·NULL值列表 二进制位的值为1时,代表该列的值为NULL。 - 标记着当前记录是否被删除,占用1个二进制位, - 这些被删除的记录之所以不立即从磁盘上移除,因为移除它们之后把其他的记录在磁盘上重新排列需要性能消耗,所以只是打一个删除标记 delete mask —— ·所有被删除掉的记录都会组成一个垃圾链表 ,在这个链表中的记录占用的空间称之为所谓的可重用空间,可复用 - 将这个delete_mask位设置为1和将被删除的记录加入到垃圾链表中其实是两个阶段 记录的额外信息 / min_rec_mask ———— B+树的每层非叶子节点中的最小记录都会添加该标记 n_owned ——— 当前记录拥有的记录数 · 行格式 记录头信息 - 当前记录在本页中的位置 - InnoDB会自动给每个页里边儿加 最小记录Infimum(0) 最大记录Supremum(1) - 0 表示普通记录 - 1 表示B+树非叶节点记录 - 2 表示最小记录 - 3 表示最大记录 next_record — 从当前记录的真实数据到下一条记录的真实数据的地址偏移量(链表) ─ transaction_id ── 事务ID ・记录的真实数据 roll_pointer ——— 回滚指针 1. 将所有正常的记录(包括最大和最小记录,不包括标记为已删除的记录)划分为几个组。 2. 每个组的最后一条记录(也就是组内最大的那条记录)的头信息中的 n_owned属性表示该记录拥有多少条记录,也就是该组内共有几条记录。 3. 将每个组的最后一条记录地址偏移量单独提取出来按顺序存储到靠近 页 的尾部的地方,这个地方就是所谓的 Page Directory ,也就是 页目录 ← 页目录(Page Directory) 4. 页面目录中的这些地址偏移量被称为 槽 (英文名: Slot),所以这个页面目录就是由 槽 组成的。 •如何从页目录查找记录: 因为各个槽代表的记录的主键值都是从小到大排序的,所以我们可以使用所谓的二分法来进行快速查找 1. File Header ,表示页的一些通用信息,占固定的38字节。 存储引擎 2. Page Header ,表示数据页专有的一些信息,占固定的56个字节。 3. Infimum + Supremum ,两个虚拟的伪记录,分别表示页中的最小和最大记录,占固定的 26 个字节。 ─ InnoDB数据页结构 —— —— 4. User Records :真实存储我们插入的记录的部分,大小不固定。 5. Free Space : 页中尚未使用的部分,大小不确定。 6. Page Directory: 页中的某些记录相对位置,也就是各个槽在页面中的地址偏移量 7. File Trailer:用于检验页是否完整的部分,占用固定的8个字节。 InnoDB是聚集索引,使用B+Tree作为索引结构,数据文件是和(主键)索引绑在一起的; MyISAM是非聚集索引,它也是使用B+Tree作为索引结构,但是索引和数据文件是分离的,索引保存的是数据文件的指针。 索引类型 InnoDB辅助索引和主键索引之间存在层级关系; MyISAM辅助索引和主键索引则是平级关系。 InnoDB 如果添加其他二级索引,二级索引查询就需要两次查询,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,其他索引也相应都会很大 InnoDB聚集索引, 存放主键对应的行记录, 非聚集索引, 存放索引列+主键 / 叶子结点 — MyISAM所有的索引存储的是索引列+数据存放的地址指针,指向存放数据文件的对应行地址 InnoDB 必须要有主键, MyISAM可以没有主键; InnoDB 如果我们没有明确去指定创建主键索引。它会帮我们隐藏的生成一个 6 byte 的 int 型的索引作为主键索引 事务 InnoDB 支持事务, MyISAM 不支持 MyISAM存储引擎与InnoDB存储引擎的区别? 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(注意不能加有任何WHERE条件) ·全文索引 ———— InnodbMySQL 5.7 版本以后支持全文索引,而MyISAM支持全文索引,在全文索引领域的查询效率上MyISAM速度更快高; InnoDB支持表级锁、行级锁,默认为行级锁;而 MyISAM 仅支持表级锁。 