时时间可以通过参数`innodb_lock_wait_timeout`来设置。

- 不能太大。在 InnoDB 中，`innodb_lock_wait_timeout`的默认值是 50s，意味着如果采用第一个策略，当出现死锁以后，第一个被锁住的线程要过 50s 才会超时退出，然后其他线程才有可能继续执行。对于在线服务来说，这个等待时间往往是无法接受的。
- 不能太小。比如 1s。这样当出现死锁的时候，确实很快就可以解开，但如果不是死锁，而是简单的锁等待呢？所以，超时时间设置太短的话，会出现很多误伤。

所以，正常情况下我们还是要采用第二种策略，死锁检测。

2. 死锁检测

InnoDB采用wait-for graph（等待图）方式来进行死锁检测。wait-for graph要求数据库保存以下两种信息：

- 锁的信息链表
- 事务等待链表

通过上述链表可以构造出一张图，如果这个图中存在回路，就代表存在死锁，资源间相互发生等待。

死锁检测的缺点，想象有 1000 个并发线程要同时更新同一行，那么死锁检测操作就是 100 万这个量级的。最终检测的结果是没有死锁，却耗费了大量的CPU资源。解决方法：

- 如果能确保业务一定不会死锁，临时关闭死锁检测。有风险！ 大量超时！
- 控制并发度。在服务端控制并发度。在客户端控制并发度没啥用，比如600个客户端每个客户端有2个请求，到达服务端后就有1200个请求，还是大。可以通过中间件实现，也可以有高手能够修改MySQL源码。
- 拆分记录。将一行改成逻辑上的多行来减少锁冲突。比如一个电影账户，拆成10个记录的值的和，每次选择随机一条记录来加。这样冲突概率变成原来的1/10

MVCC及实现（多版本并发控制）

参考博客
InnoDB的MVCC实现，是通过保存数据在某个时间点的快照来实现的。一个事务，不管其执行多长时间，其内部看到的数据是一致的。

InnoDB的MVCC实现，是通过保存数据在某个时间点的快照来实现的。一个事务，不管其执行多长时间，其内部看到的数据是一致的。

InnoDB 里面每个事务有一个唯一的事务 ID，叫作`transaction id`。它是在事务开始的时候向 InnoDB 的事务系统申请的，是按申请顺序严格递增的。

而每行数据也都是有多个版本的。每次事务更新数据的时候，都会生成一个新的数据版本，并且把 transaction id 赋值给这个数据版本的事务 ID，记为`row trx_id`。同时，旧的数据版本要保留，并且在新的数据版本中，能够有信息可以直接拿到它。

