**3-Mysql**

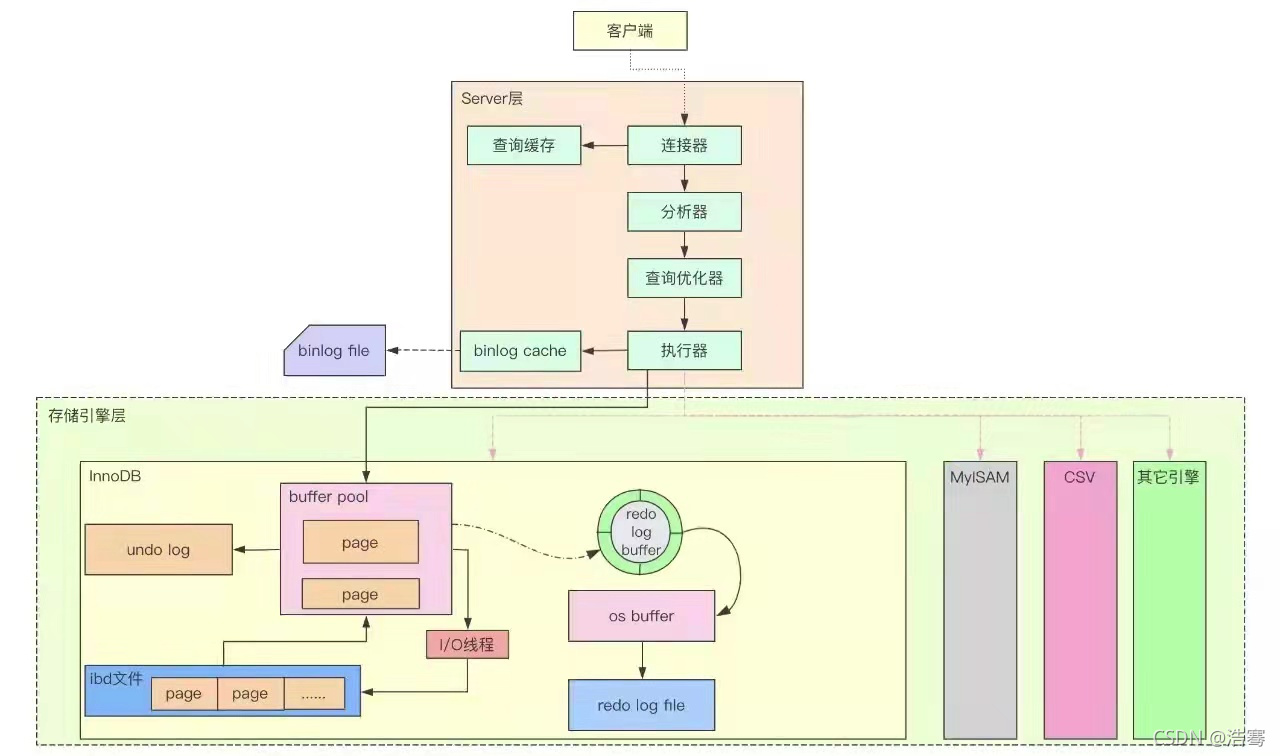
**mysql架构**

**逻辑架构**

* **连接层：**最上层是一些客户端和连接服务。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于 SSL 的安全链接。服务器也会为安全接入的每个客户端验证它所具有的操作权限。
* **服务层：**第二层服务层，主要完成大部分的核心服务功能， 包括查询解析、分析、优化、缓存、以及所有的内置函数，所有跨存储引擎的功能也都在这一层实现，包括触发器、存储过程、视图等。
* **引擎层：**第三层存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API与存储引擎进行通信。不同的存储引擎具有的功能不同，这样我们可以根据自己的实际需要进行选取
* **存储层：**第四层为数据存储层，主要是将数据存储在运行于该设备的文件系统之上，并完成与存储引擎的交互。

**物理架构**

* **数据存储**：实际存放数据库文件的地方，通常涉及到磁盘存储的组织和管理。
* **内存管理：**MySQL使用缓存来提高性能，例如查询缓存（虽然在MySQL 8.0中已被移除）和InnoDB的缓冲池。
* **网络通信：**客户端与MySQL服务器之间的通信机制，包括认证、安全措施如SSL连接，以及数据传输。
* **并发控制：**包括线程管理和锁定机制，确保多用户同时访问时的数据处理一致性和效率。
* **备份恢复：**物理备份通常涉及到数据的复制和恢复，保证数据的安全性和完整性

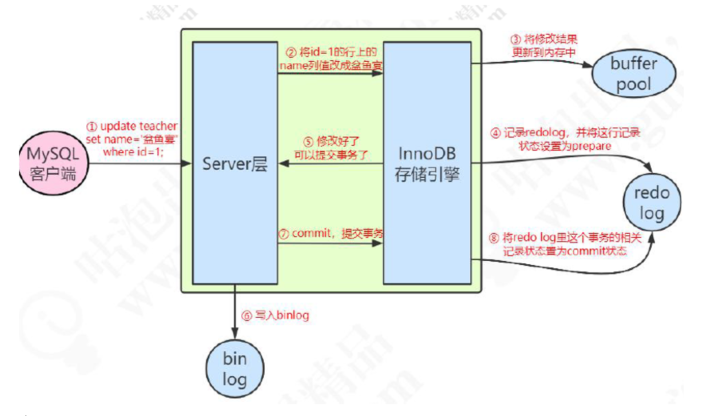


**查询语句执行流程**

* 连接器（验证用户身份，给予权限）
* 查询缓存（存在缓存则直接返回，不存在则执行后续操作） 默认关闭
* 分析器（对SQL进行词法分析和语法分析操作）
* 优化器（主要对执行的sql优化选择最优的执行方案方法）
* 查询解析：当用户提交一个查询请求时，优化器首先将查询语句进行解析，识别出关键字、表名、条件等元素。
* 查询重写：在解析之后，查询优化器可能会对查询进行重写，改进语句结构以提升性能，例如选择最佳的连接方法或利用索引加速查询。
* 查询优化：此阶段评估不同的执行计划，选择成本最低的方案。优化器会考虑表的统计信息、索引信息和系统配置等多种因素，并运用多种算法，如成本估算和连接选择。
* 执行计划生成：确定最优的执行计划后，优化器会产生详细的执行步骤供MySQL引擎执行。执行计划定义了查询的具体执行顺序和方法。
* 优化方法应用：优化器可能采用多种策略来提升查询效率，比如索引优化、表分区、查询重写、数据预取以及查询缓存等技术。
* 执行器（执行时会先看用户是否有执行权限，有才去使用这个引擎提供的接口）
* MySQL的执行计划是优化器根据SQL语句、表结构、索引、查询条件等信息生成的查询执行步骤，可以使用EXPLAIN关键字来获取。
* 去引擎层获取数据返回（如果开启查询缓存则会缓存查询结果）

**一个SQL更新操作的完整过程：**

1. 事务开始后，数据进行更新操作，加载缓存数据，如果有的话就从缓存中获取，没有的话需要从磁盘上拉取，放到buffer pool中；
2. 数据的旧值写入到undo log，方便回滚。
3. 更新数据页里的数据记录，此时数据页面变为脏页，数据页在buffer pool里，buffer pool 中的页可以区分为freelist，lrulist, flushlist，当从磁盘中加载一个数据的时候，需要从freelist中申请一个页放在lrulist中，当页被更新后，也会被加到flushlist。lrulist的长度有限，新的访问的页会放在前面，后面的页会慢慢被淘汰，如果淘汰的页面正好是脏页的话需要把脏页的内容同步到磁盘上。
4. 然后把数据的更新记录在redo log buffer上，redo log bufer也不能每次都直接flush到磁盘，那样效率太低，在内存中有个redo log buffer，会存储redo log。可通过三种方式设置redo log buffer的内容同步到磁盘的redo log的方式，一种是每隔一秒一次；第二种是每次事务在commit时候，会把redo log buffer的内容更新到系统的缓存，系统缓存会定时把内容sync到磁盘；第三种是每次提交时候都把他同步到磁盘。默认是第二种。buffer pool中的脏页会在一定是时机同步到磁盘：
5. 内存数据页的更新内容记录在 redolog buffer中，此时，buffer中的这条语句状态为prepare。然后告知执行器执行完成了，随时可以提交事务。
6. server层提交事务时，会先将这个操作的日志写入binlog buffer中，
7. 再调用引擎的事务提交接口，引擎会将刚写入的redolog记录状态修改为commit。更新完成。
8. 数据刷新到磁盘。定期或是bufferpool满了。或是服务down机。