InnoDB 的行锁是实现在索引上的,而不是锁在物理行上。如果访问未命中索引,也是无法使用行锁,将会退化为表锁 ► 外键 ——— InnoDB 支持外键 , 而 MyISAM 不支持 Innodb存储文件有frm、ibd , ; 【InnoDB中,.frm文件:保存的是表结构定义描述文件;.ibd文件:保存的是表中的数据内容,索引和数据保存在一起】 ▶ 存储方式 ——— Myisam是frm、MYD、MYI。 【MyISAM中,.frm文件:保存的是表结构定义描述文件,.MYD文件:保存的是数据内容,.MYI文件:保存的是索引内容, 索引数据分开】 Memory中基于Hash索引,底层是 Hash 表,对于精确查询非常高效。无法通过索引做区间查询。只能扫描全表。(范围查询、排序、组合索引 数据都是存储在内存中, IO效率比其他引擎高很多; (优点:效率高) Memory存储引擎 服务重启后数据会丢失,内存数据表默认只有16M(缺点:保证不了持久性) - 支持 Hash索引,B Tree索引,默认为 Hash索引 不支持大数据存储类型,如 blog,text等 1. 在 页目录 中使用二分法快速定位到对应的槽 以主键为搜索条件 —— 2. 然后再遍历该槽对应分组中的记录即可快速找到指定的记录。 一个页中的查找 1. 因为在数据页中并没有对非主键列建立所谓的 页目录 ,所以我们无法通过二分法快速定位相应的 槽 。 没有索引 · 以其他列作为搜索条件 —— 2. 只能从 最小记录 开始依次遍历单链表中的每条记录,然后对比每条记录是不是符合搜索条件。很显然,这种查找的效率是非常低的。 1. 没有索引, 只能从第一个页沿着双向链表一直往下找到记录所在的页 在很多页中查找 ——— 2. 从所在的页内中查找相应的记录 ← 目录项记录 ────── 目录项 中的两个列是 主键 和 页号 , 复用了之前存储用户记录的数据页来存储目录项 , 用来表示目录项的记录称为 目录项记录 ALTER TABLE table_name ADD INDEX index_name (column_list); 创建删除索引 ———— CREATE INDEX index_name ON table_name (column_list); alter table user_index drop KEY name; - 页内的记录是按照主键的大小顺序排成一个单向链表。 - 存放目录项记录的页分为不同的层次, 在同一层次中的页也是根据页中目录项记录的主键大小顺序排成一个双向链表。 聚簇索引 - B+ 树的叶子节点存储的是完整的用户记录 ------- 所谓完整的用户记录 , 就是指这个记录中存储了所有列的值(包括隐藏列) 。 聚簇索引 就是数据的存储方式 (所有的用户记录都存储在了叶子节点), - 页内的记录是按照 索引列的大小顺序排成一个单向链表。 使用索引列(非主键)的值进行记录和页的排序 —— 💳 - 各个存放用户记录的页也是根据页中记录的索引 列大小顺序排成一个双向链表。 - 存放目录项记录的页分为不同的层次,在同一层次中的页也是根据页中目录项记录的 索引 列大小顺序排成一个双向链表 1. B+ 树的叶子节点存储的并不是完整的用户记录,而只是索引列+主键这两个列的值。 2. B+ 树的非叶子节点中不再是 主键+页号 的搭配, 而变成了 索引列+主键+页号 的搭配。 1. 如果我们需要完整的用户记录或者其他当前信息,我们必须再根据主键值去聚簇索引中再查找一遍完整的用户记录。 2. 不将用户记录放在二级索引的叶子结点因为太占空间, 相当于每建立一棵 B+ 树都需要把所有的用户记录再都拷贝一遍。 1. 先把各个记录和页按照 c2 列进行排序。 一 非聚簇索引 2. 在记录的 c2 列相同的情况下,采用 c3 列进行排序 ·联合索引 ———— 以多个列的大小为排序规则建立的B+树称为联合索引,本质上也是一个二级索引。(c2列+c3列) — 3. 