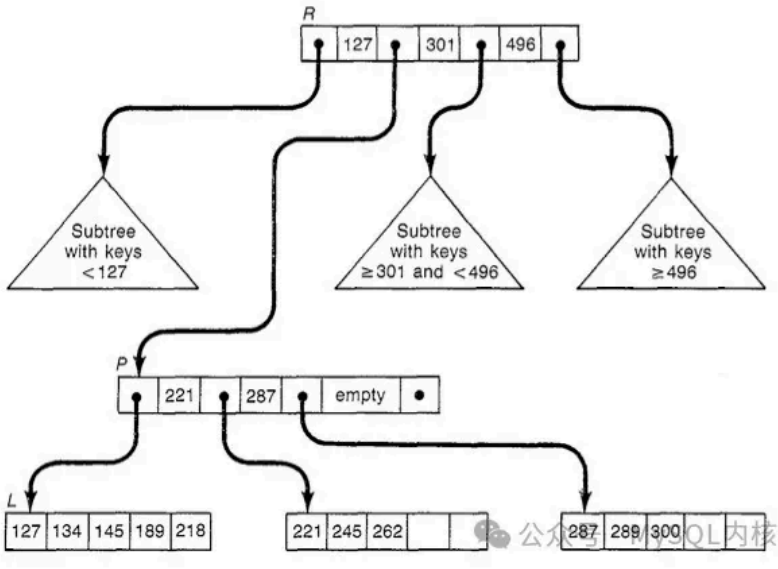
庖丁解InnoDB之B+Tree

原创 王康 MySQL内核剖析 2025年03月03日 13:51 浙江

InnoDB采用B+Tree来维护数据，处于非常核心的位置，可以说InnoDB中最重要的并发控制及故障恢复都是围绕着B+Tree来实现的。B+Tree本身是非常基础且成熟的数据结构，但在InnoDB这样一个成熟的工业产品里，面对的是复杂的用户场景，多样的需求，高性能高稳定的要求，以及长达几十年的代码积累，除此之外，InnoDB中的B+Tree在实现上并没有一个清晰的接口分层，这些都让这部分的代码显得复杂晦涩。本文希望从中剥茧抽丝，聚焦B+Tree本身的结构和访问来进行介绍，首先会简要介绍什么是B+Tree，之后介绍InnoDB中的B+Tree所处的位置和作用，然后介绍其数据组织方式，访问方式，以及并发控制。其余的，代码中交织在一起的诸如AHI (Adaptive Hash Index)、Change Buffer、RTree索引、Blob、代价估计等内容会先忽略掉。

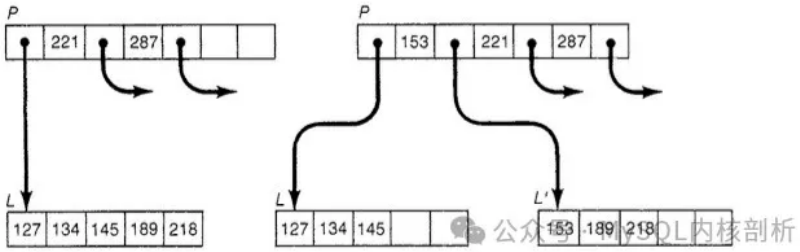
B+Tree

对MySQL这种磁盘数据库来说，当要访问的数据不在内存中的时候，就需要从磁盘中进行加载。而内存和磁盘的访问速度是有几千甚至上万的差距的，那么作为磁盘数据库的索引，能不能有效的降低从磁盘中加载数据的次数就变得非常重要。1970年，Rudolf Bayer《Organization and Maintenance of Large Ordered Indices》一文中提出了BTree[1]，之后在这个基础上演化出了B+Tree。B/B+Tree采用了多叉树的结构，显著的降低了数据的访问深度，尤其是B+Tree，通过限制所有的数据都只会存在于叶子节点，非叶子节点中只记录Key的信息，最大程度的压缩了索引的高度。这种索引结构的扁平化就意味着更少的磁盘访问，进而也意味这个更好得性能。因此，包括MySQL在内的大量主流的磁盘数据库都采用了B+Tree作为其索引的数据结构。如下图所示，是一个简单的B+Tree的示例：



B+Tree这个多叉树中的每个节点中包含一组有序的key，介于两个连续的有序key，key+1中间的是指向一个子树的指针，而这个子树中包含了所有取值在[key, key+1)的记录。所有对B+Tree内容的填、删、改、查都需要先对操作的元素定位，这个定位过程需

要从根节点出发，在每一层的节点中通过key的比较，找到需要的下一层节点，直到叶子节点。除此之外，为了方便遍历操作，叶子节点会通过右向指针串联在一起。

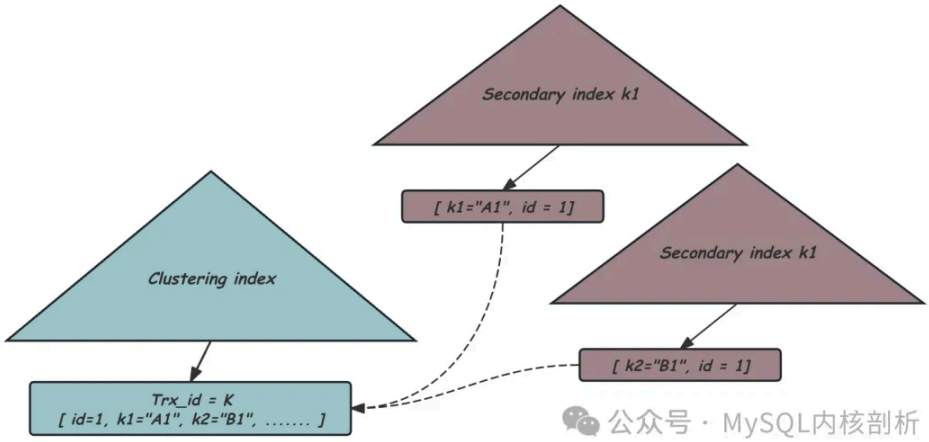


当叶子节点中的记录已经满的时候，新的插入会触发节点分裂，如上图所示，新的Key 153的插入导致原先已经满的节点L分裂成L和L'两个叶子节点。分裂会创建新的页面，将要分裂的页面的上部分数据迁移到这个新的节点上去，并将指向这个节点的指针及对应的边界key插入到父亲节点中去，同时，这次插入也有可能导致父节点的分裂，进而继续向上传导这个节点分裂动作。与之对应的是从B+Tree中删除数据后，叶子结点容量低于某个阈值（比如一半）时，会触发相邻节点的合并，同时需要从父节点中删除对应的Key及指针，因此合并操作也可能向更上层传导。所有的分裂和合并都是从叶子节点发起并向上传导的，只有根节点的分裂和合并会造成树高的变化。这也就保证了从Root到任意叶子的路径是相同的，从而保证了LogN的查找，插入以及删除复杂度。

