一、基础篇

1.1 MySQL的执行流程

MySQL 的架构共分为两层：**Server 层和存储引擎层**，

- **Server 层负责建立连接、分析和执行 SQL。**MySQL 大多数的核心功能模块都在这实现，主要包括连接器，查询缓存、解析器、预处理器、优化器、执行器等。另外，所有的内置函数（如日期、时间、数学和加密函数等）和所有跨存储引擎的功能（如存储过程、触发器、视图等。）都在 Server 层实现。
- **存储引擎层负责数据的存储和提取。**支持 InnoDB、MyISAM、Memory 等多个存储引擎，不同的存储引擎共用一个 Server 层。现在最常用的存储引擎是 InnoDB，从 MySQL 5.5 版本开始， InnoDB 成为了 MySQL 的默认存储引擎。我们常说的索引数据结构，就是由存储引擎层实现的，不同的存储引擎支持的索引类型也不相同，比如 InnoDB 支持索引类型是 B+树 ，且是默
- 认使用，也就是说在数据表中创建的主键索引和二级索引默认使用的是 B+ 树索引。

1.建立连接

MySQL 是基于 TCP 协议进行传输的，连接的过程需要先经过 TCP 三次握手。

空闲连接超过了`wait_timeout`参数（默认值是 8 小时），就会直接断开

MySQL 服务支持的**最大连接数**由 max_connections 参数控制

MySQL也有**长连接和短连接**的概念

```
// 短连接
连接 mysql 服务（TCP 三次握手）
执行sql
断开 mysql 服务（TCP 四次挥手）

// 长连接
连接 mysql 服务（TCP 三次握手）
执行sql
执行sql
执行sql
断开 mysql 服务（TCP 四次挥手）
```

因为 MySQL 在执行查询过程中临时使用内存管理连接对象，这些连接对象资源只有在连接断开时才会释放。如果长连接累计很多，将导致 MySQL 服务占用内存太大，有可能会被系统强制杀掉，这样会发生 MySQL 服务异常重启的现象。

> 长连接占用内存的解决方案

第一种，**定期断开长连接**。既然断开连接后就会释放连接占用的内存资源，那么我们可以定期断开长连接。

第二种，**客户端主动重置连接**。MySQL 5.7 版本实现了 `mysql_reset_connection()` 函数的接口，注意这是接口函数不是命令，那么当客户端执行了一个很大的操作后，在代码里调用 mysql_reset_connection 函数来重置连接，达到释放内存的效果。这个过程不需要重连和重新做权限验证，但是会将连接恢复到刚刚创建完时的状态。

2.查询缓存

如果 SQL 是查询语句（select 语句），MySQL 就会先去查询缓存（Query Cache）里查找缓存数据，看看之前有没有执行过这一条命令，这个查询缓存是以 key-value 形式保存在内存中的，key 为 SQL 查询语句，value 为 SQL 语句查询的结果。如果查到了这条语句就直接返回value的结果，否则继续执行。

MySQL 8.0 版本直接将查询缓存删掉了，也就是说 MySQL 8.0 开始，执行一条 SQL 查询语句，不会再走到查询缓存这个阶段了。

3.解析SQL

解析器会做如下两件事情。

第一件事情，**词法分析**。MySQL 会根据你输入的字符串识别出关键字出来，例如，SQL语句 select username from userinfo，在分析之后，会得到4个Token，其中有2个Keyword，分别为select和from：

关键字	非关键字	关键字	非关键字
select	username	from	userinfo

第二件事情，**语法分析**。根据词法分析的结果，语法解析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法，如果没问题就会构建出 SQL 语法树，这样方便后面模块获取 SQL 类型、表名、字段名、 where 条件等等。

4.执行SQL

有三个阶段：

- prepare 阶段，也就是预处理阶段；
  - 检查 SQL 查询语句中的表或者字段是否存在；
  - 将 `select *` 中的 `*` 符号，扩展为表上的所有列；
- optimize 阶段，也就是优化阶段；
  - **优化器主要负责将 SQL 查询语句的执行方案确定下来**，比如多个索引的选择。select id from product where id > 1 and name like 'i%' 选择二级索引

- execute 阶段，也就是执行阶段；

1.2 MySQL一行记录的存储

1.数据库文件的存放

每创建一个 database（数据库）都会在 /var/lib/mysql/ 目录里面创建一个以 database 为名的目录，然后保存表结构和表数据的文件都会存放在这个目录里。

创建一个名为my_test的数据库，在其中创建一个叫t_order的表。

在/var/lib/mysql/my_test目录中可以看到有三个文件

```
db.opt
t_order.frm
t_order.ibd
```

- **db.opt**，用来存储当前数据库的默认字符集和字符校验规则。
- **t_order.frm** ，t_order 的**表结构**会保存在这个文件。在 MySQL 中建立一张表都会生成一个.frm 文件，该文件是用来保存每个表的元数据信息的，主要包含表结构定义。
- **t_order.ibd**，t_order 的**表数据**会保存在这个文件。表数据既可以存在共享表空间文件（文件名：ibdata1）里，也可以存放在独占表空间文件（文件名：表名字.ibd）。这个行为是由参数 innodb_file_per_table 控制的，若设置了参数 innodb_file_per_table 为 1，则会将存储的数据、索引等信息单独存储在一个独占表空间，从 MySQL 5.6.6 版本开始，它的默认值就是 1 了，因此从这个版本之后， MySQL 中每一张表的数据都存放在一个独立的 .ibd 文件。

> 表空间的文件结构：

表空间由段（segment）、区（extent）、页（page）、行（row）组成

数据库表中的记录都是按行存放的。

InnoDB 的数据是按「页」为单位来读写的，默认每个页的大小为 16KB。

在表中数据量大的时候，为某个索引分配空间的时候就不再按照页为单位分配了，而是按照区（extent）为单位分配。每个区的大小为 1MB，对于 16KB 的页来说，连续的 64 个页会被划为一个区，这样就使得链表中相邻的页的物理位置也相邻，就能使用顺序 I/O 了。

段是由多个区（extent）组成的。段一般分为数据段、索引段和回滚段等。

- 索引段：存放 B + 树的非叶子节点的区的集合；
- 数据段：存放 B + 树的叶子节点的区的集合；
- 回滚段：存放的是回滚数据的区的集合，之前讲事务隔离 (opens new window)的时候就介绍到了 MVCC 利用了回滚段实现了多版本查询数据。

2.InnoDB存储引擎的行格式

InnoDB 提供了 4 种行格式：**Redundant**、**Compact**、**Dynamic**和 **Compressed** 行格式。

Compact行格式如下：

- **变长字段长度列表**：记录各个变长字段varchar(n)真实占用字节数
- **NULL值列表**：每个非NOT NULL字段组成的二进制bit位，1表示为NULL,0表示不是NULL。注意NULL 值列表必须用整数个字节的位表示（1字节8位），如果使用的二进制位个数不足整数个字节，则在字节的高位补0。当数据表的字段都定义成 NOT NULL 的时候，这时候表里的行格式就不会有 NULL 值列表了。
- **记录头信息**：包含多个字段，delete_mask、delete_mask，record_type等
- **记录的真实数据**：
  - row_id 如果我们建表的时候指定了主键或者唯一约束列，那么就没有 row_id 隐藏字段了。如果既没有指定主键，又没有唯一约束，那么 InnoDB 就会为记录添加 row_id 隐藏字段。row_id不是必需的，占用 6 个字节。
  - trx_id 这个数据是由哪个事务生成的，占用 6 个字节
  - roll_pointer 记录上一个版本的指针。占用 7 个字节

> 为什么 变长字段长度列表 和 NULL值列表 的信息要按照逆序存放?