**MYSQL 数据类型**

MySQL支持多种数据类型，用于存储不同类型的数据。具体如下：

**整数类型**

**TINYINT:** 8位，范围从-128到127或0到255。

**SMALLINT:** 16位，范围从-32,768到32,767或0到65,535。

**MEDIUMINT:** 24位，范围从-8,388,608到8,388,607或0到16,777,215。

**INT 或 INTEGER:** 32位，范围从-2,147,483,648到2,147,483,647或0到4,294,967,295。

**BIGINT:** 64位，范围从-9,223,372,036,854,775,808到9,223,372,036,854,775,807或0到18,446,744,073,709,551,615。

**浮点数类型**

**FLOAT:** 单精度浮点数。

**DOUBLE:** 双精度浮点数。

**DECIMAL:** 用于精确计算的固定精度数，适合财务计算。

**字符串类型**

**CHAR:** 定长字符串，长度固定。

**VARCHAR:** 变长字符串，长度可变。

**TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT:** 不同长度的文本数据。

**BINARY、VARBINARY:** 用于存储二进制数据。

**TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB:** 用于存储二进制大对象数据。

**日期和时间类型**

**DATE:** 日期。

**TIME:** 时间。

**DATETIME:** 日期和时间。

**TIMESTAMP:** 时间戳，记录了创建时间和更新时间。

**YEAR:** 年份。

**MySQL高级特性**

**图（Views）**

* 视图是一个虚拟表，它是根据SQL查询结果定义的，并提供了查看数据库数据的另一种方式。
* 使用视图可以简化复杂的SQL操作，提高数据安全性，通过限制对特定数据的访问来保护敏感信息。
* 视图使得数据处理更加方便，特别是在需要经常执行相同查询的情况下。

**触发器（Triggers）**

* 触发器是与表相关联的特殊类型的存储过程，它在插入、更新或删除表的数据时自动执行。
* 触发器可以用来实施复杂的业务规则，如自动维护数据完整性、审计日志记录等。
* 在MySQL中，触发器可以在BEFORE或AFTER触发事件的指定时机进行操作。

**存储过程（Stored Procedures）**

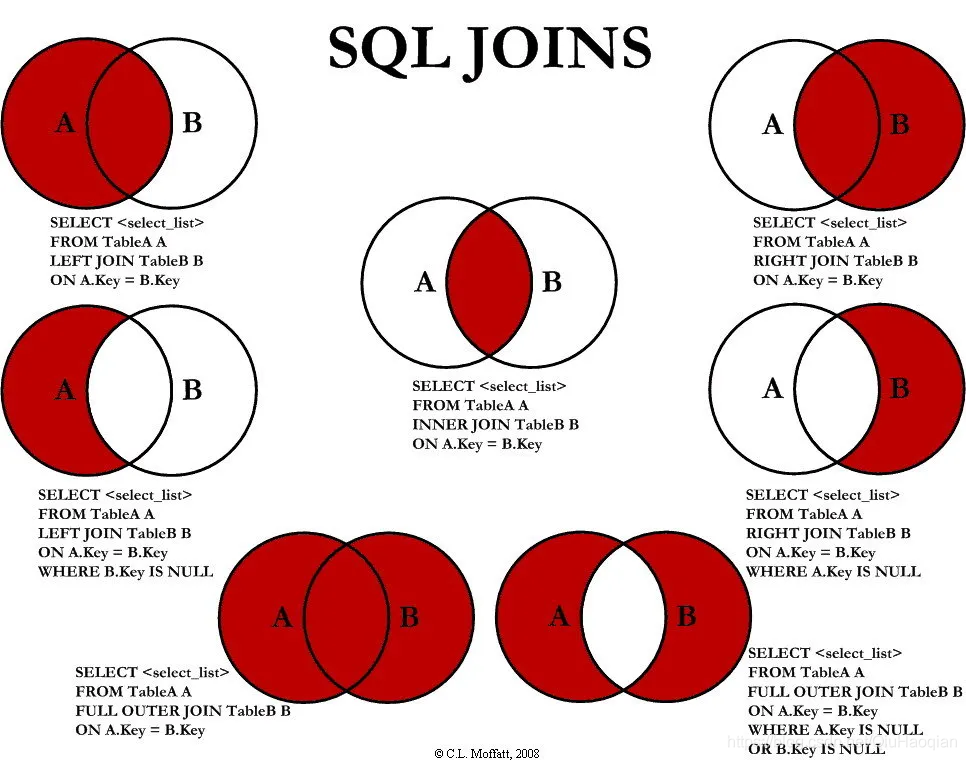
* 存储过程是一组预先编译的SQL语句，它们被命名并存储在数据库中，以便重复使用。
* 存储过程可以接受参数，这使得它们非常灵活，能够处理不同的业务场景。
* 使用存储过程可以减少网络流量，提高应用程序的性能。

**函数（Functions）**

* 函数类似于存储过程，但它必须返回一个值。
* 函数通常用于执行计算并返回结果，可以在SQL语句中调用。
* 函数的使用可以提高代码的重用性和模块化。

**JOIN**

**内连接、外连接、交叉连接、笛卡尔积**



**存储引擎**

存储引擎主要分为 InnoDB 与 MyISAM

|  |  |  |
| --- | --- | --- |
| **对比项** | **MyISAM** | **InnoDB** |
| 主外键 | 不支持 | 支持 |
| 事务 | 不支持 | 支持 |
| 行表锁 | 表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作 | 行锁,操作时只锁某一行，不对其它行有影响，适合高并发的操作 |
| 缓存 | 只缓存索引，不缓存真实数据 (**非聚集索引**) | 不仅缓存索引还要缓存真实数据(**聚集索引**)，对内存要求较高，而且内存大小对性能有决定性的影响 |
| 表空间 | 小 | 大 |
| 关注点 | 性能 | 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
* 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
* InnoDB是聚簇索引，MyISAM 是非聚簇索引。聚簇索引的文件存放在主键索引的叶子节点上，因此 InnoDB 必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而 MyISAM 是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的
* InnoDB 不保存表的具体行数，执行select count(\*) from table 时需要全表扫描。而 MyISAM 用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快
* 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

|  |
| --- |
| 一张表，里面有ID自增主键，当 insert 了17条记录之后，删除了第15,16,17条记录，再把 Mysql 重启，再insert 一条记录，这条记录的ID是18还是15 ？   * 如果表的类型是 MyISAM，那么是18。因为MyISAM表会把自增主键的最大 ID 记录到数据文件中，重启MySQL自增主键的最大ID也不会丢失； * 如果表的类型是InnoDB，那么是15。因为InnoDB 表只是把自增主键的最大ID记录到内存中，所以重启数据库或对表进行OPTION操作，都会导致最大ID丢失。 |

**MYSQL 缓存机制**

**查询缓存（Query Cache）**

查询缓存是MySQL中用来缓存查询结果的一种机制，它存储了客户端发送的SELECT查询语句及其对应的完整结果集。当同样的查询再次发生时，如果能够在缓存中找到对应的结果，则直接返回缓存中的数据，从而避免了重复解析和执行相同的查询。

查询缓存的工作流程大致为：当MySQL接收到客户端的查询请求并完成权限验证后，会先检查查询缓存中是否有匹配的结果集，如果找到则直接返回；如果没有找到，则会进行SQL解析、优化和执行，然后将结果存入查询缓存中供后续使用。

需要注意的是，从MySQL 5.7开始，官方已经移除了查询缓存的功能，因为它在某些场景下可能导致性能下降或者维护困难。

**InnoDB缓冲池（Buffer Pool）**

InnoDB缓冲池是InnoDB存储引擎用来缓存数据页和索引页的内存区域。它是以页为单位来管理的，主要缓存最频繁访问（“hot”）的数据页，以提高数据库的读取效率。

缓冲池的大小对数据库的性能有重要影响，合理配置可以显著减少磁盘I/O操作，提升数据库的响应速度。

**表缓存（Table Cache）**

表缓存是MySQL用来缓存打开的表信息的区域，包括表的结构、索引等信息。当一个表被多次查询时，表的定义信息会被加载到表缓存中，这样可以减少文件系统的访问次数，加快表的打开速度。

然而，表缓存同样在MySQL 5.6之后的版本中被移除，因为现代操作系统通常已经提供了足够的表描述符缓存，MySQL自行管理这部分缓存的优势不再明显。

**索引**

Mysql 中主要使用 B+树索引，InnoDB 使用了聚簇索引，MyISAM 使用了非聚簇索引

**索引优缺点**

**索引优势**

* 提高数据检索效率，降低数据库IO成本；
* 降低数据排序的成本，降低CPU的消耗；

**索引劣势**

* 索引也是一张表，保存了主键和索引字段，并指向实体表的记录，所以也需要占用内存。
* 虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。