InnoDB中的B+Tree

基于B+Tree的数据维护与访问

InnoDB中的每张表的都会维护一个包含所有数据的B+Tree，称为聚簇索引（Clustered Index）。通常聚簇索引这个B+Tree中的Key就是这张表的主键字段，需要保证唯一和非空，而Value就是完整的行数据。也就是说，表中所有完整的行数据是按照这个主键Key有序维护在这个聚簇索引这个B+Tree中的，因此指定一个有意义的不会频繁更新的主键字段是有好处的。除此之外，为了加速访问，通常会在表上创建一个或多个二级索引，每个二级索引也会对应维护成一个B+Tree，二级索引的B+Tree中的Key是这个索引字段，而Value是对应的聚簇索引上的Key。通过二级索引查找后，如果需要更多信息就需要用这个获取到的聚簇索引Key，再回到聚簇索引的B+Tree上做查找，这个就是我们常说的回表过程。如下图所示是一张MySQL表对应的数据结构，包含一个聚簇索引B+Tree，和多个二级索引B+Tree。

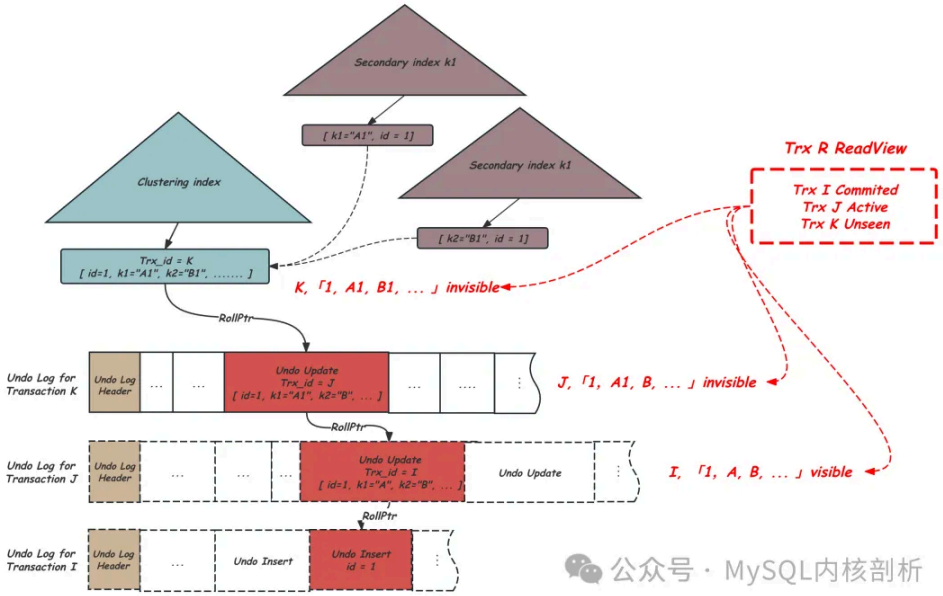


对MySQL数据库的增删改查请求，会在Server层通过优化器的代价估算，选择看起来最优的索引生成执行计划，最终转化为对对应的B+Tree的访问，从Root节点一路沿着B+Tree下降，这个过程中访问的Page如果不在Buffer Pool都需要先从磁盘中加载到Buffer Pool中，直到找到需要访问的Record所在的Page，并对其操作。

基于B+Tree的并发控制

数据库的并发控制是为了保证事务之间隔离性而设置的访问机制，《浅析数据库并发控制机制》[2]一文中曾经对常见的并发控制机制做过简单的介绍和分类。InnoDB中的并发控制采用的的**Lock + MVCC**的方式，也就是采用MVCC来避免读写之间的冲突，但在写写之间采用了悲观的Lock的并发控制方式。

InnoDB的**MVCC的实现**是以来Undo Log来做的[3]，在聚簇索引上除了用户数据外，还会记录一些隐藏字段，包括最近修改的事务ID，以及指向历史版本的Roll Ptr指针，指向Undo日志中该行的历史版本，快照读会根据配置的隔离级别持有ReadView，ReadView中记录获取ReadView时的活跃事务状态，通过对比聚簇索引上或UNDO Log上的事务ID和所持有ReadView中记录的活跃事务状态，就可以判断当前的版本是不是可见的，如果不可见就沿着Roll Ptr寻找正确的可见的版本。如下图所示，需要注意的是在InnoDB的实现中，只有聚簇索引B+Tree的记录中包含事务ID，这也导致二级索引上的可见性判断需求都必须回表到聚簇索引来满足，这也导致比如通过二级索引的查询、二级索引上的隐式锁判断、二级索引的Purge等场景的一些性能问题。



而对于**Lock的实现**，在《B+树数据库加锁历史》[4]一文中曾经介绍过，基于B+Tree的并发控制发展一直遵循着降低锁粒度的方向，直到ARIES/KVL将逻辑内容和物理内容分离，由Lock和Latch分别保护，并提供了一套相对完善的对Record Lock的实现算法，基本形成了现代B+Tree数据库的通用解法，InnoDB就是其中之一。InnoDB中维护了专门的Lock Manager模块来处理事务之间的加锁、阻塞、死锁检测等，这部分内容会在后续文章中做详细介绍。这里主要关注的是其跟B+Tree的关系，Lock Manager中的加锁对象可以简单的理解为聚簇索引或二级索引B+Tree叶子结点上的某个Key的物理位置。因此，在完成加锁之前，需要先在对应的B+Tree上定位需要的的记录。另外值得一提的是，InnoDB采用了文中提到的Ghost Record实现，也就是所有的删除操作，只会对记录设置Delete Mark标记，而将真正的删除操作推迟到后台Undo Purge中进行。从而将Delete操作转换为记录的原地Update操作，减少对区间Lock的需要。