「记录头信息」中指向下一个记录的指针，指向的是下一条记录的「记录头信息」和「真实数据」之间的位置，这样的好处是向左读就是记录头信息，向右读就是真实数据，读取的数据就和记录的真实数据一一对应起来了。

> varchar(n) 中 n 最大取值为多少?

MySQL 规定除了 TEXT、BLOBs 这种大对象类型之外，其他所有的列（不包括隐藏列和记录头信息）占用的字节长度加起来不能超过 65535 个字节。

archar(n) 中 n 最大取值为 65532。

计算公式：65535 - 变长字段字节数列表所占用的字节数 - NULL值列表所占用的字节数 = 65535 - 2 - 1 = 65532。

二、索引篇

2.1 基础

索引就相当于数据的目录

> 索引的分类

- 按「数据结构」分类：**B+tree索引、Hash索引、Full-text索引。**
- 按「物理存储」分类：**聚簇索引（主键索引）、二级索引（辅助索引）。**
- 按「字段特性」分类：**主键索引、唯一索引、普通索引、前缀索引。**
- 按「字段个数」分类：**单列索引、联合索引。**

> 在创建表时，InnoDB 存储引擎会根据不同的场景选择不同的列作为索引

- 如果有主键，默认会使用**主键作为聚簇索引**的索引键（key）；
- 如果没有主键，就选择第一个不包含 NULL 值的**唯一列作为聚簇索引**的索引键（key）；
- 在上面两个都没有的情况下，InnoDB 将自动生成一个**隐式自增 id 列作为聚簇索引**的索引键（key）；

其它索引都属于辅助索引（Secondary Index），也被称为二级索引或非聚簇索引。**创建的主键索引和二级索引默认使用的是 B+Tree 数据结构**，每一个索引都会单独创建一个B+树

> 聚簇索引与二级索引

- 主键索引的 B+Tree 的叶子节点存放的是实际数据，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子节点里；
- 二级索引的 B+Tree 的叶子节点存放的是主键值，而不是实际数据。

在查询时使用了二级索引，**如果查询的列能在二级索引里查询的到，那么就不需要回表，这个过程就是覆盖索引。**如果查询的数据不在二级索引里，就会先检索二级索引，找到对应的叶子节点，**获取到主键值后，然后再检索主键索引，就能查询到数据了，这个过程就是回表。**

> 联合索引

将多个字段组合成一个索引，该索引就被称为联合索引。

使用联合索引时，存在**最左匹配原则**，查询从索引的最左列开始不能够跳跃。如果不遵循「最左匹配原则」，联合索引会失效

比如，如果创建了一个 `(a, b, c)` 联合索引，对应的B+树先按 a 排序，在 a 相同的情况再按 b 排序，在 b 相同的情况再按 c 排序。如果查询条件是以下这几种，就不符合最左匹配原则：

- where b=2;
- where c=3;
- where b=2 and c=3;

在遇到范围查询（如 >、<）的时候，无法用到联合索引。然而，对于 ⩾、⩽、BETWEEN、like前缀匹配的范围查询，并不会停止匹配。

> **索引下推**

对于联合索引（a, b），在执行 `select * from table where a > 1 and b = 2` 语句的时候，只有 a 字段能用到索引，那在联合索引的 B+Tree 找到第一个满足条件的主键值（ID 为 2）后，还需要判断其他条件是否满足（看 b 是否等于 2）。

- 在 MySQL 5.6 之前，只能从 ID2 （主键值）开始**一个个回表**，到「主键索引」上找出数据行，再对比 b 字段值。
- 而 MySQL 5.6 引入的**索引下推优化**（index condition pushdown）， **可以在联合索引遍历过程中，对联合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。**

> **联合索引进行排序**

这里出一个题目，针对针对下面这条 SQL，你怎么通过索引来提高查询效率呢？

```
select * from order where status = 1 order by create_time asc
```

有的同学会认为，单独给 status 建立一个索引就可以了。

但是更好的方式给 status 和 create_time 列建立一个联合索引，因为这样可以避免 MySQL 数据库发生文件排序。

因为在查询时，如果只用到 status 的索引，但是这条语句还要对 create_time 排序，这时就要用文件排序 filesort，也就是在 SQL 执行计划中，Extra 列会出现 Using filesort。

所以，要利用索引的有序性，在 status 和 create_time 列建立联合索引，**对应B+树直接排好了序**，这样根据 status 筛选后的数据就是按照 create_time 排好序的，避免在文件排序，提高了查询效率。

2.2 InnoDB存储数据

InnoDB 的数据是按「数据页」为单位来读写的，数据页的默认大小是 16KB

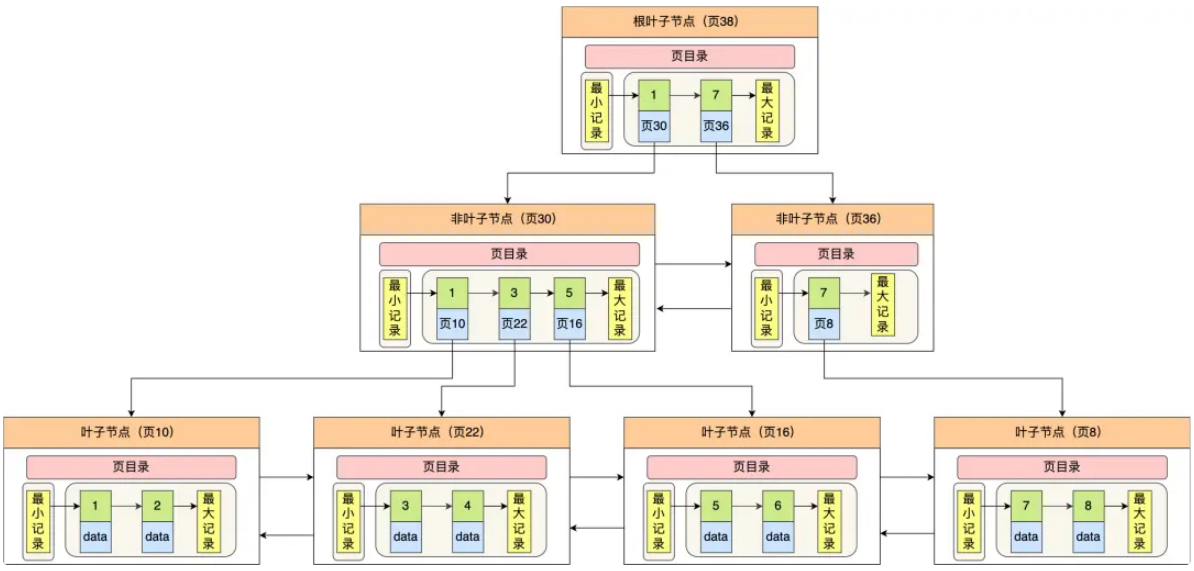
数据页包含7个部分

名称	说明
文件头 File Header	文件头，表示页的信息
页头 Page Header	页头，表示页的状态信息
最小和最大记录 Infimum+supremum	两个虚拟的伪记录，分别表示页中的最小记录和最大记录
用户记录 User Records	存储行记录内容
空闲空间 Free Space	页中还没被使用的空间
页目录 Page Directory	存储用户记录的相对位置，对记录起到索引作用
文件尾 File Tailer	校验页是否完整

> 数据页中有一个**页目录**，起到记录的索引作用

所有的记录被分为了多组，页目录用来存储每组最后一条记录的地址偏移量（槽），**页目录就是由多个槽组成的，槽相当于分组记录的索引**，通过**二分查询**的记录在哪个槽（哪个记录分组），定位到槽后，再遍历槽内的所有记录，找到对应的记录

> B+树如何查询



B+ 树中的**每个节点都是一个数据页**

只有叶子节点（最底层的节点）才存放了数据，**非叶子节点（其他上层节）仅用来存放目录项作为索引**

2.3 什么时候需要/不需要创建索引?