**为什么不用别的数据结构作为索引**

* **数组和链表**的缺点就是数据量大的时候存不了，也就是说不合适大数据量。
* **哈希**是通过hash函数计算出一个hash值的，存在哈希碰撞的情况，另外哈希也不支持部分索引查询以及范围查找。但是哈希的优点就是查找的时间复杂度是O(1)，那么什么情况下可以使用hash索引呢？就是查询条件不会变，而且没有部分查询和范围查询的时候。
* **红黑树**存储的数据量大的时候，红黑树的节点层数多，也就是树的高度比较高，查找的底层数据时，查找次数就比较多，即对磁盘IO使用比较频繁。总结为以下两点：
* **读取浪费太多**：通过计算本来树的每一层大概需要分配**16KB**的数据，但是对于红黑树来说，实际存的节点数比较少，即存的数据大小远远小于**16KB**，从而造成存储空间的浪费
* **读取磁盘的次数过多**：树的层数越多，查找数据时读取磁盘的次数也就越多
* B树是一种自平衡的树数据结构，它能够保持数据有序，使得查找、插入、删除等操作都能在对数时间内完成。
* **因为BTree不适合范围查找**。就拿上面的来举例，比如我要查找小于6的数据，则先找到6的节点，然后需要**遍历**一遍6节点(索引)的左子树，不遍历的话，就拿不到小于6的这些数据了，也就说索引失效了，所以说不适合范围查找。
* BTree的节点除了存储**索引**之外，还存储了**数据**本身，占用空间较大，但是磁盘的页大小是有限的(16KB左右)，因此，存储同样大小的数据，BTree显得比较高(相对B+Tree)，稳定性弱一些。

**B+ 树作为索引的优势**

* 叶子节点连起来了，是一条有序的双向链表，目的是为了解决范围查找。比如需要查找小于9的数据，只要找到等于9的数据，然后将9的左边数据全部拿出来即可。
* 非叶子节点不存数据，只存索引，空间利用更高效。
* 数据的个数和节点一样多，换句话说，非叶子节点存的是其子树的最大或最小值。

**计算m阶，即B+Tree该取多少合适**

m是怎么计算出来的呢？是根据磁盘的页大小来计算的，也就是说是由磁盘页大小决定的。

我们知道，磁盘的页大小大概是16K，MySql创建索引时，可以根据字段及类型来计算磁盘一页大概可以存多少数据。

根据官方文档描述，树高度等于2时(2阶)，大概可以存两万多条数据；高度等于3时(3阶)，大概可以存两千多万条数据，怎么计算的呢？  
 首先，1 千字节(KB)=1024 字节(B)，一页有16KB，假设存**主键+指针**大概有8+6=14B，则一页就可以存：16\*1024/(8+6)=**1170** 个索引了。

对于3阶的B+Tree来说，大概可以存：应该是1170^2\*16 约等于2000w，这里的16是叶子结点按照一行数据1KB计算的

**MySQL索引类型**

* **B+树索引：**这是MySQL中使用最广泛的索引类型，它适用于全范围的查询操作，如等于、大于、小于等。B+树索引能够提供有序的数据访问，这意味着可以快速进行范围查找。它分为几种不同的形式：
* **普通索引：**最基本的B+树索引，没有任何限制条件，允许数据表中存在重复值。
* **唯一索引：**确保索引列中的所有值都是唯一的，这对于防止数据重复很有用。
* **主键索引：**特殊的唯一索引，不允许有空值，每个表只能有一个主键索引。
* **哈希索引：**基于哈希表实现，它的优点是查找速度非常快，因为它可以通过一次哈希算法直接定位到数据的存储位置。然而，哈希索引不支持范围查询和排序操作，只适用于等值查询的场景。
* **全文索引：**专为全文搜索优化的索引类型，主要用于对大文本字段的内容进行高效检索。全文索引可以支持自然语言搜索和布尔模式搜索，通常用于文章、报告等内容较多的文本字段。
* **空间索引：**用于地理空间数据类型的索引，支持空间数据的快速查询，如点、线、多边形等地理位置信息

**推荐使用整型自增主键 而不是选择UUID？**

* UUID是字符串，比整型消耗更多的存储空间；
* 在B+树中进行查找时需要跟经过的节点值比较大小，整型数据的比较运算比字符串更快速；
* 自增的整型索引在磁盘中会连续存储，在读取一页数据时也是连续；UUID是随机产生的，读取的上下两行数据存储是分散的，不适合执行 类似 where id > 5 && id < 20 这样的条件查询语句。
* 在插入或删除数据时，整型自增主键会在叶子结点的末尾建立新的叶子节点，不会破坏左侧子树的结构；UUID主键是无序的，B+树为了维持自身的特性，有可能会进行结构的重构，消耗更多的时间。

**为什么非主键索引结构叶子节点存储的是主键值？**

为了保证 **数据一致性** 和 **节省存储空间** 例如，

商城系统订单表会存储一个用户ID作为关联外键，而不推荐存储完整的用户信息，因为当我们用户表中的信息（真实名称、手机号、收货地址···）修改后，不需要再次维护订单表的用户数据，同时也节省了存储空间。

**为什么Mysql索引要用B+树不是B树 或 红黑树？**

* 用 B+ 树不用 B树 考虑的是IO对性能的影响：B树在id递增的情况下会单边增长，形成链表的结构
* B树 的每个节点都存储数据，而 B+树 只有叶子节点才存储数据，所以查找相同数据量的情况下，B树的高度更高，IO更频繁。
* 数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页（对应索引树的节点）。
* 同理，红黑树虽然是具有左旋功能的平衡二叉树，但是其深度在数据量大时深度依然很深

**为何不采用Hash方式？**

* Hash索引底层用的 哈希表是一种以 key-value 存储数据的结构，用一个哈希函数把key换算成一个确定的位置，然后把value放在数组的这个位置。所以多个数据在存储关系上是完全没有任何顺序关系的，所以，对于区间查询是无法直接通过索引查询的，就需要全表扫描。所以，哈希索引只适用于等值查询的场景。
* B+ Tree是一种多路平衡查询树，所以他的节点是天然有序的（左子节点小于父节点、父节点小于右子节点），而且叶子结点还用双向链表对节点进行连接，所以对于范围查询的时候不需要做全表扫描。
* 哈希索引不支持多列联合索引的最左匹配规则，如果有大量重复键值得情况下，哈希索引的效率会很低，因为存在哈希碰撞问题。

**哪些情况需要创建索引？**

1. 主键自动建立唯一索引；
2. 频繁作为查询条件的字段；
3. 查询中与其他表关联的字段，外键关系建立索引；
4. 单键/组合索引的选择问题，高并发下倾向创建组合索引；
5. 查询中排序的字段，排序字段通过索引访问大幅提高排序速度；
6. 查询中统计或分组字段；

**索引失效场景**

* **组合索引未使用最左前缀**，例如组合索引（A，B），where B=b不会使用索引；
* **like未使用最左前缀**，where A like '%China'；
* **搜索一个索引而在另一个索引上做order by**，where A=a order by B，（一个失败一个成功）只使用A上的索引，因为查询只使用一个索引 ；
* **or 可能使第一个索引失效。如果查询字段相同，也可以使用索引。**例如where A=a1 or A=a2（生效），where A=a or B=b（失效）。index merge 技术可以优化这个问题。
* **如果列类型是字符串，要使用引号。**例如where A='China'，否则索引失效（会进行类型转换），见下图：

**索引设计原则**

1. 第一个索引原则：尽量使用主键索引原则。（主键索引是聚簇索引，叶子结点保留完整数据；非聚簇索引需要回表，多查一次B+树）
2. 第二个索引原则：控制查询字段（ID替代通配符\*），覆盖索引的手段，能够大幅度提升性能。（只查询ID不用回表）
3. 第三个索引原则：最左前缀原则。（索引复用，索引项是按照索引定义里面出现的字段顺序排序的）
4. 第四个索引原则：联合索引，索引顺序往往就是需要优先考虑采用的。
5. 第五个索引原则：索引下推原则。（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）

**创建索引注意事项？**

* 表记录太少，不要建索引；
* 字段区分度低，建立索引没有太大意义；
* 频繁更新的字段不适合创建索引（需要维护索引，会加重IO负担）；
* where、order 、join 条件里用不到的字段不创建索引；
* 组合索引把散列性高(区分度高)的字段放在前面；
* MySQL中每次只能使用一个索引，所以经常使用多个条件查询时更适合使用组合索引；
* 过长的字段，使用 前缀索引。当字段值比较长的时候，建立索引会消耗很多的空间，搜索起来也会很慢。可以通过截取字段的前面一部分内容建立索引，这个就叫前缀索引。
* 不建议用无序的值(例如身份证、UUID )作为索引。当主键具有不确定性，会造成叶子节点频繁分裂，出现磁盘存储的碎片化。

**覆盖索引是什么?**

覆盖索引（Covering Index）不需要回表操作

* select 的数据列只用从索引中就能够取得，不必读取数据行，MySQL可以利用索引返回 select 列表中的字段，而不必根据索引再次读取数据文件，换句话说查询列要被所建的索引覆盖。
* 索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此它不必读取整个行。毕竟索引叶子节点存储了它们索引的数据，当能通过读取索引就可以得到想要的数据，那就不需要读取行了。一个索引包含（覆盖）满足查询结果的数据就叫做覆盖索引。
* 判断标准：使用explain，可以通过输出的extra列来判断，对于一个索引覆盖查询，显示为using index，MySQL查询优化器在执行查询前会决定是否有索引覆盖查询;