目录项记录都由 c2 +c3 +页号 这三个部分组成,各条记录先按照 c2 列的值进行排序,如果记录的 c2 列相同,则按照 c3 列的值进行排序。 InnoDB索引分数 4. B+ 树叶子节点处的用户记录由 c2 、 c3 和主键 c1 列组成。 根据非聚簇索引查到的是索引列对应的数据和主键值,如果需要查找其他非索引列的数据或者数据完整的记录需要根据主键值到聚簇索引中查找完 整的记录, 这个过程称之为回表(一次查询用到了2棵B+树) ·太占空间,相当于每建立一棵 B+ 树都需要把所有的用户记录再都拷贝一遍。 ➤ 为什么不把所有的记录放到非聚簇索引的叶子结点? - 在修改用户记录的时候,所有的索引都需要修改叶子结点的记录,维护成本高,尤其涉及增删导致的页分裂甚至需要更新非叶子结点 类似于数据结构中HASH表,当我们在mysql中用哈希索引时,主要就是通过Hash算法将数据库字段数据转换成定长的Hash值,与这条数据的行 > 哈希索引 ———— 指针一并存入Hash表的对应位置; 如果发生Hash碰撞(两个不同关键字的Hash值相同),则在对应Hash键下以链表形式存储 index(field(10)),使用字段值的前10个字符建立索引, 使用字段的部分前缀字符建立索引,多余字段值较长,并且前面几个字段差别较大的列比较合适,节省了创建索引的空间成本 一个B+树索引的根节点自诞生之日起,便不会再移动 InnoDB的B+树索引的注意事项 一个页面最少存储2条记录 ———— 这是因为B+树本质上就是一个大的多层级目录,每经过一个目录时都会过滤掉许多无效的子目录,直到最后访问到存储真实数据的目录。 - 将表中的记录按照记录的插入顺序单独存储在一个文件中, 称之为 数据文件(MYD)。 使用B+树索引,但是却将索引和数据分开存储 ————- 这个文件并不划分为若干个数据页,有多少记录就往这个文件中塞多少记录就成了。 MyISAM中的索引 - 我们可以通过行号而快速访问到一条记录 使用 MyISAM 存储引擎的表会把索引信息另外存储到一个称为 索引文件(MYI)。 ——— 叶子结点存储的是索引列+行号,不管是不是主键索引,都需要进入数据文件通过行号找到对应的数据,所以MYISAM索引都是二级索引,都需要回表 • 搜索条件中的列和索引列一致并且是等值匹配 ・全值匹配 ≺ 一 只要索引覆盖了条件中的列,并且是等值匹配,顺序不影响,因为查询优化器会重排这些条件使得可以用索引查询 一 搜索条件中的各个列必须是联合索引中从最左边连续的列 - 匹配左边的列 -数据页和记录按照联合索引的索引列从左到右进行排序,在第一个列排序的基础上,相同的第一列按第二列排序,跳过左边的列,剩下的列不是有序的 以字符串的前缀建立索引, 只要列字符串值前缀的区分度大可以到达近似按整个字符串匹配的效果但是节省了空间和时间成本, 比较的时候无需比较 - 匹配列前缀 —— 所有的字符, 并且B+树索引存储的数据减少, 无需存储所有的字符串 范围匹配最左列 ———— 找到条件范围的左右端点,在叶子结点上从左端点一直读到右端点因为记录是双链表前后连接的 最左边列等值匹配,, 查找到的记录是按照第二列排序的, 所以第二个条件如果是最左列的右边一列, 可以用到索引进行范围查找, 但是后续的索引列 匹配范围值 ─ 等值匹配最左列并范围匹配其他列 -----B+树索引适用的条件 ~ 范围查询索引最左列在内的多个列 ───── 只能用到最左列的索引, 查到符合最左列的数据后, 需要进行全部记录的搜索找到符合剩下所有条件的记录 使用联合索引进行排序 ———— ORDER BY 的子句后边的列的顺序也必须按照索引列的顺序给出 文件排序 ------ 无法使用索引排序,需要将记录加载到内存中,通过排序算法对记录进行排序(在内存或磁盘进行的排序) ← ASC、DESC混用 ───── 索引只按照索引列的顺序升序排序,对于升降序混合的排序规则,无法使用索引 ── WHERE子句中出现非排序使用到的索引列 ────── 先查找符合where条件的记录再根据要求排序,查找阶段无法使用索引 一 不可以使用索引排序 - 排序列包含非同一个索引的列 ▶ 排序列使用了复杂的表达式 顺序IO(访问二级索引) ———— 访问二级索引, 对于索引列的范围查询, 是顺序IO, 