也就是说，数据表中的一行记录，其实可能有多个版本 (row)，每个版本有自己的`row trx_id`。

快照读

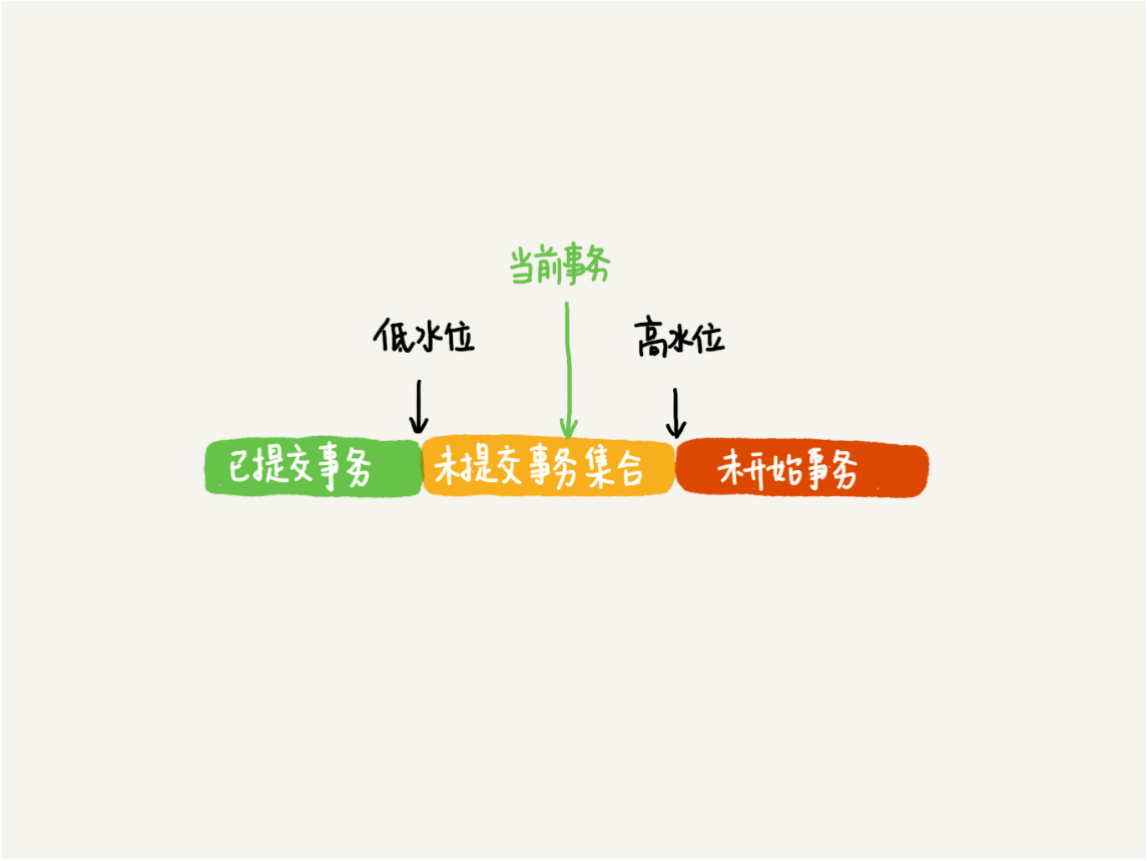
- RR下，事务在第一个Read操作时，会建立Read View
- RC下，事务在每次Read操作时，都会建立Read View

如果数据版本是启动之前生成的，就认；如果是启动之后生成的就不认。如果这个数据时事务自己更新的，它自己还是要认的。

在实现上，InnoDB 为每个事务构造了一个数组，用来保存这个事务启动瞬间，当前正在“活跃”的所有事务 ID。“活跃”指的就是，启动了但还没提交。数组里面事务 ID 的最小值记为低水位，当前系统里面已经创建过的事务 ID 的最大值加 1 记为高水位。这个数组里的数据版本，当前事务不认。

这个视图数组和高水位，就组成了当前事务的一致性视图（read-view）。

而数据版本的可见性规则，就是基于数据的 row trx_id 和这个一致性视图的对比结果得到的。这个视图数组把所有的 row trx_id 分成了几种不同的情况。



这样，对于当前事务的启动瞬间来说，一个数据版本的 row trx_id，有以下几种可能：

1. 如果落在绿色部分，表示这个版本是已提交的事务或者是当前事务自己生成的，这个数据是可见的；
2. 如果落在红色部分，表示这个版本是由将来启动的事务生成的，是肯定不可见的；
3. 如果落在黄色部分，那就包括两种情况
    - 若 row trx_id 在数组中，表示这个版本是由还没提交的事务生成的，不可见；
    - 若 row trx_id 不在数组中，表示这个版本是已经提交了的事务生成的，可见。

更新逻辑

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），当前读能够看到当前事务之后提交事务修改的数据。如果当前的记录的行锁被其他事务占用的话，就需要进入锁等待。

引擎

InnoDB引擎

MySQL默认的事务型引擎，也是最重要和使用最广泛的存储引擎。它被设计成为大量的短期事务，短期事务大部分情况下是正常提交的，很少被回滚。InnoDB的性能与自动崩溃恢复的特性，使得它在非事务存储需求中也很流行。除非有非常特别的原因需要使用其他的存储引擎，否则应该优先考虑InnoDB引擎。

聚簇索引，有且仅有一个主键。

普通索引，多个，叶子节点存储主键值。

普通索引查询过程(回表查询，先定位主键，再定位行记录）：

（1）先遍历普通索引B+树获得主键值；

（2）然后遍历聚簇索引获得行记录对应的值。

优点：避免直接读取磁盘

实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+ Next-Key Locking 防止幻影读。。

内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其它存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