**带索引的查询方式**

例如，对于一个简单的B-Tree索引：

|  |
| --- |
| SQL SELECT \* FROM users WHERE id = 10; |

如果id是索引字段，MySQL会从索引树的根节点开始，根据id的值进行二分查找，直至找到具体的数据行。

如果索引是复合索引，例如：

|  |
| --- |
| SQL SELECT \* FROM users WHERE last\_name = 'Smith' AND first\_name = 'John'; |

对于复合索引(last\_name, first\_name)，MySQL会先找到所有last\_name为'Smith'的行，然后在这些行中查找first\_name为'John'的行。

总结：使用WHERE子句查询索引字段时，MySQL会利用索引来快速定位目标数据行，从而避免全表扫描，大大提高查询效率。

**什么是回表查询**

MySQL中的回表查询是指在执行查询时，首先通过索引定位到满足条件的行的位置，但由于索引可能不包含查询所需的所有数据，因此需要再次访问数据表以获取完整的数据行的过程。这个过程被称为回表查询。具体来说，回表查询发生在以下几种情况：

1. 索引不Cover所有需要查询的字段‌：当查询语句中需要返回的列不在索引列上时，即使通过索引定位了相关行，仍然需要回表获取其他列的值。
2. 使用了非聚簇索引‌：非聚簇索引（Secondary Index）只包含了索引列的副本以及指向对应主键的引用，查询需要通过回表才能获取完整的行数据。
3. 使用了覆盖索引但超过了最大索引长度‌：在MySQL的InnoDB存储引擎中，每个索引项的最大长度是767字节。如果查询需要返回的字段长度超过了该限制，同样会触发回表操作。

回表操作主要发生在读取操作（SELECT）中，写入操作（INSERT、UPDATE、DELETE）一般不会触发回表。为了减少回表查询的开销，可以考虑使用覆盖索引或优化聚簇索引的选择，以避免不必要的回表操作，从而提高查询性能。

**锁**

**锁粒度**

锁粒度指的是锁的范围大小。MySQL中的锁粒度主要有行锁、页锁和表锁。

行锁是细粒度锁，只锁定影响的行，允许其他不受影响的行为并发执行，提高了系统的并发能力。

页锁是中等粒度的锁，主要锁定数据页。

表锁是粗粒度锁，它会锁定整张表，适用于需要快速锁定整个表的操作。

**锁类别**

除了基本的表锁和行锁，MySQL还引入了意向锁（intent locks），分为意向共享锁（IX）和意向排他锁（IS）。

意向锁不是真正的锁，而是表明事务打算给数据加上共享或排他锁，用于在更高层次上避免锁冲突。

**从对数据操作的类型分类：**

* 读锁（共享锁）：针对同一份数据，多个读操作可以同时进行，不会互相影响;
* 写锁（排他锁）：当前写操作没有完成前，它会阻断其他写锁和读锁;

**从对数据操作的粒度分类：**

了尽可能提高数据库的并发度，每次锁定的数据范围越小越好，理论上每次只锁定当前操作的数据的方案会得到最大的并发度，但是管理锁是很耗资源的事情（涉及获取，检查，释放锁等动作），因此数据库系统需要在高并发响应和系统性能两方面进行平衡，这样就产生了“锁粒度（Lock granularity）”的概念。

* **表级锁：**开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低（MyISAM 和 MEMORY 存储引擎采用的是表级锁）；
* **行级锁：**开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高（InnoDB 存储引擎既支持行级锁也支持表级锁，但默认情况下是采用行级锁）；
* **页面锁：**开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

**MyISAM 表锁**

MyISAM 的表锁有两种模式：

* **表共享读锁 （Table Read Lock）：**不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求；
* **表独占写锁 （Table Write Lock）：**会阻塞其他用户对同一表的读和写操作；

MyISAM 表的读操作与写操作之间，以及写操作之间是串行的：当一个线程获得对一个表的写锁后， 只有持有锁的线程可以对表进行更新操作。其他线程的读、 写操作都会等待，直到锁被释放为止。

默认情况下，写锁比读锁具有更高的优先级：当一个锁释放时，这个锁会优先给写锁队列中等候的获取锁请求，然后再给读锁队列中等候的获取锁请求。

**InnoDB 行锁：**

* **共享锁（S）：**允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
* **排他锁（X）：**允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB 还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁：

* **意向共享锁（IS）**：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。
* **意向排他锁（IX）：**事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。

索引失效会导致行锁变表锁。比如 vchar 查询不写单引号的情况。

**加锁机制**

乐观锁与悲观锁是两种并发控制的思想，可用于解决丢失更新问题

* **乐观锁** 会“乐观地”假定大概率不会发生并发更新冲突，访问、处理数据过程中不加锁，只在更新数据时再根据版本号或时间戳判断是否有冲突，有则处理，无则提交事务。用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式
* 加上version的版本控制（后续进行更新操作，先比较version是否一致，不一致就需要重新取一遍数据）
* 也可以使用timestamp的控制（此处的时间戳应该使用数据库的而非业务系统的）。
* **悲观锁** 会“悲观地”假定大概率会发生并发更新冲突，访问、处理数据前就加排他锁，在整个数据处理过程中锁定数据，事务提交或回滚后才释放锁。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。
* 表锁(Table Lock)：对整个表进行加锁。
* 行锁(Record Lock)：对索引记录加锁。
* 间隙锁(Gap Lock)：锁住整个区间，包括：区间里具体的索引记录，不存在的空闲空间（可以是两个索引记录之间，也可能是第一个索引记录之前或最后一个索引记录之后的空间）。
* next-key锁：行锁和间隙锁组合起来。

**锁模式(InnoDB有三种行锁的算法)**

InnoDB这种行锁实现特点意味着：只有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！假设有个表单 products ，里面有 id 跟 name 二个栏位，id是主键。

* **记录锁(Record Locks)：**单个行记录上的锁。对索引项加锁，锁定符合条件的行。其他事务不能修改和删除加锁项；

|  |
| --- |
| SQL SELECT \* FROM table WHERE id = 1 FOR UPDATE; UPDATE SET age = 50 WHERE id = 1; |

它会在 id=1 的记录上加上记录锁，以阻止其他事务插入，更新，删除 id=1 这一行;

在通过 主键索引 与 唯一索引 对数据行进行 UPDATE 操作时，也会对该行数据加记录锁：

* **间隙锁（Gap Locks）：**当使用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁。对于键值在条件范围内但并不存在的记录，叫做“间隙”。隙锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况

|  |
| --- |
| SQL SELECT \* FROM table WHERE id BETWEN 1 AND 10 FOR UPDATE; |

* **临键锁(Next-key Locks)：**临键锁，是记录锁与间隙锁的组合，它的封锁范围，既包含索引记录，又包含索引区间。(临键锁的主要目的，也是为了避免幻读(Phantom Read)。如果把事务的隔离级别降级为RC，临键锁则也会失效。)

**锁的使用场景**

行锁通常用于InnoDB存储引擎，它在处理大量并发操作时能够提供更好的性能。

表锁适用于对全表进行操作的情况，如ALTER TABLE或批量更新。

**锁的获取与释放**

当事务需要修改数据时，必须先获得相应的锁。事务完成后，锁会被自动释放。

锁的获取和释放是通过特定的SQL语句来实现的，如SELECT FOR UPDATE来获取行锁，LOCK TABLES和UNLOCK TABLES来获取和释放表锁。

**锁冲突**

当多个事务试图同时获得同一资源上的不同类型的锁时，会发生锁冲突。

解决锁冲突的方法包括合理设计事务逻辑，减少锁的竞争，以及使用合适的隔离级别来降低锁冲突的可能性。

**InnoDB的行锁**

InnoDB存储引擎支持多种类型的行锁，包括记录锁、间隙锁、临键锁等。

这些锁类型可以帮助实现多版本并发控制（MVCC），提高并发性能。

**锁的缺点**

虽然锁可以保证数据的一致性，但不当的使用会导致死锁或降低系统性能。

**事务**

**ACID**

* **A (Atomicity) 原子性：**整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样
* **C (Consistency) 一致性：**在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏
* **I (Isolation)隔离性：**一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰
* **D (Durability) 持久性：**在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库文件中;

**并发事务**

**会带来什么问题**

**脏读(Dirty Reads)**

事务A读取了事务B中未提交的数据。如果B执行回滚操作，那么A读取到的数据是脏数据。

**幻读（Phantom Reads)**

一个事务A读取了几行数据，接着另一个并发事务B 插入 了一些数据时。在随后的查询中，事务A就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

**不可重复读（Non-Repeatable Reads)**

事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了 更新 并提交，导致事务A多次读取同一数据时，结果不一致。