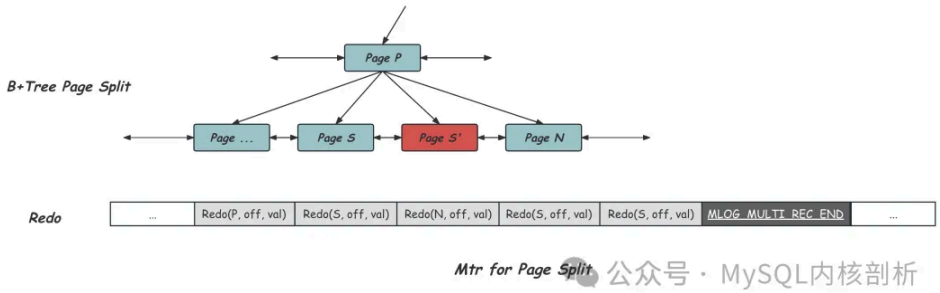
另外，依照ARIES/KVL[5]的设计，InnoDB中采用**Latch来保护B+Tree本身物理结构**在多线程访问下的正确，比如正在修改的Page不应该被其他线程看到，又比如一个分裂操作中的中间状态不应该被其他线程看到。在本文后面的并发控制章节会对这个部分详细讨论。

基于B+Tree的故障恢复

类似于并发控制，在故障恢复上InnoDB也区分了逻辑和物理两层[6]，简单的说，逻辑层的故障恢复需要保证的是在发生故障重启后，已经提交的事务依然存在，未提交的事务的任何修改都不存在，也就是我们常说的ACID中的Failure Atomic及Durability，在InnoDB中，这一点是靠Redo和Undo Log来实现的。本文这里主要关注的是跟B+Tree更相关的物理层的故障恢复，也就是如何保证在数据库重启后，B+Tree可以恢复到正确的位置，而不是一个树结构变更的中间状态。InnoDB中有一个很关键的数据结构：**Min-transaction（Mtr）**，也就是《B+树数据库故障恢复概述》[6]中提到的System Transaction的实现。

InnoDB中针对B+Tree结构变更的操作，例如节点的分裂或者合并，会涉及到多个兄弟节点以及其祖先节点的修改，Mtr中会持有需要的节点及索引锁，并且将这些修改产生的Redo Record记录到私有的Redo Buffer中，只有当所有这些修改都完成后，收集到所有Redo Record的Mtr才会进入Commit阶段，这时会在这组Redo的末尾添加

MLOG_MULTI_REC_END的特殊类型的日志，等待这些Redo Records从私有的Redo Buffer中连续拷贝到全局Redo Buffer并写入磁盘。在这之后该Mtr才能完成Commit，对应的脏页才能落盘。如下图所示，Page S分裂成S和S'的过程同时造成了父节点P，兄弟节点N的修改，这个过程对应操作被记录在连续的Redo日志中：



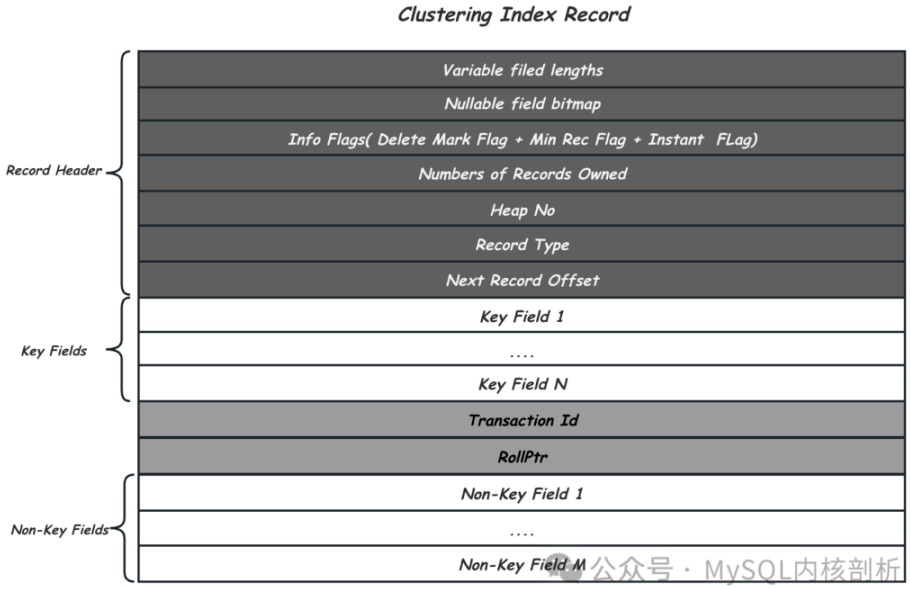
当发生故障重启的时候，在真正做Redo重放之前，Redo Parse过程会先尝试去找MLOG_MULTI_REC_END日志，如果看到这个标记，那么这段连续的Redo才能被重放，否则这个Mtr中的所有redo都不能重放。从而保证在重启后，这个B+Tree结构变更导致的多节点的修改，要不都存在，要不都不存在，以此来保证B+Tree本身结构的正确。

B+Tree的数据组织

B+Tree是很基础的数据结构，但通常在学术上讨论的时候为了简化，树上维护的数据项都会限定为简单的定长Key Value，这使得实现上对数据的访问，以及判断否需要分裂或合并等操作都非常容易，但对面向现实需求的存储引擎InnoDB来说，这样显然是不够的，再加上上面所讲到的并发控制、故障恢复的需要，InnoDB中的B+Tree数据组织上会显得复杂不少。

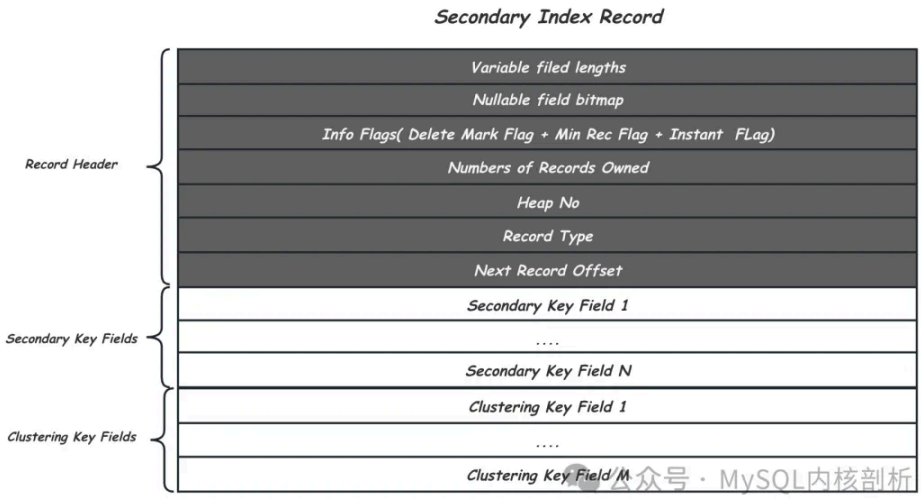
B+Tree中的数据项：Record

前面讲到，InnoDB中用B+Tree来维护数据，对数据库来说就是一行行表内容。当用户创建一张表的时候，会在SQL语句中指定这张表中的每一行包含那些列，每一列的类型和长度，是否可以为Null，这些信息都会记录在MySQL的数据字典（Data Dictionary）中，本文不对这里展开讲述，这里只需要知道，数据字典的些列及列长信息，会在对InnoDB的B+Tree进行写入或者读取的时候，通过dict_index_t的数据结构传递下来，并以这个格式进行每一行数据Record的序列化及反序列化，这里只介绍5.1后已经默认的Compact记录格式，以聚簇索引中记录的完整Record为例，一行用户数据序列化后会维护成如下格式：