MyISAM引擎

在MySQL 5.1 及之前的版本，MyISAM是默认引擎。

MyISAM提供的大量的特性，包括全文索引、压缩、空间函数（GIS）等，但MyISAM并不支持事务以及行级锁，而且一个毫无疑问的缺陷是崩溃后无法安全恢复。

对于只读的数据，或者表比较小，可以忍受修复操作，则依然可以使用MyISAM（但请不要默认使用MyISAM，而是应该默认使用InnoDB）

~~Memory引擎~~

InnoDB和MyISAM比较

1. 事务： InnoDB 支持事务，MyISAM 不支持事务。这是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一。
2. 外键： InnoDB 支持外键，而 MyISAM 不支持。对一个包含外键的 InnoDB 表转为 MYISAM 会失败。
3. 锁粒度：InnoDB 最小的锁粒度是行锁，MyISAM 最小的锁粒度是表锁。一个更新语句会锁住整张表，导致其他查询和更新都会被阻塞，因此并发访问受限。这也是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一。
4. 索引： MyISAM主键索引是非聚簇索引，InnoDB主键索引是聚簇索引
5. MyISAM：如果有大量的SELECT，使用MyISAM

   InnoDB：如果有大量的INSERT或UPDATE，使用InnoDB表

MySQL调优

使用Explain

id列

select_type列

这一列显示是简单还是复杂SELECT(如果是复杂，那么是复杂类型中的哪一种)。

- **SIMPLE** 意味着查询不包含子查询和UNION
- **PRIMARY** 如果查询有复杂的子部分，最外层部分标为PRIMARY，其余部分标记如下四种
- **SUBQUERY**
- **DERIVED**

- **UNION**
- **UNION RESULT**

table 列

显示对应行正在访问哪个表。就是那个表的表名，或是该表的别名（如果SQL定义了别名）

type列 （重要）

访问类型

- **ALL** 全表扫描。
- **INDEX** 全索引扫描，遍历索引树来获取数据行。如果数据分布不集中，会产生随机IO。如果在Extra列中看到 “Using index"， 说明MySQL正在使用覆盖索引， 它只扫描索引的数据，而不是按索引次序的每一行。它比按索引次序全表扫描的开销要少很多。
- **range** 只检索给定范围的行，使用一个索引来选择行。常见于使用 =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, IN()或者like等运算符的查询中。
- **ref** 一种索引匹配，返回所有匹配某个单个值的行。访问非唯一索引或者唯一性索引的非唯一前缀才会发生。可能会找到多个符合条件的行，因此，是查找和扫描的混合体。
- **eq_ref** 唯一性索引扫描，最多只返回一条符合条件的记录。MySQL对这类查找优化很好，因为无需估计匹配行的范围，找到匹配后也不会继续查找。
- **const** 通过索引1次找到，用于比较主键或者unique索引，只匹配一行数据。如将主键置于where列表中，MySQL就能将查询转换为一个常量

  ```
  SELECT * FROM tbl_name WHERE primary_key=1;
  SELECT * FROM tbl_name  WHERE primary_key_part1=1 AND primary_key_part2=2;
  ```

- **system** 表只有一行记录，const类型特例
- **NULL**

rows列

MySQL估计需要查找的行数

Extra列

包含不适合在其他列显示的额外信息。

- "Using index" 将使用覆盖索引，避免回表
- "Using where"
- "Using temporary" 对查询结果排序时会使用一个临时表
- "Using filesort"
- "Range checked for each record(index map: N)"

为什么SQL语句执行的很慢?