因为叶子结点根据索引列排序, 查询到的记录是几种分布在一个或多个数据页, 效率较高 索引 但是根据索引列记录得到的主键进行回表,访问聚簇索引的过程因为,这些查询二级索引得到的主键值并不是排序的,可能分散在内存的不同的位 随机IO(访问聚簇索引) —— 置,导致查询磁盘的过程分散,速度也就慢了,因为不能一次读取连续的数据,需要等待磁盘旋转到对应的磁道扇区才能获得数据 · 需要回表的记录数越多,就越倾向于使用全表扫描 - 回表的代价 全表扫描 vs 二级索引 + 回表・ ─ 加了 LIMIT 子句,这样需要回表的记录特别少,优化器就会倾向于使用 二级索引 + 回表 的方式执行查询。 查询列表全部包含在二级索引或者联合索引, 查询到的记录中包含了这些查询列的值, 就无须到聚簇索引中回表查询了 覆盖索引 🔫 索引不推荐使用 select * 而是查询列表写全需要查询的列, 这样存在对应的二级索引或联合索引就无须回表了 一只为用于搜索、排序或分组的列创建索引 ———— 用不到的列,无须创建索引,节省时间空间成本 · 列的基数(某列不重复的数据个数) ———— 为那些列的基数大的列建立索引, 基数大表示不同的值多, 这样查询可以一次性过滤掉比较多的记录, 节省了IO成本, 获得更高的查询效率 - 尽量让索引列使用较小的类型,数据类型越小,查询比较的操作越快,保存索引的成本更小,一个页可以保存更多的记录 - 索引列的类型尽量小 ≺ 尤其是主键, 因为聚簇索引存储主键, 其他二级索引也都存储了二级索引 长字符串在查询比较的时候更耗时,存储消耗更大的空间,对于前缀差别较大的列比较适合,因为只需要前缀就可以得到相同的区分度,但是减少了比 如何选择索引 一 索引字符串值的前缀 ——— 较的时间和存储的成本 → 主键自增 ———— 插入记录最好按照主键递增的顺序插入,这样避免了在一堆数据中间插入一条记录导致满页产生页分裂,这样可能导致内结点的结构发生变化,如果 主键自增,每次插入的记录都在叶子结点最右边,最多造成最右页的也分裂而不会对前面的页造成影响 ~ 避免冗余和重复索引 ———— 增加维护成本 尽量不要让索引列存在任何表达式中出现而是单独存在,这样可以使用索引,否则堆索引列进行运算后,只能全表扫描.因为查询条件因为表达式已经 - 存放用户记录的数据页,目录项记录的数据页,都把它们存放到 B+ 树这个数据结构中了,所以我们也称这些数据页为节点。 - 实际用户记录其实都存放在B+树的最底层的节点上,这些节点也被称为 叶子节点 或 叶节点 , - 其余用来存放 目录项 的节点称为 非叶子节点 或者 内节点 ,其中 B+ 树最上边的那个节点也称为 根节点 B+树 ———— - 规定最下边的那层 , 也就是存放我们用户记录的那层为第 0 层 - B+ 树都不会超过4层 - 通过主键值去查找某条记录最多只需要做4个页面内的查找 针对数据页所做的目录就是索引 很难找到合理的hash算法降低hash碰撞 不能进行范围查询, 只支持等值查询再进行回表 - 不使用Hash索引 ----需要将hash表全部加载到内存占据较多宝贵的内存空间 memory使用hash索引因为数据都在内存,范湖查询快于磁盘交互 ·为什么使用B+数索引 一 不用二叉树 红黑树 ———— 一个结点只能分二叉,存储的数据有限,因为数的深度过深导致IO次数变多,影响数据读取的效率 > B树 ———— 内结点存储数据,一个页包含的索引信息,一次读取页可以过滤的信息较少,并且每层结点独立,对于返回查询较慢 B+树内部节点都是索引,没有记录,叶子节点同时存放键和记录 B树索引和记录存放在内部节点和叶子节点, 叶子节点各自独立 _ B+树内结点只存储索引不包含记录, 一次读取, 可以在内存页中获取更多的索引信息, 有利于更快地缩小查找范围, 并且减少IO次数 **-** IO效率 ----B树所有结点都存储记录,对于一次读取只能获得有限的索引信息,增加了IO次数 B树 vs B+树 B+树所有记录都放在叶子结点,都需要从根结点到叶子结点的查询,效率比较稳定 → 热点数据查询 -----B树将频繁访问的数据放在靠近根节点的地方将会大大提高热点数据的查询效率 B+树通过索引可以确定记录在叶子结点层的起始和终止页,从左到右查询即可,因为是双链表. 