索引最大的好处是提高查询速度，但是索引也是有缺点的，比如：

- 需要占用物理空间，数据数量越多，索引占用空间越大；
- 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增大；
- 会降低表的增删改的效率，因为每次增删改数据时都要修改索引，B+ 树为了维护索引有序性，都需要进行动态维护。

所以，索引不是万能钥匙，它也是根据场景来使用的。

> 什么情况下需要索引?

- 经常用于 `WHERE` 查询条件的字段，这样能够提高整个表的查询速度
- 经常用于 `GROUP BY` 和 `ORDER BY` 的字段

> 什么情况下不需要创建索引?

- **表记录太少**
- **经常更新的字段**不用创建索引，由于要维护 B+Tree的有序性，那么就需要频繁的重建索引，这个过程是会影响数据库性能的
- **字段中存在大量重复数据且分布平均，不需要创建索引**，假如一个表有10万行记录，有一个字段A只有T和F两种值，且每个值的分布概率大约为50%，那么对这种表A字段建索引一般不
  会提高数据库的查询速度。
- 经常和主字段一块查询但主字段索引值比较多的表字段

2.4 索引优化方式

1.前缀索引优化

前缀索引顾名思义就是**使用某个字段中字符串的前几个字符建立索引**，那我们为什么需要使用前缀来建立索引呢?

使用前缀索引是为了减小索引字段大小，可以增加一个索引页中存储的索引值，有效提高索引的查询速度。在一些大字符串的字段作为索引时，使用前缀索引可以帮助我们减小索引项的大小。

不过，前缀索引有一定的局限性，例如：

- `order by` 就无法使用前缀索引；
- 无法把前缀索引用作覆盖索引；

2.覆盖索引优化

在使用二级索引的时候尽量达到一个索引覆盖的效果

覆盖索引是指 SQL 中 query 的所有字段，在索引 B+Tree 的叶子节点上都能找得到的那些索引，从二级索引中查询得到记录，而不需要通过聚簇索引查询获得，可以避免回表的操作。

假设我们只需要查询商品的名称、价格，有什么方式可以避免回表呢?

我们**可以建立一个联合索引**，即「商品ID、名称、价格」作为一个联合索引。如果索引中存在这些数据，查询将不会再次检索主键索引，从而避免回表。

所以，使用覆盖索引的好处就是，不需要查询出包含整行记录的所有信息，也就减少了大量的 I/O 操作。

3.使用自增主键

如果我们使用自增主键，那么每次插入的新数据就会按顺序添加到当前索引节点的位置，不需要移动已有的数据，当页面写满，就会自动开辟一个新页面。因为每次**插入一条新记录，都是追加操作，不需要重新移动数据**，因此这种插入数据的方法效率非常高。

如果我们使用非自增主键，由于每次插入主键的索引值都是随机的，因此每次插入新的数据时，就可能会**插入到现有数据页中间的某个位置**，这将不得不移动其它数据来满足新数据的插入，甚至需要从一个页面复制数据到另外一个页面，我们通常将这种情况称为**页分裂。** 页分裂**还有可能会造成大量的内存碎片，导致索引结构不紧凑，从而影响查询效率。**

4.索引最好设置为 NOT NULL

- 第一原因：索引列存在 NULL 就会导致**优化器在做索引选择的时候更加复杂**，更加难以优化，因为可为 NULL 的列会使索引、索引统计和值比较都更复杂，比如进行索引统计时，count 会省略值为NULL 的行。
- 第二个原因：NULL 值是一个没意义的值，但是它会占用物理空间，所以会带来的存储空间的问题，因为 InnoDB 存储记录的时候，如果表中存在允许为 NULL 的字段，那么行格式 中**至少会用 1 字节空间存储 NULL 值列表**

5.防止索引失效

这里简单说一下，发生索引失效的情况：

- 当我们使用左或者左右模糊匹配的时候，也就是 like %xx 或者 like %xx%这两种方式都会造成索引失效；
- 当我们在查询条件中对索引列做了计算、函数、类型转换操作，这些情况下都会造成索引失效；
- 联合索引要能正确使用需要遵循最左匹配原则，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。
- 在 WHERE 子句中，如果在 OR 前的条件列是索引列，而在 OR 后的条件列不是索引列，那么索引会失效。

2.5 为什么MySQL采用B+树作为索引

二分查找树虽然是一个天然的二分结构，能很好的利用二分查找快速定位数据，但是它存在一种极端的情况，**每当插入的元素都是树内最大的元素，就会导致二分查找树退化成一个链表，此时查询复杂度就会从 O(logn)降低为 O(n)。**

为了解决二分查找树退化成链表的问题，就出现了自**平衡二叉树**，保证了查询操作的时间复杂度就会一直维持在 O(logn) 。但是它本质上还是一个二叉树，每个节点只能有 2 个子节点，随着元素的增多，树的高度会越来越高。

B 树和 B+ 都是通过多叉树的方式，会将树的高度变矮，所以这两个数据结构非常适合检索存于磁盘中的数据。

> B树和B+树的区别：

- **B+ 树的非叶子节点不存放实际的记录数据，仅存放索引**，因此数据量相同的情况下，相比存储即存索引又存记录的 B 树，B+树的非叶子节点可以存放更多的索引，因此 **B+ 树可以比 B 树更「矮胖」**，查询底层节点的磁盘 I/O次数会更少。
- **B+ 树有大量的冗余节点（叶子节点包含了非叶子节点中的索引）**，这些冗余索引让 B+ 树在**插入、删除的效率都更高**，比如删除根节点的时候，不会像 B 树那样会发生复杂的树的变化；
- **B+ 树叶子节点之间用双向链表连接了起来，有利于范围查询**，而 B 树要实现范围查询，因此只能通过树的遍历来完成范围查询，这会涉及多个节点的磁盘 I/O 操作，范围查询效率不如 B+ 树。

2.6 索引失效

6 种会发生索引失效的情况：

- 当我们使用**左或者左右模糊匹配**的时候，也就是 `like %xx` 或者 `like %xx%`这两种方式都会造成索引失效；
- 当我们在查询条件中**对索引列使用函数**，就会导致索引失效。`where length(name)=6;`
- 当我们在查询条件中**对索引列进行表达式计算**，也是无法走索引的。`where id+1=10;`
- MySQL 在遇到字符串和数字比较的时候，会自动把字符串转为数字，然后再进行比较。如果字符串是索引列，而条件语句中的输入参数是数字的话，那么索引列会发生隐式类型转换，由于**隐式类型转换是通过 CAST 函数实现的，等同于对索引列使用了函数**，所以就会导致索引失效。
- 联合索引要能正确使用需要遵循**最左匹配原则**，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。
- 在 WHERE 子句中，如果在 OR 前的条件列是索引列，而**在 OR 后的条件列不是索引列，那么索引会失效。**

三、事务篇

3.1 事务的特性

事务必须要遵守 4 个特性（ACID），分别如下：

- **原子性（Atomicity）**：一个事务中的所有操作，**要么全部完成，要么全部不完成**，不会结束在中间某个环节，而且事务在执行过程中发生错误，会被回滚到事务开始前的状态，就像这个事务从来没有执行过一样，就好比买一件商品，购买成功时，则给商家付了钱，商品到手；购买失败时，则商品在商家手中，消费者的钱也没花出去。
- **一致性（Consistency）**：是指**事务操作前和操作后，数据满足完整性约束**，多个事务对同一个数据读取的结果是正确的。数据库保持一致性状态。比如，用户 A 和用户 B 在银行分别有 800 元和 600 元，总共 1400 元，用户 A 给用户 B 转账 200 元，分为两个步骤，从 A 的账户扣除 200 元和对 B 的账户增加 200 元。一致性就是要求上述步骤操作后，最后的结果是用户 A 还有 600 元，用户 B 有 800 元，总共 1400 元，而不会出现用户 A 扣除了 200 元，但用户 B 未增加的情况（该情况，用户 A 和 B 均为 600 元，总共 1200 元）。
- **隔离性（Isolation）**：数**据库允许多个并发事务同时对其数据进行读写和修改的能力**，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致，因为多个事务同时使用相同的数据时，不会相互干扰，每个事务都有一个完整的数据空间，对其他并发事务是隔离的。也就是说，消费者购买商品这个事务，是不影响其他消费者购买的。
- **持久性（Durability）**：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

3.2 并行事务会引发什么问题?