第一种情况，大多数情况下很正常，偶尔很慢

- 数据库在刷新脏页，redo log写满了需要同步到磁盘
- 执行时遇到锁：行级锁、表级锁

第二种情况，这条SQL语句一直都很慢
<<<<<<< HEAD

- 无索引
- 索引失效：例如：由于对字段进行运算、函数操作导致无法用索引。导致全表扫描。
- 选错索引
- 快照读，回滚日志巨大，需要计算，导致查询慢。

刷新脏页（flush）

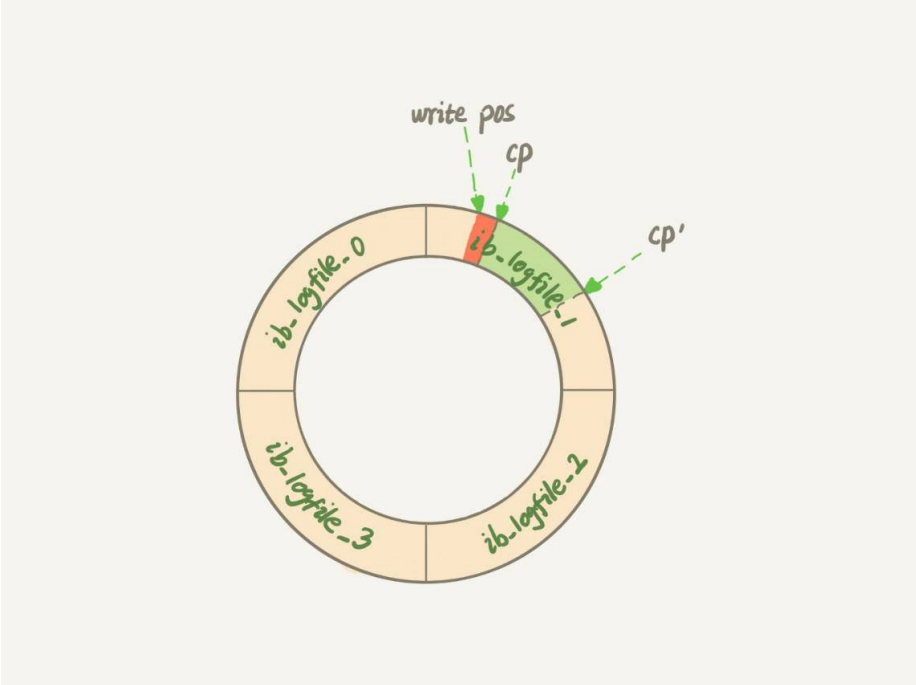
InnoDB 在处理更新语句的时候，只做了写日志这一个磁盘操作。这个日志叫作 redo log（重做日志），在更新内存写完 redo log 后，就返回给客户端，本次更新成功。

当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。内存数据写入到磁盘后，内存和磁盘上的数据页的内容就一致了，称为“干净页”。

刷脏页的时候，会写硬盘，会很慢。**什么情况会引发数据库的flush过程**?

- Innodb的redo log写满
- 内存不足
- 系统空闲时
- 系统关闭时

第一种情况，Innodb的redo log写满了。这时候系统会停止所有更新操作，把 checkpoint 往前推进，redo log 留出空间可以继续写。



比如图 中，把 checkpoint 位置从 CP 推进到 CP’，就需要将两个点之间的日志（浅绿色部分），对应的所有脏页都 flush 到磁盘上。之后，图中从 write pos 到 CP’之间就是可以再写入的 redo log 的区域。