查询全部数据找到记录最小主键的记录从叶子结点从 范围查询 ——— 左到右扫描即可 B树所有结点都存储记录, 范围查询效率较低, 涉及更多次IO. 查询全部数据, 需要按层扫描, 涉及更多的页面置换 hash索引底层是hash表,进行查找时,对索引列进行hash就可以获取到相应的hash值(key)得到对应的主键,之后进行回表查询获得实际数据。 ┏ B+树底层实现是多路平衡查找树。对于每一次的查询都是从根节点出发,查找到叶子节点方可以获得所查键值,然后根据查询判断是否需要回表 / 等值查询 ———— hash索引进行等值查询更快(一般情况下无hash碰撞) 一因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询 Hash索引 vs B+树索引 ─ hash索引不支持使用索引进行排序 hash索引不支持模糊查询以及多列索引的最左前缀匹配 ~回表 ———— hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询。 hash索引虽然在等值查询上较快,但是不稳定。性能不可预测,当某个键值存在大量重复的时候,发生hash碰撞,此时效率可能极差。 - 聚簇索引叶子结点存放的是数据, 并且是以主键排序的 ▼聚簇索引和非聚簇索引的区别 ———— - 非聚簇索引叶子结点存放的是索引键对应记录的主键 主要区别在于数据与索引的存储形式,聚簇索引是在一起的而非聚簇索引是分开的 - 通过二级索引或者联合索引, 需要查找的数据不包括在索引列中, 所以二级索引联合索引的叶子结点没有包含需要的所有数据 - 需要根据查询出来的行记录的主键值去到聚簇索引找到这条主键对应的完全行记录得到我们需要的数据 - 通过二级索引或者联合索引, 叶子结点中的数据包含了我们需要的列, 无需再去聚簇索引查询, ╱ 索引覆盖 ──── - 索引的列包含了全部的查询的列, 省去了回表的时间 → 最左匹配 → 查询条件对应的列按照联合索引列的顺序从左到右依次匹配,遇到范围查询就不能再使用索引查询,因为联合索引按照索引列从左到右升序排列,这 样对于索引列从左到右的排列, 自然的用到了索引排序的特点查询效率较快 常见名词 一家引下推 ——— 查询条件用到了索引,这样索引列在磁盘上排序的,在从磁盘获取数据的时候,直接根据查询条件返回符合条件的记录,而不是遇到符合条件一部分就 -返回再在内存中做筛选,因为索引覆盖,数据是排序的,影响较小,但是减少了IO次数, - 利用二级索引查找age > 10的记录得到了1000条记录的id(主键)值 · MRR(multi_range_read) ———— - 正常需要根据这些id去聚簇索引中找到对应的行记录, 每个都需要从根结点到叶子结点1000 O(log(depth)) - 我们得到id值后可以对这些id进行排序 然后到聚簇索引进行范围查询这样减少了查询的次数,尽可能将随机IO变成多个顺序IO 🔽 FIC(fast index create) ———— 给当前表添加一个Share锁,插入删除数据不会有创建临时表的消耗,还是源文件 / 连接 的本质就是把各个连接表中的记录都取出来将匹配的组合加入结果集并返回给用户 两表连接查询中,驱动表只需要访问一次,被驱动表可能被访问多次 连接查询的执行过程 一 第一个需要查询的表,这个表称之为驱动表。根据这个表的查询条件得到对应的记录 - 针对驱动表产生的结果集中的每一条记录 , 分别需要到 t 2 表中查找匹配的记录 驱动表中的记录在被驱动表中找不到匹配的记录,该记录不会加入到最后的结果集 内连接 (INNER JOIN) · - 驱动表和被驱动表是可以互换的 , 并不会影响最后的查询结果。 驱动表中的记录即使在被驱动表中没有匹配的记录,也仍然需要加入到结果集。对应的被驱动表记录的各个字段使用 NULL 值填充 - 内连接 和 外连接 必须使用 ON 子句来指出连接条件。 外连接 (LEFT | RIGHT JOIN) ── 左外连接 ──── 选取左侧的表为驱动表。 连接 - 右外连接 ———— 选取右侧的表为驱动表。 ➤ 使用索引加快连接速度 ———— 对于驱动表查询出来的记录,到被驱动表中查询的时候可以使用索引加快查询的速度,这取决于二级索引+回表的方式是否快于全表扫描 核心思想将驱动表中的记录放入join buffer 将被驱动表加载到内存内的时候,一次性和join buffer中的驱动表记录进行匹配,减少了将被驱动表加载 扫描一个表的过程其实是先把这个表从磁盘上加载到内存中,然后从内存中比较匹配条件是否满足。 ,被驱动表被访问多次的,如果被驱动表中的数据多而且不能使用索引进行访问,那就相当于要从磁盘上读好几次这个表,这个 I/O 代价就非常大了 驱动表结果集中有多少条记录,就得把被驱动表从磁盘上加载到内存中多少次进行比较 可不可以在把被驱动表的记录加载到内存的时候,一次性和多条驱动表中的记录做匹配,这样就大大减少重复从磁盘上加载被驱动表的代价了? ➤ 基于块的嵌套循环连接(Block Nested-Loop Join) - 执行连接查询前申请的一块固定大小的内存,先把若干条驱动表结果集中的记录装在这个 join buffer 中, - 然后开始扫描被驱动表,每一条被驱动表的记录一次性和 join buffer 中的多条驱动表记录做匹配, - 因为匹配的过程都是在内存中完成的, 所以这样可以显著减少被驱动表的 I/O 代价 - 最好的情况是 join buffer 足够大,能容纳驱动表结果集中的所有记录,这样只需要访问一次被驱动表就可以完成连接操作了 - 驱动表的记录并不是所有列都会被放到 join buffer 中,只有查询列表中的列和过滤条件中的列才会被放到 join buffer 中,所以再次提醒我们, 最好不要把*作为查询列表,只需要把我们关心的列放到查询列表就好了,这样还可以在 join buffer 中放置更多的记录 / 为了缓存磁盘中的页,在 MySQL 服务器启动的时候就向操作系统申请了一片连续的内存, 叫做 Buffer Pool Buffer Pool内部组成 ———— 前面存放每个缓存页对应的控制块 ———— 页所属的表空间编号、页号、缓存页在 Buffer Pool 中的地址、链表节点信息、一些锁信息以及 LSN 信息 所有空闲的缓存页对应的控制块作为一个节点放到一个链表中 **──** 这个链表定义了一个 基节点 ,里边儿包含着链表的头节点地址,尾节点地址,以及当前链表中节点的数量等信息 ·free链表 • - 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 · 存储脏页(dirty page)的链表,凡是修改过的缓存页对应的控制块都会作为一个节点加入到一个链表中 flush链表: - 每次修改缓存页后,我们并不着急立即把修改同步到磁盘上,而是在未来的某个时间点进行同步 创建一个链表,这个链表是为了按照最近最少使用的原则去淘汰缓存页的 · 访问的页不存在于buffer pool: 把页从磁盘加载到 Buffer Pool 中的缓存页时,就把该缓存页对应的控制块 作为节点塞到链表的头部。 ── 访问的页存在于buffer pool: 直接把该页对应的 控制块 移动到 LRU链表 的头部。 **Buffer Pool** / 简单LRU链表 预读机制导致:加载到 Buffer Pool 中的页不一定被用到。 ➤ 全表扫描导致: 使用频率偏低的页被同时加载到 Buffer Pool 时,可能会把那些使用频率非常高的页从Buffer Pool 中淘汰掉。 · 热数据(young区域) ———— 一部分存储使用频率非常高的缓存页 **LRU**链表 · 冷数据(old区域) ———— 另一部分存储使用频率不是很高的缓存页 当磁盘上的某个页面在初次加载到Buffer Pool中的某个缓存页时,该缓存页对应的控制块会被放到old区域的头部。 ╱ 针对预读的页面可能不进行后续访情况的优化 ──── 这样针对预读到 Buffer Pool 却不进行后续访问的页面就会被逐渐从old 区域逐出,而不会影响 young 区域中被使用比较频繁的缓存页 划分区域的LRU链表 - 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,那么该页面就不会被从old区域移动到young区域的头部,否则将它移动到young区 ➤ 针对全表扫描时,短时间内访问大量使用频率非常低的页面情况的优化 —— - innodb_old_blocks_time 控制间隔时间, 默认值是 1s · 只有被访问的缓存页位于 young 区域的 1/4 的后边 , 才会被移动到 LRU链表 头部 , - LRU链表分区+控制访问间隔时间: 使得用不到的预读页面以及全表扫描的页面都只会被放到 old 区域,不影响 young 区域中的缓存页。 处于一个事务中的操作要么全做要么全不做,不可分割,强调事务的不可分割. 原子性 (Atomicity) undo log日志保证,它记录了需要回滚的日志信息,事务回滚时撤销已经执行成功的sql 事务的执行的前后数据的完整性保持一致,从一个一致性的状态到了另一个一致性的状态 ─ 一致性 (Consistency) 代码层面来保证 r 数据库事务是指作为单个逻辑工作单元执行的一系列操作(SQL语句)。这些操作要么全部执行,要么全部不执行。 一个事务执行的过程中,不应该受到其他事务的干扰 **~** 隔离性 (Isolation) MVCC来保证 事务一旦结束,数据就持久到数据库 · 持久性 (Durability) 内存+redo log来保证, mysql修改数据同时在内存和redo log记录这次操作,事务提交通过redo log刷盘, 宕机的时候可以从redo log恢复 READ ONLY: — 标识当前事务是一个只读事务 START TRANSACTION; ─ READ WRITE: ──── 标识当前事务是一个读写事务 ~ WITH CONSISTENT SNAPSHOT : ──── 启动一致性读 ROLLBACK 语句就代表中止并回滚一个事务, 如果事务在执行过程中遇到了某些错误而无法继续执行的话,事务自身会自动的回滚 事务对应的数据库语句中打几个点,我们在调用 ROLLBACK 语句时可以指定会滚到哪个点,而不是回到最初的原点 一 中止事务 事务 SAVEPOINT 保存点名称; ─ MySQL中事务的语法 ROLLBACK TO 保存点名称 RELEASE SAVEPOINT 保存点名称 SET autocommit = ON DDL(Data definition language)
 CREATE \ ALTER \ DROP 个事务还没提交或者回滚时就又使用 START TRANSACTION 或者 BEGIN 语句开启了另一个事务时,会隐式的提交上一个事务 ・事务控制或关于锁定的语句 当前的 autocommit 系统变量的值为 OFF ,我们手动把它调为 ON 时,也会隐式的提交前边语句所属的事务 ➤ 加载数据的语句 ———— 使用 LOAD DATA 语句来批量往数据库中导入数据时 , 也会隐式的提交前边语句所属的事务。 · 脏读(Dirty Read) ———— 一个事务读到了另一个未提交事务修改过的数据 ── 不可重复读 (Non-Repeatable Read) ───── 一个事务读到另一个已经提交的事务修改过的数据 脏读, 幻读, 不可重复读 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 - READ UNCOMMITTED:未提交读。 ・事务的隔离级别 - REPEATABLE READ:可重复读。 ➤ SERIALIZABLE : 可串行化。 `binlog`用于记录数据库执行的更新数据库操作信息,以二进制的形式保存在磁盘中。 `binlog`是`mysql`的逻辑日志,并且由`Server`层进行记录,使用任何存储引擎的`mysql`数据库都会记录`binlog`日志。 `binlog`是通过追加的方式进行写入的,可以通过`max_binlog_size`参数设置每个`binlog`文件的大小, 当文件大小达到给定值之后,会生成新的文件来保存日志。 - 主从复制:在`Master`端开启`binlog`,然后将`binlog`发送到各个`Slave`端,`Slave`端重放`binlog`从而达到主从数据一致。 · binlog使用场景