**幻读和不可重复读的区别**

* 不可重复读的重点是修改：在同一事务中，同样的条件，第一次读的数据和第二次读的数据不一样。（因为中间有其他事务提交了修改）
* 幻读的重点在于新增或者删除：在同一事务中，同样的条件,，第一次和第二次读出来的记录数不一样。（因为中间有其他事务提交了插入/删除）

**并发事务处理的解决办法?**

“脏读” 、“幻读” 、 “不可重复读”，其实都是数据库读一致性问题，必须由数据库提供的事务隔离机制来解决：

* **加锁**：在读取数据前，对其加锁，阻止其他事务对数据进行修改。
* **数据多版本并发控制**（MultiVersion Concurrency Control，简称 MVCC 或 MCC）：不用加锁， 通过一定机制生成一个数据请求时间点的一致性 数据快照 （Snapshot)， 并用这个快照来提供一定级别 （语句级或事务级）

**事务隔离级别以及它们之间的区别**

* **读未提交（READ UNCOMMITTED）：**这是最低的隔离级别，它允许事务读取尚未提交的数据，也就是其他事务可能还没有完成的修改。这种级别的问题是它不能防止脏读、不可重复读和幻读，实现方式基本上是不加锁，因此性能影响较小，但数据一致性是最弱的。
* **读提交（READ COMMITTED）：**这个级别只允许事务读取已经提交的数据。它可以防止脏读，因为事务不会读取到其他事务未提交的数据。但是，它仍然可能遇到不可重复读和幻读的问题。通常通过加锁的方式来实现，同一时刻只允许一个事务写入数据。
* **可重复读（REPEATABLE READ）：**这是MySQL的默认隔离级别。它确保在一个事务内多次读取同样记录的结果是一致的，即防止了不可重复读的问题。不过，它还是可能遇到幻读的问题。在这个级别下，写操作会锁定行，而读操作不会阻塞写操作，为了提高性能，MySQL使用多版本并发控制（MVCC）来处理读-写和写-写的冲突。
* **串行化（SERIALIZABLE）：**这是最高的隔离级别，可以防止脏读、不可重复读和幻读。它通过加锁来实现，包括读锁和写锁，使得事务必须顺序执行，大大减少了并发性，因此对性能的影响也是最大的。

**MVCC机制**

MySQL的MVCC（多版本并发控制）是一种优化数据库读写操作的技术，它允许多个事务同时读取同一数据而无需等待其他事务完成。

**MVCC在MySQL中的实现**

* **隐藏字段：**InnoDB存储引擎会在每行记录中添加一些隐藏字段，如row\_id、trx\_id和roll\_pointer。这些字段用于维护每行数据的版本信息和事务信息。
* **undo log（回滚日志）：**当事务对数据进行修改时，InnoDB会将这些修改作为旧版本记录在undo log中。这样，即使数据被修改，其他事务也可以访问到该数据之前的版本，从而保持数据的一致性。
* **ReadView（一致性读视图）：**当一个事务开始时，它会创建一个ReadView，这个视图包含了该事务可见的所有活跃事务列表。通过这个视图，事务可以确定它所访问的数据版本是否是最新的，或者是否需要从undo log中获取旧版本的数据。
* **当前读和快照读：**MVCC区分了两种读取方式，即当前读和快照读。当前读总是读取最新的数据版本，而快照读则是基于事务开始时的快照来读取数据。这种区分使得即使在有读写冲突的情况下，也能实现非阻塞的并发读取。
* **性能和一致性：**MVCC的主要目的是在保证数据一致性的同时，提高数据库在高并发环境下的性能。如果没有MVCC机制，MySQL可能无法在高并发下同时保证数据的一致性和访问性能。

综上所述，MVCC通过维护数据的不同版本，确保了在并发环境下事务能够高效地读取和修改数据，同时保持了数据的一致性。这是MySQL InnoDB存储引擎能够支持高并发操作的重要特性之一。

**InnoDB 可重读隔离级别下MVCC如何工作？**

**SELECT**

InnoDB只查找版本早于当前事务版本的数据行，这样可以确保事务读取的行，要么是在开始事务之前已经存在要么是事务自身插入或者修改过的；

行的删除版本号要么未定义，要么大于当前事务版本号，这样可以确保事务读取到的行在事务开始之前未被删除；

**INSERT**

InnoDB为新插入的每一行保存当前系统版本号作为行版本号；

**DELETE**

InnoDB为删除的每一行保存当前系统版本号作为行删除标识

**UPDATE**

InnoDB为更新的一行新纪录保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行作为删除标识；

**事务的实现**

**隔离性**：通过锁实现

**原子性、持久性：**通过 redo log（重做日志）实现

**一致性：**通过 undo log（回滚日志）实现 ：

**日志**

**错误日志（Error Log）**

这是MySQL服务器的主要日志之一，记录了MySQL运行时出现的错误和警告信息。它对于问题的诊断和解决至关重要，因为它提供了关于异常情况的详细信息。

**慢查询日志（Slow Query Log）**

此日志记录执行时间超过long\_query\_time设置值的SQL语句，或者那些没有使用索引的查询。通过分析慢查询日志，可以找出性能瓶颈并优化相应的查询。

**二进制日志（Binary Log）**

记录了MySQL Server层执行的所有修改操作，包括DDL和DML语句。它主要用于复制过程中，允许在从服务器上重放主服务器上的更改，以及用于数据恢复。二进制日志是以事件的形式记录的，并且可以通过不同的日志格式（如statement、row、mixed）来配置。

**中继日志（Relay Log）**

在复制过程中，从服务器使用中继日志来存储从主服务器接收到的二进制日志事件。这些事件随后会被从服务器执行。

**DDL日志（DDL Log）**

专门用于记录数据定义语言（DDL）操作的日志类型。

**通用查询日志（General Log）**

记录MySQL Server层的所有查询语句，包括SELECT和SHOW等不修改数据的查询。这个日志通常用于审计或分析查询模式，但请注意开启它会对性能产生影响。

**InnoDB日志**

InnoDB作为MySQL的默认存储引擎，有自己特定的日志系统，包括redo log（重做日志）和undo log。redo log是物理日志，保证即使在系统崩溃的情况下也能保持数据的一致性；而undo log用来支持事务的回滚操作和MVCC（多版本并发控制）。MySQL采用日志先行，在真正写数据之前，会首先记录一个日志，叫Redo Log，会定期的使用CheckPoint技术将新的Redo Log刷入磁盘

* redo log（重做日志） 实现 原子性、持久性
* 事务提交之后，先将事务写入redo log日志，后续再将 Buffer Pool 中修改数据慢慢写入到磁盘。
* 此时如果数据库崩溃或者宕机，Buffer Pool 中的数据丢失，可能有部分已修改数据未能同步到磁盘。
* 那么，当系统重启进行恢复时，就可以根据redo log中记录的日志，把数据库恢复到崩溃前的一个状态。
* undo log（回滚日志）实现 一致性
* undo log记录了数据在操作前的状态，如果事务执行过程中需要回滚，就可以根据undo log进行回滚操作。单个事务的回滚，只会回滚当前事务做的操作，并不会影响到其他的事务做的操作。

**以一个update语句执行为例：**

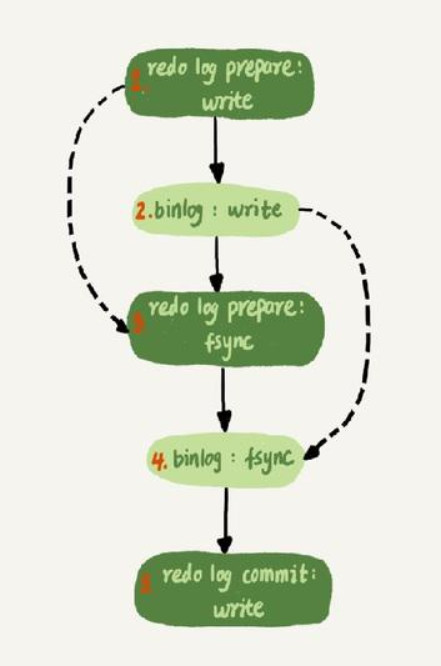
* Innodb引擎在收到一个update语句后，会先根据条件找到数据所在的页，并将该页缓存在 BufferPool 中（也就是内存中）。
* 执行update语句，修改 BufferPool 中的数据。
* 针对update语句生成一个 RedoLog 对象，并存入 LoggerBuffer 中。
* 针对update语句生成 UndoLog 日志，用于事务回滚。
* 如果提交事务，那么久将 RedoLog 对象进行持久化，后续还有其他机制将 BufferPool 中修改的数据页持久化到磁盘。
* 如果事务回滚，那么则利用 UndoLog 日志进行数据恢复。