第二种情况，内存不足。 当需要新的内存页，而内存不够用的时候，就要淘汰一些数据页，空出内存给别的数据页使用。如果淘汰的是“脏页”，就要先将脏页写到磁盘。

第三种情况，系统空闲。 MySQL认为系统空闲时，自动flush。

第四种情况，系统关闭。 这时候，MySQL 会把内存的脏页都 flush 到磁盘上，这样下次 MySQL 启动的时候，就可以直接从磁盘上读数据，启动速度会很快。

控制刷脏页的速度，要合理地设置`innodb_io_capacity`的值，并且平时要多关注脏页比例，不要让它经常接近 **75%**。

- 无索引
- 索引失效：例如：由于对字段进行运算、函数操作导致无法用索引。导致全表扫描。
- 选错索引
- 快照读，回滚日志巨大，需要计算，导致查询慢。

刷新脏页（flush）

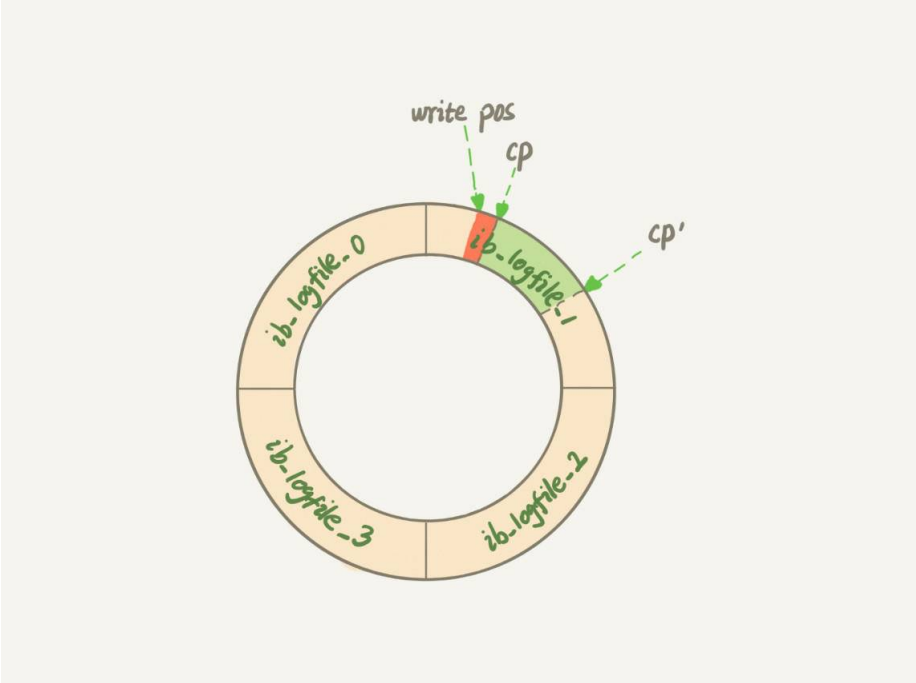
InnoDB 在处理更新语句的时候，只做了写日志这一个磁盘操作。这个日志叫作 redo log（重做日志），在更新内存写完 redo log 后，就返回给客户端，本次更新成功。

当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。内存数据写入到磁盘后，内存和磁盘上的数据页的内容就一致了，称为“干净页”。

刷脏页的时候，会写硬盘，会很慢。**什么情况会引发数据库的flush过程**?

- Innodb的redo log写满
- 内存不足
- 系统空闲时
- 系统关闭时

第一种情况，Innodb的redo log写满了。这时候系统会停止所有更新操作，把 checkpoint 往前推进，redo log 留出空间可以继续写。



比如图 中，把 checkpoint 位置从 CP 推进到 CP’，就需要将两个点之间的日志（浅绿色部分），对应的所有脏页都 flush 到磁盘上。之后，图中从 write pos 到 CP’之间就是可以再写入的 redo log 的区域。

第二种情况，内存不足。当需要新的内存页，而内存不够用的时候，就要淘汰一些数据页，空出内存给别的数据页使用。如果淘汰的是“脏页”，就要先将脏页写到磁盘。

第三种情况，系统空闲。 MySQL认为系统空闲时，自动flush。

第四种情况，系统关闭。 这时候，MySQL 会把内存的脏页都 flush 到磁盘上，这样下次 MySQL 启动的时候，就可以直接从磁盘上读数据，启动速度会很快。

控制刷脏页的速度，要合理地设置`innodb_io_capacity`的值，并且平时要多关注脏页比例，不要让它经常接近 **75%**。

等待MDL锁（表锁）

使用`show processlist`命令查看`Waiting for table metadata lock`的示意图

```
mysql> show processlist;
+----+------+-----------------+------+---------+------+---------------------------------+----------------------------+
| Id | User | Host            | db   | Command | Time | State                           | Info                       |
+----+------+-----------------+------+---------+------+---------------------------------+----------------------------+
|  5 | root | localhost:61558 | test | Query   |    0 | init                            | show processlist           |
|  7 | root | localhost:63852 | test | Sleep   |   31 |                                 | NULL                       |
|  8 | root | localhost:63870 | test | Query   |   25 | Waiting for table metadata lock | select * from t where id=1 |
+----+------+-----------------+------+---------+------+---------------------------------+----------------------------+
3 rows in set (0.00 sec)
```

出现这个状态表示的是，现在有一个线程正在表 t 上请求或者持有 MDL 写锁，把 select 语句堵住了。

查找`sys.schema_table_lock_waits`表，我们就可以直接找出造成阻塞的 process id，把这个连接用 kill 命令断开即可。

查询方法：

```
select blocking_pid from sys.schema_table_lock_waits;
```

```
mysql> select blocking_pid  from sys.schema_table_lock_waits;
+---------------------+
| blocking_pid        |
+---------------------+
|                   4 |
+---------------------+
```