─ 数据恢复:通过使用`mysqlbinlog`工具来恢复数据

— binlog日志格式

・MySQL服务器层面 ——— binlog

- 基于`SQL`语句的复制(`statement-based replication, SBR`),每一条会修改数据的sql语句会记录到`binlog`中。

- 基于行的复制(`row-based replication, RBR`), 不记录每条sql语句的上下文信息, 仅需记录哪条数据被修改了。

~ `STATMENT` ───── - 优点:不需要记录每一行的变化,减少了`binlog`日志量,节约了`IO`, 从而提高了性能;

- 缺点:会产生大量的日志,尤其是`altertable`的时候会让日志暴涨

- 缺点:在某些情况下会导致主从数据不一致,比如执行`sysdate()`、`slepp()`等。

─ `ROW` ───── - 优点:不会出现某些特定情况下的存储过程、或function、或trigger的调用和触发无法被正确复制的问题;

- 基于`STATMENT`和`ROW`两种模式的混合复制('mixed-based replication, MBR'),

列字段是变长数据类型的列值长度按照逆序存放

VARCHAR(M) 、 VARBINARY(M) 、各种 TEXT 类型 , 各种 BLOB 类型

将每个允许存储 NULL 的列对应一个二进制位,二进制位按照列的顺序逆序排列

变长字段长度列表

- SELECT 子句 ------ SELECT (SELECT m1 FROM t1 LIMIT 1);

・标量子查询 ----- 只返回一个单一值的子查询称之为 标量子查询 -----

一 不相关子查询 ———— 如果子查询可以单独运行出结果,而不依赖于外层查询的值

→ FROM 子句中 → 子查询的查询结果当作是一个表 → SELECT m, n FROM (SELECT m2 + 1 AS m, n2 AS n FROM t2 WHERE m2 > 2) AS t;

← 行子查询 ───── 一条记录的子查询,不过这条记录需要包含多个列 ───── SELECT * FROM t1 WHERE (m1, n1) = (SELECT m2, n2 FROM t2 LIMIT 1);

→ 列子查询 ———— 询出一个列的数据喽,不过这个列的数据需要包含多条记录 ———— SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2);

表子查询 ———— 子查询的结果既包含很多条记录,又包含很多个列 ———— SELECT * FROM t1 WHERE (m1, n1) IN (SELECT m2, n2 FROM t2);

SELECT (SELECT m1 FROM t1 LIMIT 1);

SELECT * FROM t1 WHERE m1 = (SELECT MIN(m2) FROM t2);

SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2);

基于规则

子查询可以出现的位置

按返回的结果集区分子查询

→ 垃圾口本海兰系並区公之本海

✓

- **琥珀LIR**E

仅一기法旦内大尔木心刀丁旦内 🥆 ── 相关子查询 ── 子查询的执行需要依赖于外层查询的值 ── SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2 WHERE n1 = n2); / 子查询必须用小括号扩起来。不扩起来的子查询是非法的 子查询优化 在 SELECT 子句中的子查询必须是标量子查询。 ─ 在想要得到标量子查询或者行子查询,但又不能保证子查询的结果集只有一条记录时,应该使用 LIMIT 1 语句来限制记录数量 子查询语法注意事项・ 一对于 [NOT] IN/ANY/SOME/ALL 子查询来说, 子查询中不允许有 LIMIT 语句。 ~ 子查询中 ORDER BY,DISTINCT,没有聚集函数以及 HAVING 子句的 GROUP BY 子句,都是多余的因为子查询的结果是集合相当于去重了 不允许在一条语句中增删改某个表的记录时同时还对该表进行子查询 不相关标量子查询或者行子查询 ———— 先执行子查询得到结果作为外层查询的参数进行外层查询 - 标量子查询、行子查询 ━ 子查询结果集中的记录保存到临时表的过程称之为 物化(Materialize) 因为单独执行子查询后的结果集太多导致: 1. 内存放不下, 2.IN参数太多, 只能对外层查询执行全表扫描 子查询执行顺序 写入临时表的记录会被去重(为表中记录的所有列建立主键或者唯一索引)建立的是基于内存使用的Memory存储引擎的hash索引,IN语句的本质就是判断某个操作数在不在某个集合里 ~ IN子查询优化 ──── 物化表・ _ 如果子查询的结果集非常大,超过了系统变量 tmp_table_size 或者 max_heap_table_size ,临时表会转而使用基于磁盘的存储引擎来保存结果集 中的记录,索引类型也对应转变为 B+ 树索引 ← table ———— EXPLAIN语句输出的每条记录都对应着某个单表的访问方法,该条记录的table列代表着该表的表名 ✓ 连接查询的执行计划中,每个表都会对应一条记录,但是这些记录的id值都是相同的 ───── 查询语句中每出现一个 SELECT 关键字,为它分配一个唯一的 id 值。这个 id 值就是EXPLAIN 语句的第一个列 一子查询的查询语句的执行计划中,每个 SELECT 关键字都会对应一个唯一的 id 值 Explain 一 查询优化器可能对涉及子查询的查询语句进行重写,从而转换为连接查询。两条记录的id就相同了说明子查询转换成了连接 _ possible_keys 列表示在某个查询语句中,对某个表执行单表查询时可能用到的索引有哪些,key 列表示实际用到的索引有哪些 possible_keys和key ——