1.脏读

如果一个事务「读到」了另一个「未提交事务修改过的数据」，就意味着发生了「脏读」现象。

2.不可重复读

在一个事务内多次读取同一个数据，如果出现前后两次读到的数据不一样的情况，就意味着发生了「不可重复读」现象。数据修改了

3.幻读

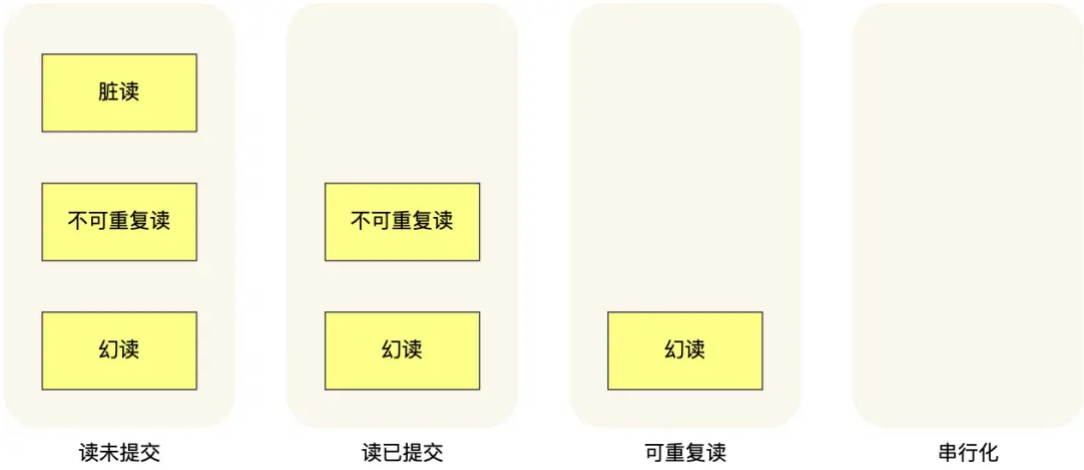
在一个事务内多次查询某个符合查询条件的「记录数量」，如果出现前后两次查询到的记录数量不一样的情况，就意味着发生了「幻读」现象。新增或删除了数据

3.3 事务的隔离级别有哪些?

SQL 标准提出了四种隔离级别来规避这些现象，隔离级别越高，性能效率就越低，这四个隔离级别如下：

- **读未提交（read uncommitted）**，指一个事务还没提交时，它做的变更就能被其他事务看到；
- **读提交（read committed）**，指一个事务提交之后，它做的变更才能被其他事务看到；
- **可重复读（repeatable read）**，指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，**MySQL InnoDB 引擎的默认隔离级别**；
- **串行化（serializable）**；会对记录加上读写锁，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行；

针对不同的隔离级别，并发事务时可能发生的现象也会不同。



> MySQL 可重复读隔离级别，完全解决幻读了吗?

MySQL InnoDB 引擎的默认隔离级别虽然是「可重复读」，但是它很大程度上避免幻读现象（并不是完全解决了），解决的方案有两种：

- 针对**快照读**（普通 select 语句），是**通过 MVCC 方式解决了幻读**，因为可重复读隔离级别下，事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，即使中途有其他事务插入了一条数据，是查询不出来这条数据的，所以就很好了避免幻读问题。
- 针对**当前读**（select ... for update 等语句），是**通过 next-key lock（记录锁+间隙锁）方式解决了幻读**，因为当执行 select ... for update 语句的时候，会加上 next-key lock，如果有其他事务在 next-key lock 锁范围内插入了一条记录，那么这个插入语句就会被阻塞，无法成功插入，所以就很好了避免幻读问题。

MySQL 可重复读隔离级别并没有彻底解决幻读，只是很大程度上避免了幻读现象的发生。

3.4 MVCC工作原理

数据记录中都包含下面两个隐藏列：

- trx_id，当一个事务对某条聚簇索引记录进行改动时，就会**把该事务的事务 id 记录在 trx_id 隐藏列里**；
- roll_pointer，每次对某条聚簇索引记录进行改动时，都会把旧版本的记录写入到 undo 日志中，然后**这个隐藏列是个指针，指向每一个旧版本记录**，于是就可以通过它找到修改前的记录。

读提交隔离级别是在**每个语句执行前都会重新生成一个 Read View，**而可重复读隔离级别**是启动事务时生成一个 Read View，**然后整个事务期间都在用这个 Read View。

通过Read View中的活跃事务id和undo日志中的旧版本数据可以实现事务的隔离

3.5 可重复读级别完全解决了幻读吗?

没有

第一种情况：

- 事务 A 第一次执行普通的 select 语句时生成了一个 ReadView，
- 事务 B 向表中新插入了一条 id = 5 的记录并提交。
- 接着，事务 A 对 id = 5 这条记录进行了更新操作，在这个时刻，这条新记录的 trx_id 隐藏列的值就变成了事务 A 的事务 id，事务A对该记录可见了
- 事务 A 再使用普通 select 语句去查询这条记录时就可以看到这条记录了，于是就发生了幻读。

第二种情况：

- T1 时刻：事务 A 先执行「快照读语句」：select * from t_test where id > 100 得到了 3 条记录。
- T2 时刻：事务 B 往插入一个 id= 200 的记录并提交；
- T3 时刻：事务 A 再执行「当前读语句」 select * from t_test where id > 100 for update 就会得到 4 条记录，此时也发生了幻读现象。

四、锁篇

4.1 各种锁及各自作用

锁分为**互斥锁/排他锁/写锁/X锁**和**共享锁/读锁/S锁**两种。只有读锁与读锁可以同时存在，而互斥锁与其他锁不能同时存在。

4.1.1 全局锁

```
//要使用全局锁，则要执行这条命令：
flush tables with read lock
//如果要释放全局锁，则要执行这条命令,当然，当会话断开了，全局锁会被自动释放。
unlock tables
```

执行后，**整个数据库就处于只读状态了**，这时其他线程执行以下操作，都会被阻塞：

- 对数据的增删改操作，比如 insert、delete、update等语句；
- 对表结构的更改操作，比如 alter table、drop table 等语句。

> 全局锁应用场景

主要应用于做**全库逻辑备份**

但是备份就会花费很多的时间，关键是备份期间，业务只能读数据，而不能更新数据，这样会造成业务停滞。

在备份数据库之前可以先开启事务，会先创建 Read View，然后整个备份事务执行期间都在用这个 Read View，在可重复读的隔离级别下，即使其他事务更新了表的数据，也不会影响备份数据库时的 Read View。

4.1.2 表级锁

1.表锁

```
//表级别的共享锁，也就是读锁；
lock tables t_student read;
//表级别的独占锁，也就是写锁；
lock tables t_stuent write;
//释放当前所有表锁
unlock tables
```