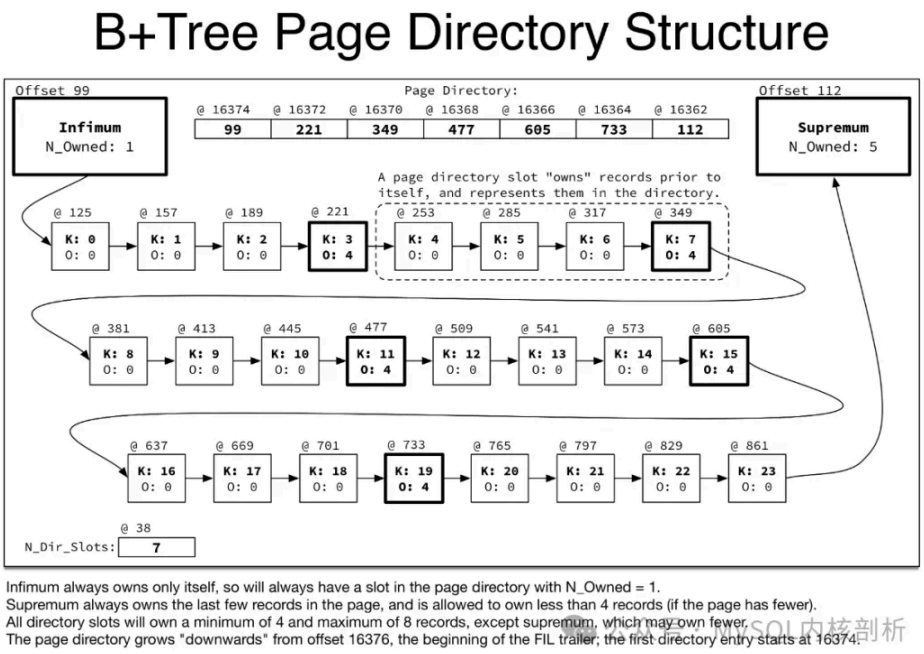
可以看到，其中包括Key字段以及Non-Key字段，之所以这里Key也会有多个，是因为可能有组合索引的情况，事务ID及Rollptr作用在上面介绍过，用来对读提供MVCC访问。除此之外，还有一个**Record Header**，其中记录了一些访问过程中需要的元信息，依次是变长列的长度数组Variable Field Lengths，维护了所有变长列在当前Record的真实长度；Nullable Field Bitmap为允许Null的字段在当前Record是否为Null；一个1字节的Info Flags，截止8.0包括用作标记Record删除的Delete Mark Flag，用作标记是否是非叶子节点层最小Record的Min Rec Flag，以及标记是否为Instant Add Column之后加入的Instant Flag；Numbers of Records Owned用作下一节要介绍的B+Tree Page内数据维护，Heap No是该Record在其所在的Page上的序号，也是事务锁的加锁对象；Record Type标记Record的类型，包括叶子结点数据、非叶子节点指针等；最后是下一条Record偏移的Next指针，通过这个指针可以将Record串成链。

上面介绍的是记录完整的行数据的聚簇索引B+Tree中的Record格式，而对二级索引B+Tree中的Record格式略有区别，如下图所示，主要区别在于，其中的Key是建表是定义的二级索引列，其中的Value是对应的聚簇索引列，并且由于二级索引不维护MVCC信息，这里没有事务ID和RollPtr：