等待行锁

如果你用的是 MySQL 5.7 版本，可以通过`sys.innodb_lock_waits`表查到谁在占有锁。

查询方法：

```
select * from sys.innodb_lock_waits where locked_table='`test`.`t`'\G
```

```
mysql> select * from sys.innodb_lock_waits where locked_table='`test`.`t`'\G
*************************** 1. row ***************************
                wait_started: 2018-12-13 20:12:35
                    wait_age: 00:00:08
               wait_age_secs: 8
                locked_table: `test`.`t`
                locked_index: PRIMARY
                 locked_type: RECORD
              waiting_trx_id: 421668144410224
         waiting_trx_started: 2018-12-13 20:12:35
             waiting_trx_age: 00:00:08
     waiting_trx_rows_locked: 1
   waiting_trx_rows_modified: 0
                 waiting_pid: 8
               waiting_query: select * from t where id=1 lock in share mode
             waiting_lock_id: 421668144410224:23:4:2
           waiting_lock_mode: S
             blocking_trx_id: 1101302
                blocking_pid: 4
              blocking_query: NULL
            blocking_lock_id: 1101302:23:4:2
          blocking_lock_mode: X
        blocking_trx_started: 2018-12-13 20:01:57
            blocking_trx_age: 00:10:46
    blocking_trx_rows_locked: 1
  blocking_trx_rows_modified: 1
     sql_kill_blocking_query: KILL QUERY 4
sql_kill_blocking_connection: KILL 4
1 row in set, 3 warnings (0.00 sec)
```

可以看到，这个信息很全，4 号线程是造成堵塞的罪魁祸首。而干掉这个罪魁祸首的方式，就是 `KILL 4`，断开这个连接。连接断开，自动回滚，释放锁。

选错索引

优化器选择索引的目的，是找到一个最优的执行方案，并用最小的代价去执行语句。在数据库里面，扫描行数是影响执行代价的因素之一。扫描的行数越少，意味着访问磁盘数据的次数越少，消耗的 CPU 资源越少。当然，扫描行数并不是唯一的判断标准，优化器还会结合是否使用临时表、是否排序等因素进行综合判断。

用explain分析优化器是否选错索引。

解决方法：

1. 采用 `force index` 强行选择一个索引
2. 修改语句，引导MySQL使用我们期望的索引。
3. 新建一个更合适的索引，来提供给优化器做选择，或者删掉误用的索引。

change buffer

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，InnoDB 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。

需要说明的是，虽然名字叫作 change buffer，实际上它是可以持久化的数据。也就是说，change buffer 在内存中有拷贝，也会被写入到磁盘上。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。

显然，如果能够将更新操作先记录在 change buffer，减少读磁盘，语句的执行速度会得到明显的提升。而且，数据读入内存是需要占用 buffer pool 的，所以这种方式还能够避免占用内存，提高内存利用率。

change buffer的应用场景（普通索引、写多读少）

change buffer 只限于用在普通索引的场景下

通索引的所有场景，使用 change buffer 都可以起到加速作用吗?

因为 merge 的时候是真正进行数据更新的时刻，而 change buffer 的主要目的就是将记录的变更动作缓存下来，所以在一个数据页做 merge 之前，change buffer 记录的变更越多（也就是这个页面上要更新的次数越多），收益就越大。因此，对于**写多读少**的业务来说，页面在写完以后马上被访问到的概率比较小，此时 change buffer 的使用效果最好。

这种业务模型常见的就是账单类、日志类的系统。反过来，假设一个业务的更新模式是写入之后马上会做查询，那么即使满足了条件，将更新先记录在 change buffer，但之后由于马上要访问这个数据页，会立即触发 merge 过程。这样随机访问 IO 的次数不会减少，反而增加了 change buffer 的维护代价。所以，对于这种业务模式来说，change buffer 反而起到了副作用。

change buffer和redo log区别

redo log 主要节省的是随机写磁盘的 IO 消耗（转成顺序写），而 change buffer 主要节省的则是随机读磁盘的 IO 消耗。

以下放到一个流程中来解释。

现在，我们要在表上执行这个插入语句：

```
insert into t(id,k) values(id1,k1),(id2,k2);
```

这里，我们假设当前 k 索引树的状态，查找到位置后，k1 所在的数据页在内存 (InnoDB buffer pool) 中，k2 所在的数据页不在内存中。如图 2 所示是带 change buffer 的更新状态图。

分析这条更新语句，你会发现它涉及了四个部分：内存、redo log（ib_log_fileX）、 数据表空间（t.ibd）、系统表空间（ibdata1）。

这条更新语句做了如下的操作（按照图中的数字顺序）：

- Page 1 在内存中，直接更新内存；
- Page 2 没有在内存中，就在内存的 change buffer 区域，记录下“我要往 Page 2 插入一行”这个信息
- 将上述两个动作记入 redo log 中（图中 3 和 4）。