尽量避免在使用 InnoDB 引擎的表使用表锁，因为表锁的颗粒度太大，会影响并发性能，**InnoDB 牛逼的地方在于实现了颗粒度更细的行级锁。**

2.元数据锁（MDL）

- 对一张表进行 CRUD 操作时，加的是 **MDL 读锁**；
- 对一张表做结构变更操作的时候，加的是 **MDL 写锁**；

我们对数据库表进行操作时，会自动给这个表加上 MDL，且**MDL 是在事务提交后才会释放。**

3.意向锁

- 在使用 InnoDB 引擎的表里对某些记录加上「共享锁」之前，需要先在表级别加上一个**意向共享锁；**
- 在使用 InnoDB 引擎的表里对某些纪录加上「独占锁」之前，需要先在表级别加上一个**意向独占锁；**

当执行插入、更新、删除操作，需要先对表加上「意向独占锁」，然后对该记录加独占锁。普通快照读通过 MVCC 实现一致性读，是无锁的。而当前读可以对记录加共享锁和独占锁。

意向共享锁和意向独占锁是表级锁，**意向锁的目的是在占锁时快速判断表里是否有记录被加锁。**

4.AUTO-INC 锁

AUTO-INC 锁是特殊的表锁机制，锁**不是再一个事务提交后才释放，而是再执行完插入语句后就会立即释放。**

在插入数据时，会加一个表级别的 AUTO-INC 锁，然后为被 AUTO_INCREMENT 修饰的字段赋值递增的值，等插入语句执行完成后，才会把 AUTO-INC 锁释放掉。其他事务的如果加锁期间要向该表插入语句都会被阻塞，从而保证插入数据时，被 AUTO_INCREMENT 修饰的字段的值是连续递增的。

4.1.3 行级锁

InnoDB 引擎是支持行级锁的，而 MyISAM 引擎并不支持行级锁。

普通的 select 语句是不会对记录加锁的，因为它属于快照读。如果要在查询时对记录加行锁，可以使用下面这两个方式，这种查询会加锁的语句称为**锁定读/当前读。**

```
//对读取的记录加共享锁
select ... lock in share mode;

//对读取的记录加独占锁
select ... for update;
```

上面这两条语句必须在一个事务中，**因为当事务提交了，锁就会被释放**，所以在使用这两条语句的时候，要加上 **begin、start transaction 或者 set autocommit = 0。**

共享锁（S锁）满足读读共享，读写互斥。独占锁（X锁）满足写写互斥、读写互斥。

1.Record Lock

记录锁有共享锁和排他锁之分（S型/X型）。当一个事务对一个行记录进行修改（UPDATE、DELETE）或者当前读（select...for update），MySQL会自动对该行记录加排他锁（X）。如果**另一个事务**也尝试修改这条记录，会被阻塞并等待排他锁被释放。如果另一个事务只读取这条记录，则会加共享锁（S）

如果你想要在特定条件下手动加锁，可以使用`SELECT ... FOR UPDATE`或`SELECT ... LOCK IN SHARE MODE`语句。`FOR UPDATE`会加排他锁，而`LOCK IN SHARE MODE`会加共享锁。

例如，下面sql语句为user表中主键 id 为 1 的这条记录加上 X 型的记录锁

```
//update、delete、select..for update可能会加记录锁
update user set name = "zs" where id = 1;
delete from user where id = 1;
select * from user where id = 1 for update;
```

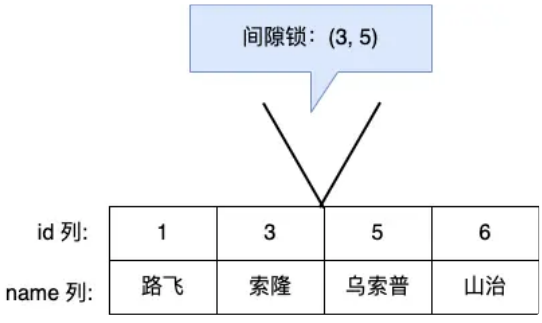
当事务执行 commit 后，事务过程中生成的锁都会被释放。

2.Gap Lock

```
//update、delete、select..for update可能会加间隙锁
update user set name = "zs" where id = 1;
delete from user where id = 1;
select * from user where id = 1 for update;
```

Gap Lock 称为**间隙锁**，只存在于可重复读隔离级别，目的是为了解决可重复读隔离级别下幻读的现象。可以阻塞其他事务的insert操作。

假设，表中有一个范围 id 为（3，5）间隙锁，防止其他事务在这个范围内插入数据，从而导致幻读。



间隙锁之间是兼容的，即两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系，因为间隙锁的目的是防止插入幻影记录而提出的。

3.Next-Key Lock

```
//update、delete、select..for update、可能会加临建锁
update user set name = "zs" where id = 1;
delete from user where id = 1;
select * from user where id = 1 for update;
```

Next-Key Lock 称为**临键锁**，是 Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。

next-key lock 是包含间隙锁+记录锁的，如果一个事务获取了 X 型的 next-key lock，那么另外一个事务在获取相同范围的 X 型的 next-key lock 时，是会被阻塞的。

4.插入意向锁

一个事务在插入一条记录的时候，需要判断插入位置是否已被其他事务加了间隙锁（next-key lock 也包含间隙锁）。

如果有的话，插入操作就会发生**阻塞**，直到拥有间隙锁的那个事务提交为止（释放间隙锁的时刻），在此期间会生成一个**插入意向锁**，表明有事务想在某个区间插入新记录，但是现在处于等待状态。

插入意向锁名字虽然有意向锁，但是它并**不是意向锁，它是一种特殊的间隙锁，属于行级别锁。**

4.2 行级锁的加锁规则

> 为什么要加行级锁（next-key lock、record lock、gap lock）？

防止出现不可重复读（第二次读记录被修改）和幻读（第二次读记录有新增或删除）的问题。

我们可以通过 `select * from performance_schema.data_locks\G;` 这条语句，查看事务执行 SQL 过程中加了什么锁。

加锁时不一定都是加临建锁，也有可能是加间隙锁或记录锁，要分情况讨论。

4.2.1 唯一索引

id:	1	5	10	15	20
name:	zs	ls	w	ww	www
age:	19	21	22	20	39

1.等值查询

记录存在的情况

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from user where id = 1 for update;
+----+--------+-----+
| id | name   | age |
+----+--------+-----+
|  1 | zs     |  19 |
+----+--------+-----+
1 row in set (0.02 sec)
```

临键锁会退化为索引1上的记录锁，因为是唯一索引，其他事务在插入id为1的记录时会失败，下一次执行该语句的结果不会变。所以不会产生幻读。也就不需要间隙锁，只要记录锁防止记录被修改。

记录不存在的情况

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

临建锁会退化为索引5上的间隙锁（1,5），因为锁是加在索引上的，而记录不存在，自然就不会被修改，故不需要记录锁。这次的案例是查询 id = 2 的记录，只要保证前后两次查询 id = 2 的结果集相同，因此只需要在 id = 5 加间隙锁就能避免幻读的问题了

2.范围查询

范围查询会根据扫描会对满足where的查询结果都加上锁

```
select * from user where id > 10 for update;
```

加锁情况为：索引15上的临建锁（10，15】，索引20上的临建锁（15，,20】，索引20上的临键锁（20，+∞）。

如果是id ≥ 10，还要防止id=10的记录被修改，要对唯一索引10加上记录锁