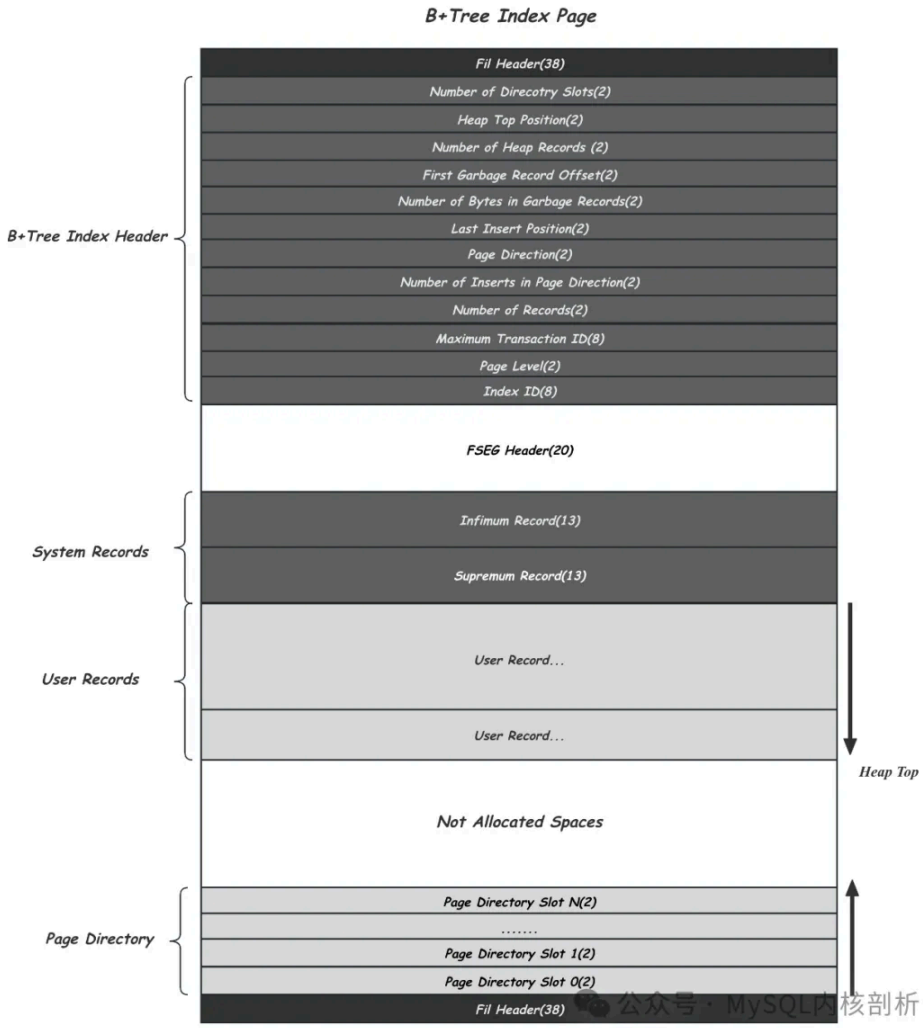


B+Tree中的节点：Page

InnoDB中B+Tree中的每一个节点，对应InnoDB中的一个默认16KB大小的Page，一个这样的Page中通常会维护很多的Record，这些Record通过上面介绍过的Record上的Next指针在Page内串成一个单向链表，除了真实的用户Record外，为了实现逻辑的简单，类似于我们通常处理链表问题时候采用的“哨兵”优化，InnoDB的B+Tree Page中也预留了两个固定位置固定值的System Records，在链表头的Infimum记录以及在链表尾的Supremum记录。由于B+Tree Page上维护的Records并不是定长的，无法通过Key方便的在页内做直接的定位，每次都做遍历查找的开又太大，因此InnoDB在Page内还维护了一个Record目录（Directory）来加速检索，这个Directory由多个连续存放的定长的Slot组成，每个Slot占用两字节，记录其指向的Record的页内偏移。综上一个B+Tree Page中的内容会如下图所示：



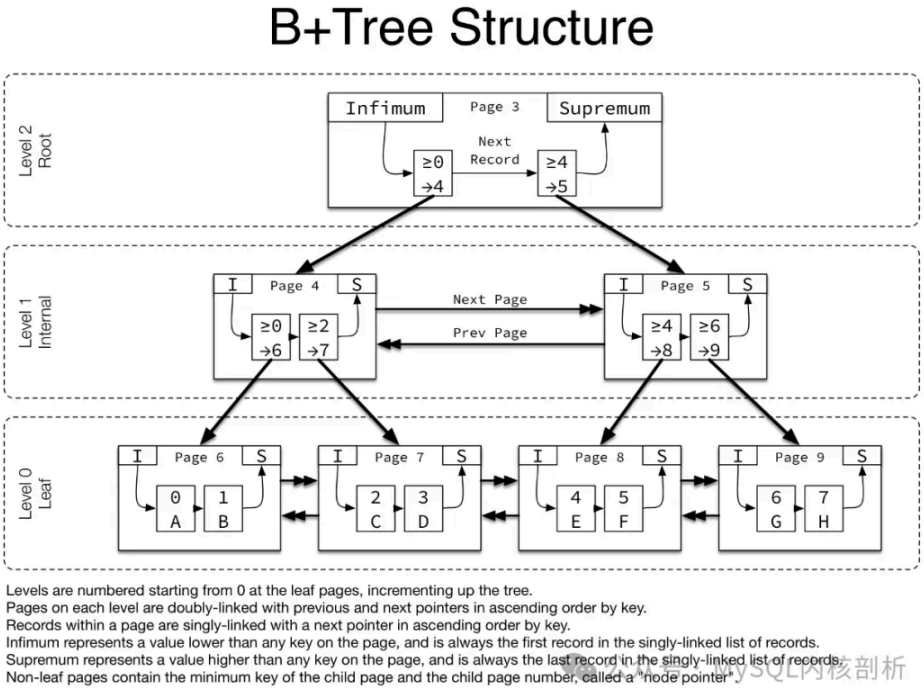
图中最上面所示的Page Directory中的每个Slot指向一个标粗的Record的页内偏移，开头和结束分别是Infimum记录和Supremum记录，两个这样的Record中间的Record称为被这个Slot Owned，一个Slot Owned的Record个数会记录在其指向的标粗的Record Header上的Numbers of Records Owned中。在插入和删除的过程中，这些Slot会动态的进行分裂或平衡，来保证除了最后一个Slot外，所有的slot所owned的Record在4到8之间。由于这些这些Directory slot本身是定长的，在做页内的Key查找的时候就可以很方便地通过这些连续的Slot数组来实现二分查找，找到其所Owned的这一组4到8个Records，再做顺序查找就大大提高了页内查询的效率。由于Page Directory中的Slot需要连续，并且随着Record的插入也需要不断的扩展，因此在实现上，不同于Record在页内自上而下的生长方式，Page Directory是从页尾向上的扩展的，二者中间的部分就是页上的空闲区域，如下图所示：



除了上面已经介绍过的System Record、User Record、Page Direcory以及他们之间的的未分配空间外，每个B+Tree的还会有一个定长的Index Header和一个定长的FSEG Header信息，其中FSEG Header只在B+Tree Root节点上有用，用户维护B+Tree的落盘结构，这部分我们放到最后一节的文件组织中介绍。这里我们主要关注Index Header，其中维护了很多元信息，依次为：Directory Slot的个数；User Record自上而下增长的位置Heap Top，这里是未分配空间的开头，新的记录都是从这里分配的；当前Page中曾经从Heap Top上分配出去的Record编号，每次新分配会加一，来作为这个Record位置的唯一标识记录在Record Header的Heap No中；除此之外，由于Record是可能删除的，Page还维护了一个简单的Free List，First Garage Record Offset就是这个链表头，紧接着的是这个Free List链表的元素个数，不过这个空闲链表得实现相当简单，当新的Record写入时，仅仅会尝试链表头的那个Record位置是不是够用，够就摘下来复用，不够就从Heap Top的位置上新分配；接下来的Last Insert Position维护的是上一次插入的的记录，为的是在页分裂的时候可以做一些启发式的规则，来避免顺序导入数据场景下，默认一半一半分裂的策略会造成的空间浪费；之后Page Direction是上一条记录插入的方向以及这个方向上连续插入的个数；当前Page的有效的记录数；修改当前Page的最大事务ID，这个值通常只在二级索引Page上有用，用作一些类似二级索引回表场景的过滤条件，以优化二级索引没有MVCC信息带来的回表开销；Page Levle是当前Page在B+Tree上的层级，叶子节点是0；最后是所属的Page所属的B+TreeIndex ID。

在开头的38个字节的Fil Header里除了当前Page的Checksum及Page LSN之外，还会有两个链表指针，在B+Tree中指向其兄弟节点的页编号，通过这个指针，每一层的