**RedoLog 和 BinLog 区别**

* 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。
* **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 ”**。
* 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。
* bin log是在事务提交后才写入的，所以如果数据库在事务提交前崩溃，那么这个事务的修改就无法通过bin log恢复。redo log是在事务执行过程中就持续写入的，当事务提交时，必须确保相关的redo log已经写入磁盘。这样就可以保证即使数据库在事务提交前崩溃，也可以通过redo log把这个事务的修改恢复出来。

**mysql的两阶段提交原理**



**阶段1：**InnoDB redo log 写盘（引擎层），InnoDB 事务进入 prepare 状态；

**阶段2：**如果前面prepare成功，binlog 写盘（Server层），那么再继续将事务日志持久化到binlog，如果持久化成功，那么 InnoDB 事务 则进入 commit 状态(实际是在redo log里面写上一个commit记录)；

**好处：**

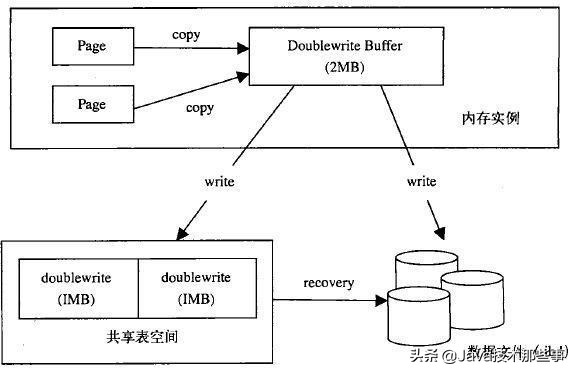
两阶段提交是跨系统维持数据逻辑一致性时常用的一个方案，这个方案也同时解决磁盘IO的性能。Redo两阶段是为了防止写入binlog失败。

**两次写**

鉴于都聊到了插入缓冲，我就不得不需要提一嘴两次写，因为我认为这两个InnoDB的特性是相辅相成的。

插入缓冲提高了MySQL的性能，而两次写则在此基础上提高了数据的可靠性。我们知道，当数据还在缓冲池中的时候，当机器宕机了，发生了写失效，有Redo Log来进行恢复。但是如果是在从缓冲池中将数据刷回磁盘的时候宕机了呢?

这种情况叫做部分写失效，此时重做日志就无法解决问题。



在刷网页时，并不是直接刷入磁盘，而是copy到内存中的Doublewrite Buffer中，然后再拷贝至磁盘共享表空间(你可以就理解为磁盘)中，每次写入1M，等copy完成后，再将Doublewrite Buffer中的页写入磁盘文件。

有了两次写机制，即使在刷脏页时宕机了，在实例恢复的时候也可以从共享表空间中找到Doublewrite Buffer的页副本，直接将其覆盖原来的数据页即可。

**重做日志缓冲**

上面聊过，InnoDB中缓冲池中的页数据更新会先于磁盘数据更新的，InnoDB也会采用日志先行(Write Ahead Log)策略来刷新数据，什么意思呢?当事务开始时，会先记录Redo Log到Redo Log Buffer中，然后再更新缓冲池页数据。

Redo Log Buffer中的数据会按照一定的频率写到重做日志中去。被更改过的页就会被标记成脏页，InnoDB会根据CheckPoint机制来将脏页刷到磁盘。

**InnoDB日志**

InnoDB日志就只有两种，Redo Log和Undo Log，

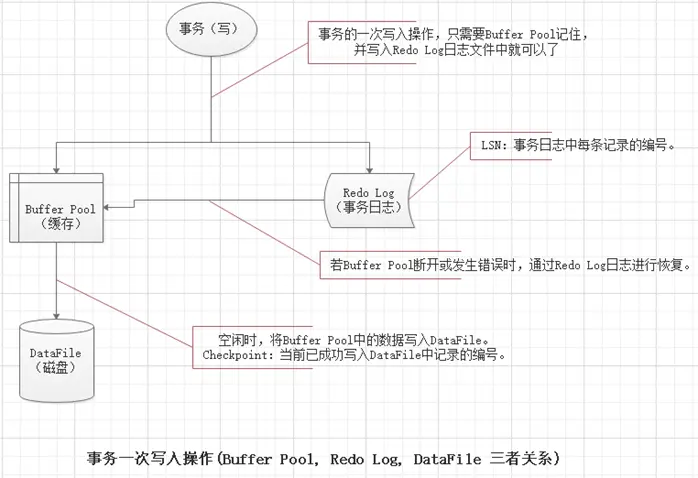
* Redo Log 重做日志，用于记录事务操作的变化，且记录的是修改之后的值。不管事务是否提交都会记录下来。例如在更新数据时，会先将更新的记录写到Redo Log中，再更新缓存中页中的数据。然后按照设置的更新策略，将内存中的数据刷回磁盘。
* Undo Log 记录的是记录的事务开始之前的一个版本，可用于事务失败之后发生的回滚。

Redo Log记录的是具体某个数据页上的修改，只能在当前Server使用，而Binlog可以理解为可以给其他类型的存储引擎使用。这也是Binlog的一个重要作用，那就是主从复制，另外一个作用是数据恢复。

上面提到过，Binlog中记录了所有对数据库的修改，其记录日志有三种格式。分别是Statement、Row和MixedLevel。

* Statement 记录所有会修改数据的SQL，其只会记录SQL，并不需要记录下这个SQL影响的所有行，减少了日志量，提高了性能。但是由于只是记录执行语句，不能保证在Slave节点上能够正确执行，所以还需要额外的记录一些上下文信息
* Row 只保存被修改的记录，与Statement只记录执行SQL来比较，Row会产生大量的日志。但是Row不用记录上下文信息了，只需要关注被改成啥样就行。
* MixedLevel 就是Statement和Row混合使用。

具体使用哪种日志，需要根据实际情况来决定。例如一条UPDATE语句更新了很多的数据，采用Statement会更加节省空间，但是相对的，Row会更加的可靠。



redo log开始随着事务中的数据的操作，写在redo log buff中，每次操作都会记录一个，然后在事务提交时候，先把他设置为prepare返回给执行器，执行器记录完binlog以后，执行最终的commit，redolog把 buffer中的内容（也是做后的结果）更新到os cache。然后os cache以他的频率通常是1s，fsync到磁盘。那么可以说，只要事务提交的话，就一定会进入binlog，redo log。而数据的更新则是一方面定时更新到磁盘，另一方面在不得已需要处理脏页的时候更新到磁盘。

force会在刷新完redo log以后也把数据的更新刷新到磁盘，就不有事务提交了但是数据还没有持久化的问题。

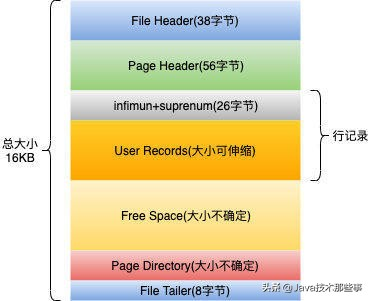
no-steal会保证没有commit的数据不会刷新到磁盘，就不会有数据已经在磁盘上了，但是还没有提交事务的情况。

就是数据更新到磁盘的线程和redo log刷新到磁盘的线程是两个独立的线程，两个线程执行先后不一样会造成数据和redo log的不一致。数据到了但是redo log没到或者redo log到了但是数据没到。这两种问题，如果是数据到了redo log没到，就需要通过undo log回滚。而redo log到了，数据没到，就需要用redo log重做。

redo log不能一直追加，要不然恢复的时候数据量太大效率很低，redolog有大小限制，可以重复利用。循环写入，假设编号依此是，1、2、3、4，从1到4顺序写入，在4写入以后在从1开始写，但是1的内容需要是里面数据的修改已经同步到磁盘的了。所以redo log file上有两个标志位，一个是checkpoint，一个是write\_pos，checkpoint以前的代表数据已经同步到磁盘，checkpoint后面是还没有同步到磁盘。

**页**

页是InnoDB中数据管理的最小单位。当我们查询数据时，其是以页为单位，将磁盘中的数据加载到缓冲池中的。同理，更新数据也是以页为单位，将我们对数据的修改刷回磁盘。每页的默认大小为16k，每页中包含了若干行的数据，页的结构如下图所示。



不用太纠结每个区是干嘛的，我们只需要知道这样设计的好处在哪儿。每一页的数据，可以通过FileHeader中的上一页和下一页的数据，页与页之间可以形成双向链表。因为在实际的物理存储上，数据并不是连续存储的。你可以把它理解成G1的Region在内存中的分布。 而一页中所包含的行数据，行与行之间则形成了单向链表。我们存入的行数据最终会到User Records中，当然最初User Records并不占据任何存储空间。随着我们存入的数据越来越多，User Records会越来越大，Free Space的空间会越来越小，直到被占用完，就会申请新的数据页。 User Records中的数据，是按照主键id来进行排序的，当我们按照主键来进行查找时，会沿着这个单向链表一直往后找