```
select * from user where id < 10 for update;
```

加锁情况为：索引10上的间隙锁（5，10），索引5上的临建锁（1，5】，索引20上的临键锁
（-∞，1）。

如果是id ≤ 10，索引10上的间隙锁（5，10)改为临建锁（5,10】

4.2.2 非唯一索引

age	19	20	21	22	39
id:	1	15	5	10	20
name:	a	aaa	a	a	aaa

在加锁时，会分别对这个非唯一索引和主键索引加锁。但是对主键索引加锁的时候，只有满足查询条件的记录才会对它们的主键索引加锁。

1.等值查询

记录不存在

```
select * from user where age = 25 for update;
```

扫描到39，记录不存在，无需记录锁。在索引39上加间隙锁（22,39），主键不加锁。

注意：插入数据时，不仅要看二级索引还要看主键。列如age:22，id:9插入成功，age:22,id:10插入被阻塞。

记录存在

```
select * from user where age = 22 for update;
```

为防止age=22的记录插入，加临建锁（21,22】，间隙锁（22,39），**主键10加记录锁。**

如果不加间隙锁（22,39）的话，age:22 id:11就可以插入，以致产生幻读。

2.范围查询

```
select * from user where age ⩾ 22  for update
```

加锁（21,22】（22，39】（39，+∞）主键10,20加记录锁

4.2.3 没有加索引的语句

如果锁定读查询语句或者增、删、改，没有使用索引列作为查询条件，或者查询语句没有走索引查询，导致扫描是全表扫描。那么，每一条记录的索引上都会加 next-key 锁，这样就相当于锁住的全表，这时如果其他事务对该表进行增、删、改操作的时候，都会被阻塞。

因此，**在线上在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引，如果是全表扫描的话，会对每一个索引加 next-key 锁，相当于把整个表锁住了**，这是挺严重的问题。

4.3 insert语句加锁过程

Insert 语句在正常执行时是不会生成锁结构的，它是靠聚簇索引记录自带的 trx_id 隐藏列来作为**隐式锁**来保护记录的。

> 什么是隐式锁?

当事务需要加锁的时，如果这个锁不可能发生冲突，InnoDB会跳过加锁环节，只有在可能发生冲突时才加锁，这种机制称为隐式锁。从而减少了锁的数量，提高了系统整体性能。

> insert语句加锁有三种情况:

- 记录之间有间隙锁，插入时**生成插入意向锁而被阻塞**
- Insert 的记录和已有记录存在唯一键冲突，此时也不能插入记录并生成锁。
- 事务A执行Insert语句后未提交，事务B也执行Insert语句插入同一个索引，此时**事务A的隐式锁转换为显示锁**，且锁的类型为x型记录锁，**事务B**因为发生唯一键冲突**生成S型的锁而阻塞。**

遇到唯一键冲突时，

- 如果主键索引重复，插入新记录的事务会给已存在的主键值重复的聚簇索引记录**添加 S 型记录锁。**
- 如果唯一二级索引重复，插入新记录的事务都会给已存在的二级索引列值重复的二级索引记录**添加 S 型 next-key 锁。**

4.4 死锁

id	order_no	create_date
1	1001	2021-12-28 13:59:07
2	1002	2021-12-28 13:59:14
3	1003	2021-12-28 13:59:21
4	1004	2021-12-28 13:59:30
5	1005	2021-12-28 13:59:36
6	1006	2021-12-28 14:24:33

事务 A	事务 B
begin;	begin;
//检查 1007 订单是否存在 select id from t_order where order_no = 1007 for update;	
	//检查 1008 订单是否存在 select id from t_order where order_no = 1008 for update;

//如果没有，则插入订单记录
insert into t_order (order_no, create_date) values (1007, now());
 	
阻塞等待....	

	//如果没有，则插入订单记录
	insert into t_order (order_no, create_date) values (1008, now());
	
	阻塞等待....

事务A对(1006, +∞)加间隙锁后，事务B也对(1006, +∞)加间隙锁，间隙锁是兼容的，多个事务可以持有同一个间隙锁，之后事务A插入1007订单被B事务的间隙锁阻塞，事务B插入1008订单被A事务的间隙锁阻塞，形成资源占有且互相等待的死锁局面。

> 死锁避免方法

死锁的四个必要条件：**互斥、占有且等待、不可抢占、循环等待。**只要系统发生死锁，这些条件必然成立，但是只要破坏任意一个条件就死锁就不会成立。

- **设置事务等待锁的超时时间。**当一个事务的等待时间超过该值后，就对这个事务进行回滚，于是锁就释放了，另一个事务就可以继续执行了。在 InnoDB 中，参数 `innodb_lock_wait_timeout` 是用来设置超时时间的，默认值时 50 秒。
- **开启主动死锁检测。**主动死锁检测在发现死锁后，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，表示开启这个逻辑，默认就开启。

五、日志篇

我们在执行执行一条“增删改”语句的时候，虽然没有输入 begin 开启事务和 commit 提交事务，但是 MySQL 会**隐式开启事务**来执行“增删改”语句的，执行完就自动提交事务的，这样就保证了执行完“增删改”语句后，我们可以及时在数据库表看到“增删改”的结果了。

5.1 undo log

undo log 两大作用：

- **实现事务回滚，保障事务的原子性。**事务处理过程中，如果出现了错误或者用户执行了 ROLLBACK 语句，MySQL 可以利用 undo log 中的历史数据将数据恢复到事务开始之前的状态。
- **实现 MVCC（多版本并发控制）关键因素之一。**MVCC 是通过 ReadView + undo log 实现的。undo log 为每条记录保存多份历史数据，MySQL 在执行快照读（普通 select 语句）的时候，会根据事务的 Read View 里的信息，顺着 undo log 的版本链找到满足其可见性的记录。

每当 InnoDB 引擎对一条记录进行操作（修改、删除、新增）时，要把回滚时需要的信息都记录到 undo log 里，**undo log 会写入 Buffer Pool 中的 Undo 页面。在发生回滚时，就读取 undo log 里的数据，然后做原先相反操作**

每一个记录中都有一个trx_id保存修改该记录的id，roll_pointer指针指向undo log链表串。

5.2 Buffer Pool

5.2.1 为什么要有Buffer Pool

MySQL 的数据都是存在磁盘中的，那么我们要更新一条记录的时候，得先要从磁盘读取该记录，然后在内存中修改这条记录，磁盘IO效率非常低

有了 Buffer Pool 后：

- **当读取数据时**，如果数据存在于 Buffer Pool 中，客户端就会**直接读取** Buffer Pool 中的数据，否则再去磁盘中读取。
- **当修改数据时**，如果数据存在于 Buffer Pool 中，那直接修改 Buffer Pool 中数据所在的页，**然后将其页设置为脏页**（该页的内存数据和磁盘上的数据已经不一致），为了减少磁盘I/O，不会立即将脏页写入磁盘，后续由后台线程选择一个合适的时机将脏页写入到磁盘。

> Buffer Pool 缓存什么？

在 MySQL 启动的时候，**InnoDB 会为 Buffer Pool 申请一片连续的内存空间（虚拟空间），然后按照默认的16KB的大小划分出一个个的页， Buffer Pool 中的页就叫做缓存页。**

Buffer Pool 除了缓存「索引页」和「数据页」，还包括了 Undo 页，插入缓存、自适应哈希索引、锁信息等等。

5.2.2 如何管理Buffer Pool

1.如何管理空闲页