Page都会组成一个横向的Page双向链表。同时，前面我们还讲过，B+Tree的非叶子节点中只会维护Key而没有Value，对非叶子节点来说，他的Value就是以这个Key为最小值的子节点页编号。最终一个B+Tree上的所有Page会通过纵向的子节点指针和横向的兄弟节点指针串成一个如下图所示的B+Tree：



B+tree的访问

我们已经知道MySQL的所有的增删改查操作，经过Server层到InnoDB层，都会转换为一次次对表的聚簇索引或二级索引的B+Tree的查询、修改、插入及删除。这种从对行数据的操作到对B+Tree的操作的转换，并不总是一一对应的，主要包括如下三方面的因素：

1. **行到Key Value的转换因素：**
   对行数据的修改操作，会造成行数据中某些或全部Field的变化，而这些Filed对于B+Tree来说，有些属于Key有些数据Value，当修改涉及到属于Key的Filed时，修改前后的行对于B+Tree来说其实是两条不同的记录，因此这次Update最终会变成B+Tree上老的Record的删除和新的Record的插入。
2. **二级索引的实现因素：**
   上面在数据组织章节介绍过，InnoDB中的二级索引B+Tree的Value其实是其对应聚簇索引上的Key字段。从实现复杂度，尤其是对Lock的维护的复杂度上考虑，InnoDB不会对二级索引B+Tree上的Value做任何的修改，即使这个聚簇索引Key包含多个Field。如此一来，二级索引上的所有修改都会转换成B+Tree上对原来记录的删除和一次新记录的插入。
3. **Ghost Record(Delete Mark)的因素：**
   InnoDB采用了《B+树数据库加锁历史》[4]一文中提到的Ghost Record实现，也就是删除操作只会对记录设置Delete Mark标记，而将真正的B+Tree上的记录删除推迟到后台Undo Purge中进行。而这个Delete Mark标记的引入，导致对Record的

Delete和Insert操作可能会变成对Delete Mark标记的修改。
无论如何，这些操作最终还是会转换到B+Tree上的查询、修改、插入或删除，本章节还是站在B+Tree的角度上，来看看这一过程的实现，由于B+Tree上的Value都只存在在叶子节点上，因此所有的操作，第一步都需要先在B+Tree上完成对这个Key得定位。

B+Tree的定位

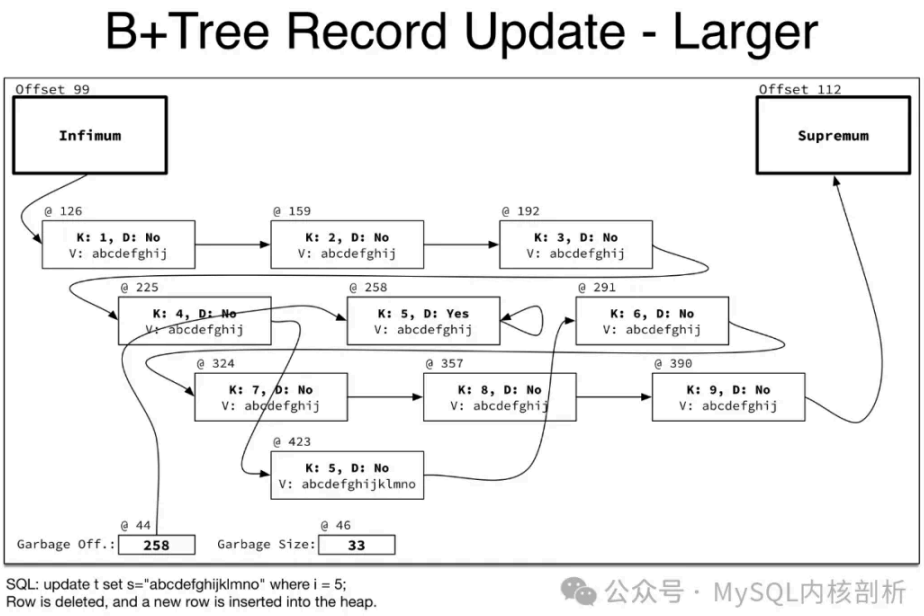
这个定位过程的逻辑主要在**btr_cur_search_to_nth_level**中，这个是个非常长且晦涩的函数，这里我们先忽略RTree、AHI、Insert Buffer、Blob以及下一章节将要介绍的并发控制相关内容后，其逻辑其实是非常简单的：首先，这个函数的调用者会指定要查找的Key Field值以及Search_Mode(大于、大于等于、小于、小于等于)；然后从描述这个B+Tree的数据字典元信息dict_index_t中，获得这个B+Tree的Root Page Number，通过**buf_page_get_gen**去获取这个这个Page，如果不在Buffer Pool会先从磁盘中加载进内存；之后，通过**page_cur_search_with_match**去做页内的搜索，包括根据Directory Slot的二分查找，以及定位到对应Slot后对其Owned Records的顺序遍历，找到需要的Key范围，得到下一层的Page Number，重复上面的这个buf_page_get_gen + page_cur_search_with_match过程，直到找到满足Search_mode的叶子节点上的Record位置。这个位置信息会维护在一个**btr_cur_t**结构中，供调用者使用，也可能固化在当前线程的**btr_pcur_t**结构中，btr_pcur_t中的除了包括位置信息外，还包括一个获得这个cursor时候的版本号，通过这个版本号，后续的访问只要发现这个Page没有改动就可以直接使用这个位置信息，避免重复的B+Tree定位过程。后续的操作都会基于这个位置信息Cursor做操作，其中读取是最简单的，直接获取Value信息即可。

B+Tree的修改、插入、删除

InnoDB中对B+Tree的修改、插入以及删除操作都会存在乐观和悲观两个版本，乐观版本假设本次操作不会导致树结构的变化，因此持有较轻量的锁，如果失败，那么就获取更重的锁，再通过悲观版本来完成变更。这里得修改包括对非Key Filed的修改，也包括对Delete Mark的修改，一次Update的过程如下：

- **btr_cur_optimistic_update**
  会比较要修改的Record的老值和新值占用的空间大小，如果新值更小，那么简单的通过**btr_update_in_place**在当前位置直接更新，并接受Record变小带来的碎片。
- 如果新值需要的空间更大，那么就需要先在Page上删除老的Record，再插入新的Record，这时会先计算删除后是否有空间插入新的的Record，如果能，那么通过**page_cur_delete_rec**删除，之后再通过**btr_cur_insert_if_poossible**再次插入就好。
- 但如果Page上无法放下新值，那么就需要返回失败，并在加更重的锁之后通过**btr_cur_pessimistic_update**来完成，这里先page_cur_delete_rec完成删除，然后通过**btr_cur_pessimistic_insert**来做悲观插入，其中可能需要先完成B+Tree的节点分裂甚至是树层数的增高。