**分布式**

**主从复制**

MySQL的主从复制是一种数据同步机制，它允许数据从一个MySQL数据库服务器（主节点）复制到一个或多个其他服务器（从节点）。

**主从复制的基本原理包括以下几个步骤：**

**日志记录：**主节点上的所有更改都会被记录在二进制日志（binlog）中。

**日志发送：**从节点连接到主节点，并请求从指定的位置开始读取binlog。

**日志接收：**从节点接收来自主节点的binlog，并将其写入自己的中继日志（relay log）。

**日志应用：**从节点读取中继日志中的事件，并按照这些事件的指示执行相应的操作，从而保持与主节点的数据一致性。

此外，**主从复制还支持三种同步策略**，分别是异步复制、半同步复制和全同步复制。**异步复制**是最常见的方式，主节点在完成事务提交后不等待从节点确认就直接继续处理新的事务。**半同步复制**要求主节点在提交事务后至少得到一个从节点的确认才继续执行新的事务。**全同步复制**则是在所有从节点都确认接收到日志事件后，主节点才会提交事务。

综上所述，主从复制不仅提高了数据的可用性和安全性，还能够通过读写分离来提升数据库的性能。然而，主从复制也可能引入一定的延时，并且需要妥善处理网络分区等复杂情况。

**分库分表**

MySQL的分库分表策略是一种解决数据库性能瓶颈的方法，主要包括垂直分表和水平分表两种策略。具体介绍如下：

* **垂直分表：**这种策略是将一张表按照列进行拆分，每个表只包含原表中的一部分列。这通常用于处理那些包含大量无关字段的表，通过拆分可以减少单行数据的大小，提高查询效率。例如，将用户信息和订单信息拆分成两个表，使得它们可以分别存储在不同的表中。
* **水平分表：**这种策略是按照记录进行拆分，通常是基于某个关键字段（如用户ID）的值来分散数据到不同的表中。这种方法适用于单表数据量巨大，增长速度快的情况，可以通过分表来减少单表的数据量，提高查询和写入的速度。

**分库分表的优点包括**

* **提升性能：**通过分散数据到不同的数据库或表中，可以减少单一数据库或表的负载，提高查询和写入的速度。
* **便于扩展：**当业务增长时，可以通过增加更多的数据库或表来应对数据量的增长，而不需要对现有结构进行大规模的改动。

**分库分表的缺点包括**

* **复杂性增加：**分库分表引入了额外的复杂性，需要处理数据分布、跨库或跨表的联合查询等问题。
* **维护成本：**随着数据库实例的增加，维护的难度和成本也会相应增加。
* **事务管理：**跨库或跨表的操作可能会影响事务管理的复杂性，特别是在需要保证一致性的情况下。

**mysql分库分表的场景下，如何查询数据？**

在MySQL分库分表的场景下，查询数据可以通过多种方式实现，包括使用分库分表中间件（如ShardingSphere、MyCAT等），或者手动管理数据的路由。

以下是一个使用MySQL原生SQL语句进行跨库跨表查询的简单示例：

假设有两个数据库：db1 和 db2，分别包含表 users 和 orders。

|  |
| --- |
| SQL *-- 查询db1中的users表和db2中的orders表，连接条件是用户ID和订单所属用户ID* SELECT db1.users.id, db1.users.name, db2.orders.order\_id, db2.orders.order\_date FROM db1.users INNER JOIN db2.orders ON db1.users.id = db2.orders.user\_id WHERE db1.users.name = 'Alice'; |

在实际应用中，为了避免直接编写多个数据库的SQL语句，通常会使用分库分表中间件来透明地管理数据的分布和查询路由。例如，使用ShardingSphere时，可以通过其提供的分片键来路由数据。

如果你需要进一步的代码级别的实现，可以使用数据库连接池（如HikariCP）来管理多个数据源，并使用框架或自定义代码来实现跨库跨表的查询。

请注意，直接跨多个数据库执行复杂查询可能会导致性能问题和事务一致性难以保证，因此在设计系统时需要充分考虑这些因素。

**高可用和容灾的常见方案**

**双主复制（Master-Master Replication）**

这种模式下，两个MySQL服务器互为主从关系，任何一个服务器接收到的数据变更都会复制到另一个服务器。这样即使其中一个服务器宕机，另一个服务器仍然可以提供服务，从而实现高可用性。

**集群（Cluster）**

MySQL集群通过将多个服务器组织在一起，提供了一个单一的数据视图。如果集群中的某个节点失败，其他节点可以继续提供服务，从而保证了高可用性。MySQL提供了多种集群技术，如InnoDB Cluster、Galera Cluster等。

**MMM（Multi-Master Replication Manager）**

MMM是一种基于Perl的工具，用于管理多个主服务器之间的复制。它可以实现负载均衡和故障转移，提高了数据库的高可用性。

**主从复制（Master-Slave Replication）**

这是MySQL自带的功能，通过将一个主服务器的数据复制到一个或多个从服务器，可以实现数据的冗余备份和读写分离。在主服务器出现故障时，可以快速切换到从服务器，以减少停机时间。

**性能优化**

**MySQL常见瓶颈**

* CPU ：CPU在饱和的时候一般发生在数据装入内存或从磁盘上读取数据时候;
* IO ：磁盘I/O瓶颈发生在装入数据远大于内存容量的时候;
* 服务器硬件的性能瓶颈：top，free，iostat 和 vmstat来查看系统的性能状态;

**SQL执行时间长的原因分析**

* 查询语句写的烂 ；
* 索引失效（单值、复合）；
* 关联查询太多join（设计缺陷或不得已的需求）；
* 服务器调优及各个参数设置（缓冲、线程数等）；

**优化手段**

**合理使用索引**

索引是提高数据库查询速度的重要手段。合理创建和维护索引可以大幅度减少查询时间。这包括为主键选择合适的字段、创建复合索引以及使用覆盖索引等策略。

**优化查询语句**

编写高效的SQL查询语句对于性能至关重要。这可能涉及到避免不必要的关联查询、减少使用子查询、利用EXPLAIN分析查询计划等技巧。

**适当调整服务器配置**

服务器的硬件和软件配置对MySQL性能有显著影响。优化配置文件中的参数，如内存缓冲区大小、连接数等，可以改善性能。

**定期维护数据库**

定期清理垃圾数据、重新组织表和索引、更新统计信息等维护活动有助于保持数据库的良好状态。

**合理分配资源和连接管理**

优化资源分配策略，如连接池的使用，可以减少资源的浪费和等待时间。

**使用缓存技术**

利用缓存层可以减少数据库的读取操作，提高数据获取的速度。这可以是本地缓存或者分布式缓存，如Redis。

**数据分区和分表**

对于大型表，通过数据分区和分表可以减少单一表的数据量，提高查询和维护的效率。

**应用负载均衡技术**

在高负载环境下，使用负载均衡技术可以分散请求，防止单点过载。

**优化数据库设计**

遵循数据库设计原则和范式规范，设计出结构合理、扩展性好的数据库模式。

**监控性能和调优**

通过监控系统性能，及时发现并解决性能问题。这包括使用慢查询日志、EXPLAIN分析查询、profiling分析等工具和方法。

**MySQL的故障排查和诊断**

**检查系统资源使用情况**

当遇到MySQL无响应或crash的情况时，首先应检查内存使用情况。可以使用操作系统提供的工具（如free -m）来查看当前内存使用量，并判断是否出现了内存泄漏或OOM（内存溢出）现象。

**检查MySQL服务状态**

确认MySQL服务是否已经安装并正在运行。在Windows系统中，可以在服务管理器中查看MySQL服务的状态，确保它是处于“运行”状态。

**检查MySQL配置**

审查MySQL的配置文件（通常是my.cnf或my.ini），以确定是否有错误或需要调整的参数。例如，如果存在连接问题，可能需要修改skip-name-resolve来加快远程连接速度。

**使用MySQL自带工具**

利用MySQL自带的工具进行故障排查，如mysqldump用于备份和还原数据库，mytop实时监控MySQL状态，以及MySQL Performance Schema提供性能统计信息帮助性能分析。

**检查文件权限和磁盘空间**

确保数据库文件具有正确的属组和权限，避免由于权限问题导致的错误。同时检查磁盘空间是否足够，以防止因磁盘满导致的问题。

**检查表损坏问题**

如果遇到特定的错误代码，如无法打开某个表文件，可能是由于服务器非正常关机或其他原因导致表损坏。这时可以使用myisamchk工具或phpMyAdmin来修复数据表。