为了更好的管理这些在 Buffer Pool 中的缓存页，**InnoDB 为每一个缓存页都创建了一个控制块**，控制块信息包括「缓存页的表空间、页号、缓存页地址、链表节点」等等。

为了能够快速找到空闲的缓存页，可以使用链表结构，将空闲缓存页的「控制块」作为链表的节点，这个链表称为 **Free 链表**（空闲链表）

有了 Free 链表后，**每当需要从磁盘中加载一个页到 Buffer Pool 中时，就从 Free 链表中取一个空闲的缓存页，并且把该缓存页对应的控制块的信息填上，然后把该缓存页对应的控制块从 Free 链表中移除。**

2.如何管理脏页

更新数据的时候，不需要每次都要写入磁盘，而是将 Buffer Pool 对应的缓存页标记为**脏页**，**然后再由后台线程将脏页写入到磁盘。**

那为了能快速知道哪些缓存页是脏的，于是就设计出 **Flush 链表**，它跟 Free 链表类似的，链表的节点也是控制块，区别在于 Flush 链表的元素都是脏页。

有了 Flush 链表后，后台线程就可以遍历 Flush 链表，将脏页写入到磁盘。

3.如何提高缓存命中率

> 预读失效

MySQL 在加载数据页时，会提前把它相邻的数据页一并加载进来，目的是为了减少磁盘 IO。

但是可能这些**被提前加载进来的数据页，并没有被访问**，相当于这个预读是白做了，这个就是**预读失效**。

> 预读失效解决方案

划分这两个区域后，预读的页就只需要加入到 old 区域的头部，当页被真正访问的时候，才将页插入 young 区域的头部。如果预读的页一直没有被访问，就会从 old 区域移除，这样就不会影响 young 区域中的热点数据。

> Buffer Poll污染

当某一个 SQL 语句**扫描了大量的数据**时，在 Buffer Pool 空间比较有限的情况下，可能会将 **Buffer Pool 里的所有页都替换出去，导致大量热数据被淘汰了**，等这些热数据又被再次访问的时候，由于缓存未命中，就会产生大量的磁盘 IO，MySQL 性能就会急剧下降，这个过程被称为 **Buffer Pool污染。**

> Buffer Poll污染解决方案

进入到 young 区域条件增加了一个**停留在 old 区域的时间判断。**在对某个处在 old 区域的缓存页进行第一次访问时，就在它对应的控制块中记录下来这个第一次访问时间：

- 如果后续的访问时间与第一次访问的时间**在某个时间间隔内**，那么**该缓存页就不会被从 old 区域移动到 young 区域的头部**；
- 如果后续的访问时间与第一次访问的时间**不在某个时间间隔内**，那么**该缓存页移动到 young 区域的头部**；

短时间内被大量加载的页，并不会立刻插入新生代头部，而是优先淘汰那些，短期内仅仅访问了一次的页。

5.2.3 脏页刷盘时机

可能大家担心，如果在脏页还没有来得及刷入到磁盘时，MySQL 宕机了，不就丢失数据了吗?

这个不用担心，InnoDB 的更新操作采用的是 Write Ahead Log 策略，即先写日志，再写入磁盘，通过 redo log 日志让 MySQL 拥有了崩溃恢复能力。

下面几种情况会触发脏页的刷新：

- 当 redo log 日志满了的情况下，会主动触发脏页刷新到磁盘；

- Buffer Pool 空间不足时，需要将一部分数据页淘汰掉，如果淘汰的是脏页，需要先将脏页同步到磁盘；
- MySQL 认为空闲时，后台线程会定期将适量的脏页刷入到磁盘；
- MySQL 正常关闭之前，会把所有的脏页刷入到磁盘；

5.3 redo log

Buffer Pool在内存中，而内存总是不可靠，万一断电重启，还没来得及落盘的脏页数据就会丢失。而先写redo log到磁盘就可以避免这个问题，从而实现数据的持久化

为了防止断电导致数据丢失的问题，当有一条记录需要更新的时候，InnoDB 引擎就会先更新内存（同时标记为脏页），然后将本次对这个页的修改**以 redo log 的形式记录下来**（对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了AAA 更新），这个时候更新就算完成了。在事务提交时，只要**先将 redo log 持久化到磁盘即可，可以不需要等到将缓存在 Buffer Pool 里的脏页数据持久化到磁盘**。后续，InnoDB 引擎会在适当的时候，由后台线程将缓存在 Buffer Pool 的脏页（数据页、undo页等）刷新到磁盘里，这就是 WAL （Write-Ahead Logging）技术。**WAL技术指的是，MySQL的写操作并不是立刻写到磁盘上，而是先写redo log日志，然后在合适的时间再写数据到磁盘上。**

> redo log 和 undo log 区别在哪？

这两种日志是属于 InnoDB 存储引擎的日志，它们的区别在于：

- redo log 记录了此次事务「**完成后**」的数据状态，记录的是更新**之后**的值；
- undo log 记录了此次事务「**开始前**」的数据状态，记录的是更新**之前**的值；

事务提交之前发生了崩溃，重启后会通过 undo log 回滚事务，事务提交之后发生了崩溃，重启后会通过 redo log 恢复事务

> redo log 要写到磁盘，数据也要写磁盘，为什么要多此一举？

写入 redo log 的方式使用了追加操作， 所以磁盘操作是**顺序写**，而写入数据需要先找到写入位置，然后才写到磁盘，所以磁盘操作是**随机写**。

磁盘的「顺序写 」比「随机写」 高效的多，因此 redo log 写入磁盘的开销更小。

redo log刷盘时机

执行一个事务的过程中，产生的 redo log 也不是直接写入磁盘的，因为这样会产生大量的 I/O 操作，而且磁盘的运行速度远慢于内存。

所以，redo log 也有自己的缓存—— redo log buffer，**每当产生一条 redo log 时，会先写入到 redo log buffer，后续在持久化到磁盘**

它什么时候刷新到磁盘？主要有下面几个时机：

- MySQL 正常关闭时；

- 当 redo log buffer 中记录的写入量大于 redo log buffer 内存空间的一半时，会触发落盘；
- InnoDB 的后台线程每隔 1 秒，将 redo log buffer 持久化到磁盘。
- 每次事务提交时都将缓存在 redo log buffer 里的 redo log 直接持久化到磁盘（这个策略可由 innodb_flush_log_at_trx_commit 参数控制，下面会说）。

由参数 innodb_flush_log_at_trx_commit 参数控制，可取的值有：0、1、2，默认值为 1，这三个值分别代表的策略如下：

- 当设置该**参数为 0 时**，表示每次事务提交时 ，还是**将 redo log 留在 redo log buffer 中** ，该模式下在事务提交时不会主动触发写入磁盘的操作。
- 当设置该**参数为 1 时**，表示每次事务提交时，都**将缓存在 redo log buffer 里的 redo log 直接持久化到磁盘**，这样可以保证 MySQL 异常重启之后数据不会丢失。
- 当设置该**参数为 2 时**，表示每次事务提交时，都只是缓存在 redo log buffer 里的 redo log **写到 redo log 文件，注意写入到「 redo log 文件」并不意味着写入到了磁盘**，因为操作系统的文件系统中有个 Page Cache

InnoDB 的后台线程每隔 1 秒：

- 针对参数 0 ：会把缓存在 redo log buffer 中的 redo log ，通过调用 write() 写到操作系统的 Page Cache，然后调用 fsync() 持久化到磁盘。**所以参数为 0 的策略，MySQL 进程的崩溃会导致上一秒钟所有事务数据的丢失；**
- 针对参数 2 ：调用 fsync，将缓存在操作系统中 Page Cache 里的 redo log 持久化到磁盘。**所以参数为 2 的策略，较取值为 0 情况下更安全，因为 MySQL 进程的崩溃并不会丢失数据，只有在操作系统崩溃或者系统断电的情况下，上一秒钟所有事务数据才可能丢失。**

redo log 文件写满了怎么办?

默认情况下， InnoDB 存储引擎有 1 个重做日志文件组(redo log Group)，「重做日志文件组」由有 2 个 redo log 文件组成，这两个 redo 日志的文件名叫 ：ib_logfile0 和 ib_logfile1 。