需要指出的是这里的删除和插入的Record的Key是完全一致的，对B+Tree来说他们还是同一条记录，只是由于空间变化问题换了个位置，新的Record会继承老Record的MVCC需要的Undo版本链，以及可能存在的记录锁。这个过程中触发的Page内的Record插入或者删除，都会维护上面所讲到的页内Record链表，Free List以及Directory Slot，如下图所示，是一次将Record修改成占用空间更大的值的过程，由于原来的Record空间不足以存放新值，因此需要先将老的Record空间删除，之后再插入一个新的存放了新值的Record，同时老的Record被加入到了Free List（Garbage）中。



对于插入操作而言，同样先乐观的通过**btr_cur_optimistic_insert**尝试插入，如果Page内空间充足，那么通过**page_cur_tuple_insert**完成页内插入，否则返回失败，之后通过**btr_cur_pessimistic_insert**来做悲观插入，悲观插入会带来节点的分裂，甚至树层数的升高。类似的，删除操作会先乐观地通过**btr_cur_optimistic_delete**来删除，其中需要先采用**btr_cur_can_delete_without_compress**判断删除后是否需要触发Page合并，也就是Page中的剩余Record值是否小于了阈值。如果不会，那么通过**page_cur_delete_rec**完成删除。否则，需要返回失败，并在之后通过**btr_cur_pessimistic_delete**做悲观的删除，也就是在删除后触发Page的合并，甚至是树层数的降低。

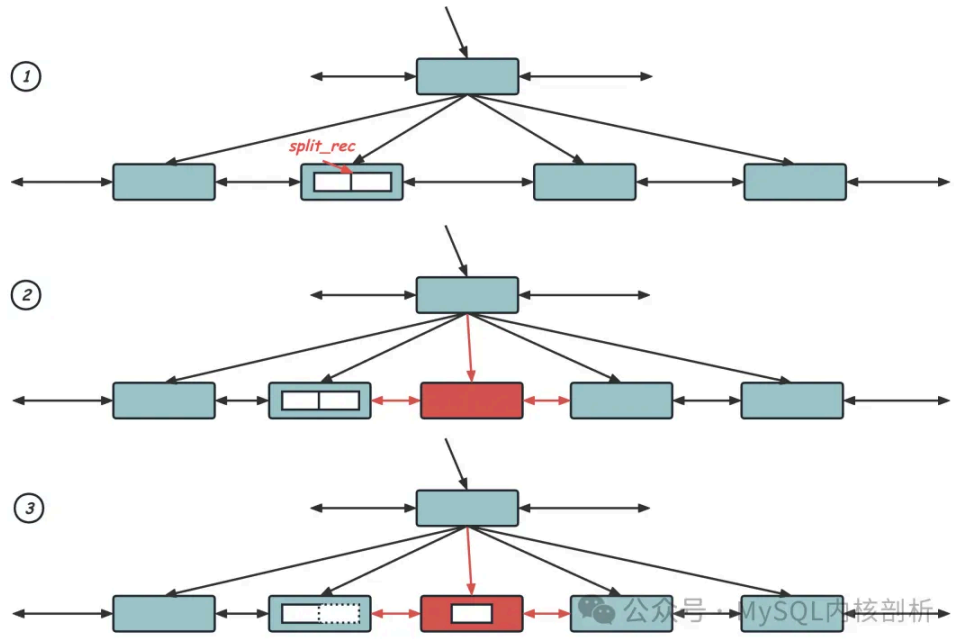
B+Tree的节点分裂及树高度上升

当前Page的空间无法放下要插入的Record的时候，就需要触发当前Page的分裂，由于B+Tree的所有Value都只存在于叶子结点，因此，通常情况下节点的分裂都是从叶子节点的分裂发起的，叶子节点分裂后，新的节点插入到父节点中，导致级联的上层Page分裂。除此之外还有一种情况，就是下层节点删除第一项，会导致当前节点的边界Key发生变化，那么就需要在其父节点中删除对应老的Key，并插入新的Key，这个插入动作也是有可能导致父节点的分裂的。

节点分裂的实现代码主要在**btr_page_split_and_insert**中，第一步是要（1）确定split_rec，也就是从那个位置开始把之后的Record移动到新分裂出来的Page上去，一般情况下，这个split_rec会选择当前Page中间的那个，也就是分裂后的两个兄弟节点每

人负责一半的数据。但这样一来，在主键Autoinc导入数据这样的场景中，插入的数据是有序递增的，那么前面的一个Page就永远只有一半的数据，整个B+Tree的空间利用率就变成了50%。为了尽量避免这种情况，InnoDB中采用了简单的启发式规则，这就要用到前面我们再Page格式中介绍的，Index Page Header上的信息：Last Insert Position，基本思路是如果这个Page上的两次连续的Insert是刚好有序的两个Record，那么很有可能当前是一个顺序插入的场景，那么就不要选择中间位置分裂，而是直接从这个插入的位置来做分裂。

确定了split_rec之后，接下来的分裂操作就比较简单了，首先（2）从文件上分配并初始化一个新的Page，并通过**btr_attach_half_pages**将这个新Page加入的B+Tree中，包括向父节点中添加这个新节点的Key及指针，以及将这个Page加入同一层的兄弟节点的双向链表中，注意向父节点的插入操作可能触发级联向上的Page分裂；之后（3）将前面找到的split_rec后面的Records拷贝到新的Page上去，并在老的Page中删除；(4)最后再完成新Record的插入即可，这个插入有可能插入新节点也有可能插入老节点，取决于前面split_rec的选择。这里实现上有一个小细节，在选取spilit_rec的时候第一次直接选中间节点可能会导致分裂后还是放不下，那么会触发第二次分裂，而这一次会参考要插入的rec大小和Page内当前的Record大小，来选择一个分裂后一定能放下的split_rec，也就是btr_page_split_and_insert中一个节点的分裂是有可能分裂成三个节点的。整个分裂过程如下图所示：



在整个B+Tree都很满的情况下，这种从叶子节点发起，一路向上的级联节点分裂，有可能一路传到到Root Page，造成Root Page的分裂，而Root Page分裂会带来整个B+Tree层数的变高。这个过程的实现主要咋**btr_root_raise_and_insert**中，由于Root Page头中有一些特殊的信息，并且这个位置是记录在数据字典中的，我们不希望Root Page的分裂带来这些信息的修改，因此这里采取了保留原Root的Page的方式：首先分配一个新的Page，将Root上的所有记录都拷贝到这个New Page上，然后清空Root Page并将这个New Page挂到Root Page上作为其当前唯一的叶子节点，之后同样调用btr_page_split_and_insert来完成一次常规的Page分裂过程。