做完上面这些，事务就可以完成了。所以，你会看到，执行这条更新语句的成本很低，就是写了两处内存，然后写了一处磁盘（两次操作合在一起写了一次磁盘），而且还是顺序写的。

同时，图中的两个虚线箭头，是后台操作，不影响更新的响应时间。

那么，之后的读请求怎么处理呢?

```
select * from t where k in (k1, k2)
```

如果读语句发生在更新语句后不久，内存中的数据都还在，那么此时的这两个读操作就与系统表空间（ibdata1）和 redo log（ib_log_fileX）无关了。所以，我在图中就没画出这两部分。

从图中可以看到：

- 读 Page 1 的时候，直接从内存返回。有几位同学在前面文章的评论中问到，WAL 之后如果读数据，是不是一定要读盘，是不是一定要从 redo log 里面把数据更新以后才可以返回？其实是不用的。你可以看一下图 3 的这个状态，虽然磁盘上还是之前的数据，但是这里直接从内存返回结果，结果是正确的。
- 要读 Page 2 的时候，需要把 Page 2 从磁盘读入内存中，然后应用 change buffer 里面的操作日志，生成一个正确的版本并返回结果。

可以看到，直到需要读 Page 2 的时候，这个数据页才会被读入内存。

主从复制

主从复制

`master`把数据通过`I/O`线程写入binlog日志，`slave`通过一个`I/O`线程与主服务器保持通信，并订阅`master`的`binlog`，如果发生变化，则会复制到自己的relaylog中，然后slave的一个SQL线程会把相关的“事件”执行到自己的数据库中。

配置：

- **主服务器：**
- 开启二进制日志
  - 配置唯一的server-id
  - 获得master二进制日志文件名及位置
  - 创建一个用于slave和master通信的用户账号
- **从服务器：**

> 094e380a47012cf19fd12ffc3781cb9870152f58

等待MDL锁（表锁）

使用`show processlist`命令查看`Waiting for table metadata lock`的示意图

```
mysql> show processlist;
+----+------+-----------------+------+---------+------+--------------------------------+---------------------------+
| Id | User | Host            | db   | Command | Time | State                          | Info                      |
+----+------+-----------------+------+---------+------+--------------------------------+---------------------------+
|  5 | root | localhost:61558 | test | Query   |    0 | init                           | show processlist          |
|  7 | root | localhost:63852 | test | Sleep   |   31 |                                | NULL                      |
|  8 | root | localhost:63870 | test | Query   |   25 | Waiting for table metadata lock | select * from t where id=1 |
+----+------+-----------------+------+---------+------+--------------------------------+---------------------------+
3 rows in set (0.00 sec)
```

出现这个状态表示的是，现在有一个线程正在表 t 上请求或者持有 MDL 写锁，把 select 语句堵住了。

查找`sys.schema_table_lock_waits`表，我们就可以直接找出造成阻塞的 process id，把这个连接用 kill 命令断开即可。

查询方法：

```
select blocking_pid from sys.schema_table_lock_waits;
```

```
mysql> select blocking_pid  from sys.schema_table_lock_waits;
+-----------------+
| blocking_pid    |
+-----------------+
|               4 |
+-----------------+
```

等待行锁

如果你用的是 MySQL 5.7 版本，可以通过 `sys.innodb_lock_waits` 表查到谁在占有锁。

查询方法：

```
select * from sys.innodb_lock_waits where locked_table='`test`.`t`'\G
```

```
mysql> select * from sys.innodb_lock_waits where locked_table='`test`.`t`'\G
*************************** 1. row ***************************
                wait_started: 2018-12-13 20:12:35
                    wait_age: 00:00:08
               wait_age_secs: 8
                locked_table: `test`.`t`
                locked_index: PRIMARY
                 locked_type: RECORD
              waiting_trx_id: 421668144410224
         waiting_trx_started: 2018-12-13 20:12:35
             waiting_trx_age: 00:00:08
     waiting_trx_rows_locked: 1
   waiting_trx_rows_modified: 0
                 waiting_pid: 8
               waiting_query: select * from t where id=1 lock in share mode
             waiting_lock_id: 421668144410224:23:4:2
           waiting_lock_mode: S
             blocking_trx_id: 1101302
                blocking_pid: 4
              blocking_query: NULL
            blocking_lock_id: 1101302:23:4:2
          blocking_lock_mode: X
        blocking_trx_started: 2018-12-13 20:01:57
            blocking_trx_age: 00:10:46
    blocking_trx_rows_locked: 1
  blocking_trx_rows_modified: 1
     sql_kill_blocking_query: KILL QUERY 4
sql_kill_blocking_connection: KILL 4
1 row in set, 3 warnings (0.00 sec)
```