重做日志组以**循环写**的方式工作，InnoDB 存储引擎会先写 ib_logfile0 文件，当 ib_logfile0 文件被写满的时候，会切换至 ib_logfile1 文件，当 ib_logfile1 文件也被写满时，会切换回 ib_logfile0 文件。

5.4 binlog

MySQL在完成一条更新操作后，Server层会生成一条binlog，**binlog 文件是记录了所有数据库表结构变更和表数据修改的日志**，等之后事务提交的时候，会将该事物执行过程中产生的所有 binlog 统一写 入 binlog 文件。

5.4.1 redo log与binlog的区别

1、适用对象不同：

- binlog 是 MySQL 的 Server 层实现的日志，所有存储引擎都可以使用；
- redo log 是 Innodb 存储引擎实现的日志；

2、文件格式不同：

- binlog 有 3 种格式类型，分别是 STATEMENT（默认格式）、ROW、 MIXED，区别如下：
  - STATEMENT：每一条修改数据的 SQL 都会被记录到 binlog 中（相当于记录了逻辑操作，所以针对这种格式， binlog 可以称为逻辑日志），主从复制中 slave 端再根据 SQL 语句重现。但 STATEMENT 有动态函数的问题，比如你用了 uuid 或者 now 这些函数，你在主库上执行的结果并不是你在从库执行的结果，这种随时在变的函数会导致复制的数据不一致；
  - ROW：记录行数据最终被修改成什么样了（这种格式的日志，就不能称为逻辑日志了），不会出现 STATEMENT 下动态函数的问题。但 ROW 的缺点是每行数据的变化结果都会被记录，比如执行批量 update 语句，更新多少行数据就会产生多少条记录，使 binlog 文件过大，而在 STATEMENT 格式下只会记录一个 update 语句而已；
  - MIXED：包含了 STATEMENT 和 ROW 模式，它会根据不同的情况自动使用 ROW 模式和 STATEMENT 模式；
- redo log 是物理日志，记录的是在某个数据页做了什么修改，比如对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了AAA 更新；

3、写入方式不同：

- binlog 是**追加写**，写满一个文件，就创建一个新的文件继续写，不会覆盖以前的日志，保存的是全量的日志。
- redo log 是**循环写**，日志空间大小是固定，全部写满就从头开始，保存未被刷入磁盘的脏页日志。

4、用途不同：

- binlog 用于备份恢复、主从复制；
- redo log 用于掉电等故障恢复。

5.4.2 主从复制

MySQL集群的主从复制过程梳理成 3 个阶段：

- **写入 Binlog**：主库写 binlog 日志，提交事务，并更新本地存储数据。
- **同步 Binlog**：把 binlog 复制到所有从库上，每个从库把 binlog 写到暂存日志中。
- **回放 Binlog**：回放 binlog，并更新存储引擎中的数据。

具体详细过程如下：

- MySQL 主库在收到客户端提交事务的请求之后，会先写入 binlog，再提交事务，更新存储引擎中的数据，事务提交完成后，返回给客户端“操作成功”的响应。
- 从库会创建一个专门的 I/O 线程，连接主库的 log dump 线程，来接收主库的 binlog 日志，再把 binlog 信息写入 relay log 的中继日志里，再返回给主库“复制成功”的响应。
- 从库会创建一个用于回放 binlog 的线程，去读 relay log 中继日志，然后回放 binlog 更新存储引擎中的数据，最终实现主从的数据一致性。

在完成主从复制之后，你就可以在写数据时只写主库，在读数据时只读从库，这样即使写请求会锁表或者锁记录，也不会影响读请求的执行。

5.4.3 binlog什么时候刷盘

事务执行过程中，先把日志写到binlog cache（Server 层的 cache），事务提交的时候，再把 binlog cache 写到 binlog 文件中。

5.4.4 两阶段提交

在持久化 redo log 和 binlog 这两份日志的时候，如果出现半成功的状态，就会造成主从环境的数据不一致性。这是因为 redo log 影响主库的数据，binlog 影响从库的数据，所以 redo log 和 binlog 必须保持一致才能保证主从数据一致。

MySQL 为了避免出现两份日志之间的逻辑不一致的问题，使用了「两阶段提交」来解决，两阶段提交其实是分布式事务一致性协议，它可以保证多个逻辑操作要不全部成功，要不全部失败，不会出现半成功的状态。

两阶段提交把单个事务的提交拆分成了 2 个阶段，分别是「准备（Prepare）阶段」和「提交（Commit）阶段」，每个阶段都由协调者（Coordinator）和参与者（Participant）共同完成。注意，不要把提交（Commit）阶段和 commit 语句混淆了，commit 语句执行的时候，会包含提交（Commit）阶段。

- **prepare 阶段**：将 XID（内部 XA 事务的 ID） 写入到 redo log，同时将 redo log 对应的事务状态设置为 prepare，然后将 redo log 持久化到磁盘（innodb_flush_log_at_trx_commit = 1 的作用）；
- **commit 阶段**：把 XID 写入到 binlog，然后将 binlog 持久化到磁盘（sync_binlog = 1 的作用），接着调用引擎的提交事务接口，将 redo log 状态设置为 commit，此时该状态并不需要持久化到磁盘，只需要 write 到文件系统的 page cache 中就够了，因为只要 binlog 写磁盘成功，就算 redo log 的状态还是 prepare 也没有关系，一样会被认为事务已经执行成功；

5.5 总结

具体更新一条记录 `UPDATE t_user SET name = 'xiaolin' WHERE id = 1;` 的流程如下：

1. 执行器负责具体执行，会调用存储引擎的接口，通过主键索引树搜索获取 id = 1 这一行记录：

    - 如果 id=1 这一行所在的数据页本来就在 buffer pool 中，就直接返回给执行器更新；
    - 如果记录不在 buffer pool，将数据页从磁盘读入到 buffer pool，返回记录给执行器。
2. 执行器得到聚簇索引记录后，会看一下更新前的记录和更新后的记录是否一样：
    - 如果一样的话就不进行后续更新流程；
    - 如果不一样的话就把更新前的记录和更新后的记录都当作参数传给 InnoDB 层，让 InnoDB 真正的执行更新记录的操作；
3. 开启事务， InnoDB 层更新记录前，首先要记录相应的 undo log，因为这是更新操作，需要把被更新的列的旧值记下来，也就是要生成一条 undo log，undo log 会写入 Buffer Pool 中的 Undo 页面，不过在内存修改该 Undo 页面后，需要记录对应的 redo log。
4. InnoDB 层开始更新记录，会先更新内存（同时标记为脏页），然后将记录写到 redo log 里面，这个时候更新就算完成了。为了减少磁盘I/O，不会立即将脏页写入磁盘，后续由后台线程选择一个合适的时机将脏页写入到磁盘。这就是 **WAL 技术**，MySQL 的写操作并不是立刻写到磁盘上，而是先写 redo 日志，然后在合适的时间再将修改的行数据写到磁盘上。
5. 至此，一条记录更新完了。
6. 在一条更新语句执行完成后，然后开始记录该语句对应的 binlog，此时记录的 binlog 会被保存到 binlog cache，并没有刷新到硬盘上的 binlog 文件，在事务提交时才会统一将该事务运行过程中的所有 binlog 刷新到硬盘。
7. 事务提交（为了方便说明，这里不说组提交的过程，只说两阶段提交）：
    - **prepare 阶段**：将 redo log 对应的事务状态设置为 prepare，然后将 redo log 刷新到硬盘；
    - **commit 阶段**：将 binlog 刷新到磁盘，接着调用引擎的提交事务接口，将 redo log 状态设置为 commit（将事务设置为 commit 状态后，刷入到磁盘 redo log 文件）；
8. 至此，一条更新语句执行完成。