**处理连接问题**

对于出现的连接数过多问题，可以临时或永久地增加max\_connections参数的值来允许更多的连接。

**解决访问控制问题**

如果由于密码问题无法登录MySQL，可以通过跳过授权表的方式来重置密码，并重新授权。

**监控和分析性能**

通过监控工具定期检查MySQL的性能指标，以便及时发现潜在问题并进行优化。

**MySQL的数据备份和恢复策略**

**备份策略**

* **物理备份：**物理备份通常指的是对数据库文件的直接复制，包括数据文件、日志文件等。这种备份方式适用于需要快速恢复整个数据库的场景。物理备份可以是“冷备份”，即在数据库关闭时进行，也可以是“热备份”，即在数据库运行时进行。
* **逻辑备份：**逻辑备份是指使用mysqldump命令将数据库内容导出为SQL文件。这种方法便于备份特定的数据库或表，并且可以在不同版本的MySQL之间迁移数据。逻辑备份的优点是备份文件较小，恢复速度较快，但可能会对运行中的数据库性能产生影响。
* **二进制日志备份：**启用二进制日志（Binary Log）后，可以记录数据库的所有更改操作。这些日志文件可以用来恢复数据，或者用于主从复制。二进制日志备份通常与全备份结合使用，以实现点时间恢复（Point-in-Time Recovery，PITR）。

**恢复策略**

* **完全恢复：**在数据库损坏或丢失的情况下，可以使用完全备份进行恢复。这通常涉及到将备份的数据文件和日志文件复制回原位置，并确保数据库服务能够正常启动。
* **部分恢复：**如果只需要恢复数据库中的某个表或部分数据，可以使用逻辑备份的SQL文件进行恢复。通过执行这些SQL脚本，可以将数据恢复到备份时的状态。
* **点时间恢复：**利用二进制日志文件（binlog），可以将数据库恢复到特定时间点的状态。这要求在恢复过程中使用到全备份以及自备份以来的所有二进制日志文件。

**问题记录**

**mysqll数据 修改 ,通过什么方式,能够使得下游迅速得到数据变化**

使用触发器

**MySQL Cluster怎么做分布式一致性的**

MySQL Cluster 是一个完整的数据库集群解决方案，它包括了数据的复制，负载平衡以及高可用性等特性。在MySQL Cluster中，数据的分布和复制是通过NDB Cluster存储引擎来实现的。

数据的分布和复制是通过以下几个方面来保证数据的一致性和完整性：

1. 事务控制：MySQL Cluster 支持标准的事务控制语句，如 BEGIN, COMMIT, 和 ROLLBACK。所有的事务都是在所有的数据节点上应用或不应用。
2. 行级锁定：MySQL Cluster 使用了NDB的行级锁定技术，保证了数据的完整性。
3. 内部复制：每个数据节点都会复制其他所有的数据节点上的数据变化。
4. 节点失效检测：如果某个节点失效，其他的节点会自动接管失效节点的工作。
5. 自动恢复：如果某个节点恢复，它会自动从其他节点同步数据。
6. 数据一致性：通过使用原子消息传递和两阶段提交机制，MySQL Cluster 保证了数据的一致性和完整性。

**Mysql中大于小于会不会走索引?**

在MySQL中，使用大于(>)、小于(<)等运算符时，索引可能会被用来优化查询。如果查询能够利用索引来直接定位行，那么这样的查询通常会走索引。但是，如果查询中还包含某些不能利用索引的操作，或者查询的范围过于宽泛，那么即使有索引，查询也可能不会走索引。

具体来说，以下情况可能会导致使用索引：

1. 查询的条件是索引列上的等值比较。
2. 查询的条件是索引列上的范围查询，并且范围较小。
3. 查询的条件是索引列上的排序或分组。

以下情况可能不会使用索引：

1. 如果查询条件是不等于(<>)或者使用了函数在索引列上。
2. 如果查询的范围过于宽泛，导致索引的用处不大。
3. 如果查询条件是使用了复合索引，但是没有正确地遵守索引的最左前缀原则。

为了确保查询能够利用索引，可以通过以下方式优化查询：

* 确保查询条件中使用的列是索引的一部分，并且查询条件遵循索引的最左前缀原则。
* 对于范围查询，尽量使用索引的范围，而不是全索引范围。
* 避免在索引列上使用函数、类型转换或其他表达式。
* 对于复合索引，确保查询条件中包含索引中的第一个列。

**mysql的联合索引的最左匹配**

MySQL中的联合索引（也称为复合索引或组合索引）是指在表中对多个列进行索引的索引方式。最左匹配原则是指在使用联合索引时，查询条件要从索引的最左边开始，不能跳过中间的列。

例如，如果有一个联合索引(col1, col2, col3)，以下查询可以有效利用这个索引：

|  |
| --- |
| SQL SELECT \* FROM table\_name WHERE col1 = 'value1'; SELECT \* FROM table\_name WHERE col1 = 'value1' AND col2 = 'value2'; SELECT \* FROM table\_name WHERE col1 = 'value1' AND col2 = 'value2' AND col3 = 'value3'; |

但是，下面的查询不会使用索引：

|  |
| --- |
| SQL SELECT \* FROM table\_name WHERE col2 = 'value2'; SELECT \* FROM table\_name WHERE col3 = 'value3'; SELECT \* FROM table\_name WHERE col2 = 'value2' AND col3 = 'value3'; |

为了确保查询能够有效利用联合索引，设计索引时应该遵循最左匹配原则，将最常用作过滤条件的列放在索引的最左边。

**MySQL中的SELECT语句查询会用到缓存吗**

MySQL提供了查询缓存功能，允许对SELECT查询的结果进行缓存。当执行一个SELECT查询时，如果查询的结果之前已经被缓存过（即之前有过相同或相似的查询并且其结果已经被保存在缓存中），MySQL可以直接从缓存中检索结果并返回给客户端，从而避免了重新执行查询和数据库访问的开销。这种机制对于提高数据库查询性能非常有帮助，尤其是在高并发环境下，可以显著减少数据库服务器的负载。

MySQL的查询缓存机制允许通过[SQL\_CACHE](https://www.baidu.com/s?sa=re_dqa_generate&wd=SQL_CACHE&rsv_pq=f7790aac028d3dc8&oq=mysql%E4%B8%ADselect%E6%9C%89%E7%94%A8%E5%88%B0mysql%E7%BC%93%E5%AD%98%E5%90%97&rsv_t=8ccfmuNefH90hK8NQ4umVPrhviSs0BAlUQpjP4EexBlc7xyfEz1yodGxBYQ&tn=baidu&ie=utf-8)和[SQL\_NO\_CACHE](https://www.baidu.com/s?sa=re_dqa_generate&wd=SQL_NO_CACHE&rsv_pq=f7790aac028d3dc8&oq=mysql%E4%B8%ADselect%E6%9C%89%E7%94%A8%E5%88%B0mysql%E7%BC%93%E5%AD%98%E5%90%97&rsv_t=8ccfmuNefH90hK8NQ4umVPrhviSs0BAlUQpjP4EexBlc7xyfEz1yodGxBYQ&tn=baidu&ie=utf-8)选项来控制是否使用查询缓存。当使用[SQL\_CACHE](https://www.baidu.com/s?sa=re_dqa_generate&wd=SQL_CACHE&rsv_pq=f7790aac028d3dc8&oq=mysql%E4%B8%ADselect%E6%9C%89%E7%94%A8%E5%88%B0mysql%E7%BC%93%E5%AD%98%E5%90%97&rsv_t=8ccfmuNefH90hK8NQ4umVPrhviSs0BAlUQpjP4EexBlc7xyfEz1yodGxBYQ&tn=baidu&ie=utf-8)选项时，如果查询结果是可缓存的，并且[query\_cache\_type](https://www.baidu.com/s?sa=re_dqa_generate&wd=query_cache_type&rsv_pq=f7790aac028d3dc8&oq=mysql%E4%B8%ADselect%E6%9C%89%E7%94%A8%E5%88%B0mysql%E7%BC%93%E5%AD%98%E5%90%97&rsv_t=8ccfmuNefH90hK8NQ4umVPrhviSs0BAlUQpjP4EexBlc7xyfEz1yodGxBYQ&tn=baidu&ie=utf-8)系统变量的值为ON或DEMAND，则缓存查询结果。而使用[SQL\_NO\_CACHE](https://www.baidu.com/s?sa=re_dqa_generate&wd=SQL_NO_CACHE&rsv_pq=f7790aac028d3dc8&oq=mysql%E4%B8%ADselect%E6%9C%89%E7%94%A8%E5%88%B0mysql%E7%BC%93%E5%AD%98%E5%90%97&rsv_t=8ccfmuNefH90hK8NQ4umVPrhviSs0BAlUQpjP4EexBlc7xyfEz1yodGxBYQ&tn=baidu&ie=utf-8)选项时，服务器不会使用查询缓存，即不检查查询缓存也不缓存查询结果1。

然而，需要注意的是，查询缓存也有其局限性。例如，如果查询中涉及的任何表的数据发生变化，那么与这些表相关的所有缓存数据都将失效。此外，对于复杂的查询或涉及大量数据更新的应用场景，查询缓存可能并不总是有效的，甚至可能成为性能瓶颈2。因此，在使用查询缓存时，需要根据具体的应用场景和需求来评估其效果。