可以看到，这个信息很全，4 号线程是造成堵塞的罪魁祸首。而干掉这个罪魁祸首的方式，就是 `KILL 4`，断开这个连接。连接断开，自动回滚，释放锁。

选错索引

优化器选择索引的目的，是找到一个最优的执行方案，并用最小的代价去执行语句。在数据库里面，扫描行数是影响执行代价的因素之一。扫描的行数越少，意味着访问磁盘数据的次数越少，消耗的 CPU 资源越少。当然，扫描行数并不是唯一的判断标准，优化器还会结合是否使用临时表、是否排序等因素进行综合判断。

用explain分析优化器是否选错索引。

解决方法：

1. 采用 `force index`强行选择一个索引
2. 修改语句，引导MySQL使用我们期望的索引。
3. 新建一个更合适的索引，来提供给优化器做选择，或者删掉误用的索引。

主从复制

主从复制

`master`把数据通过`I/O`线程写入binlog日志，`slave`通过一个`I/O`线程与主服务器保持通信，并订阅`master`的`binlog`，如果发生变化，则会复制到自己的relaylog中，然后slave的一个SQL线程会把相关的“事件”执行到自己的数据库中。

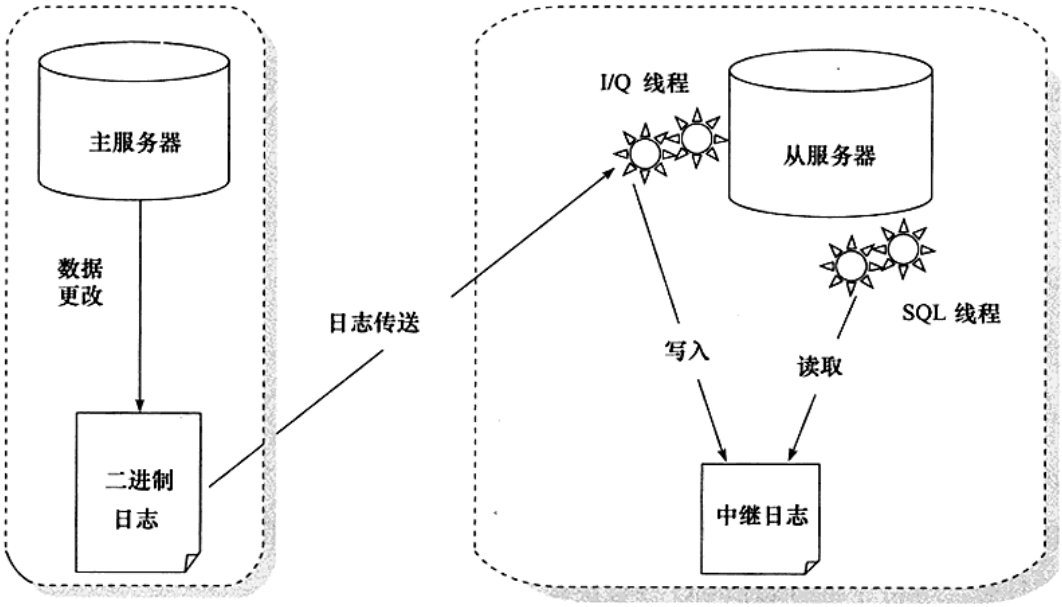


图 8-4　MySQL 数据库的复制工作原理

配置：

- **主服务器：**
- 开启二进制日志
  - 配置唯一的server-id
  - 获得master二进制日志文件名及位置
  - 创建一个用于slave和master通信的用户账号
- **从服务器：**
- 配置唯一的server-id
  - 使用master分配的用户账号读取master二进制日志
- 启用slave服务

主从延迟

原因

解决方法

半同步复制

半同步复制　master写入binlog日志之后，就会将强制此时立即将数据同步到slave，slave将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库的ack之后才会认为写操作完成了。

并行复制

slave开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这是库级别的并行。 缓解主从延迟。

读写分离

主库宕机