B+Tree的节点合并及树高度降低

从Page上删除Record的时候，如果造成Page上的空间占用低于一个阈值merge_threshold，那么就会触发节点合并，这个阈值默认是50%，当然也可以对整个Table或者单个索引通过设置COMMENT='MERGE_THRESHOLD=40'来修改为1到50中间的任何一个数。大多数的节点合并都由叶子节点上删除数据触发叶子结点的合并开始，如果需要删除父节点中的元素，就可能触发级联的上层节点合并。但有一种例外，是在做节点的插入操作时，如果当前节点无法放下并且是最后一个Record，那么就会首先尝试**btr_insert_into_right_sibling**插入到右边的兄弟节点，这个新的插入就会是右侧兄弟节点的第一个Key，而修改第一个Key就需要先从父节点中删除原先对应这个节点的Key，然后插入新的，这个删除操作同样有可能会触发父亲节点进而级联向上的节点合并。
节点合并的操作通常主要在**btr_compress**中完成，其中首先会通过**btr_can_merge_with_page**依次判断其左右兄弟节点是否足以放下要合并节点的所有记录，如果可以放下就进入正式的合并过程：将当前Page中的所有记录拷贝到要合并的兄弟节点中去，将这个节点从兄弟节点的双向链表中移除，并更新或删除父节点中的对应Key或指针，这个过程可能会触发级联向上的Page合并。
但如果当前要合并的节点已经没有左右兄弟节点，也就是说这个节点是B+Tree上当前Level的最后一个Page，那么就需要将当前节点与其父节点进行合并，也就是将当前的节点上升一层，这个过程在**btr_lift_page_up**中，首先将这个父节点清空，然后将原来节点的所有内容拷贝到其父节点中，之后将这个原来的节点Free掉即可，从而也导致整个B+Tree高度的降低。

并发控制

回顾之前讲到的，B+Tree数据库的并发控制一直沿着更小的加锁粒度的方向发展，从对整个树加锁的2PL，到只对两层节点加锁的Lock Coupling，再到只对当前节点加锁的Blink，最后ARIES区分了逻辑层和物理层，将保护B+Tree结构在多线程之间完整的Latch和保证事务之间的隔离性的Lock解耦开来。本章节关注的就是InnoDB中B+Tree上的Latch这一层的实现，在Latch这一维度，从整个树加Latch，到对两层节点加Latch的Latch Coupling，再到只对当前节点加Latch的Blink的优化方向也是同样适用的。InnoDB中的B+Tree的Latch并发控制的的发展也确实是演这个这个方向进行的。比如在5.7之前，采用的就是类似锁整个Tree的方案，而5.7和8.0对这个锁粒度做了进一步的优化， 这里我们还是以8.0为例来进行介绍。
对B+Tree多线程并发访问影响最大的其实是B+Tree结构的变化，也就是B+Tree节点的分裂、合并，以及B+Tree层数的升高或将低，也统称为SMO（Structure Modification Operation），试想如果没有SMO，也就是B+Tree树结构保持稳定不变，那么B+Tree的多线程并发访问的实现，将是非常的简单高效的：所有的读写都可以无锁的在B+Tree上做记录的搜索定位，直到找到需要读写的记录所在的叶子节点，然后对这个节点加读写锁操作即可。当然这么理想的情况在现实中通常是不存在的，但考虑到SMO相对于记录的读写可以认为是极少的，因此在实现上InnoDB采用了**先乐观再悲观**的写入方式：

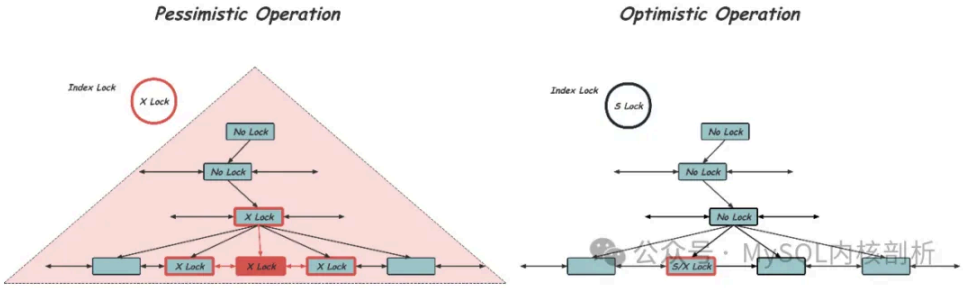
- 先假设本次修改不会导致树结构的变化，持有较轻量的锁：

- 如果发现需要造成树结构的变化，也就是上一节提到的插入发现Page放不下需要分裂，或者删除发现小于merge_threshold需要合并
- 那么退出乐观过程，再加较重的树结构锁，也就是悲观的方式重新完成写入。

这也是代码中看到B+Tree的insert、update和delete都会有optimistic以及pessimistic两个版本的原因，比如btr_cur_optimistic_insert和btr_cur_pessimistic_insert。上一个章节介绍了， InooDB中对B+Tree的访问包括读、插入、修改、删除四种，其中读取和乐观的写（插入、修改、删除）的加锁逻辑的类似的，区别只在搜索到最后的叶子结点后，对这个叶子节点加读锁还是写锁而已。因此，这里我们对B+Tree加锁的讨论将分为**乐观操作和悲观操作**来进行，其中乐观操作包括读和乐观写。这里Latch的加锁策略需要保证：

1. 任何乐观操作不应该看到悲观操作导致的树结构变化的中间状态；
2. 任何悲观操作不应该基于其他悲观操作导致的树结构变化的中间状态。

InnoDB的Latch实现主要包括两把锁，保护整个树结构的大锁index->lock，以及保护每个节点的block->lock，注意这两个锁名称虽然叫做lock，但跟上面讲的保证事务之间隔离性的Lock是没关系的，他们两个其实就是我们理论上所说的线程间同步的Latch的实现，是InnoDB中的读写锁rw_lock_t类型，加锁类型包括s lock、x lock及sx lock三种，也就是除了读写两种模式外，增加了一种不跟读锁互斥的sx lock。上一章节介绍过，对B+Tree的所有访问都需要先通过**btr_cur_search_to_nth_level**做记录在B+Tree上的搜索定位，实现上对index->lock及block->lock的加锁也都是在这个函数内完成的。为了实现上面的互斥保证，最简单的，也是InnoDB在5.6及之前版本的实现方式，如下图所示：



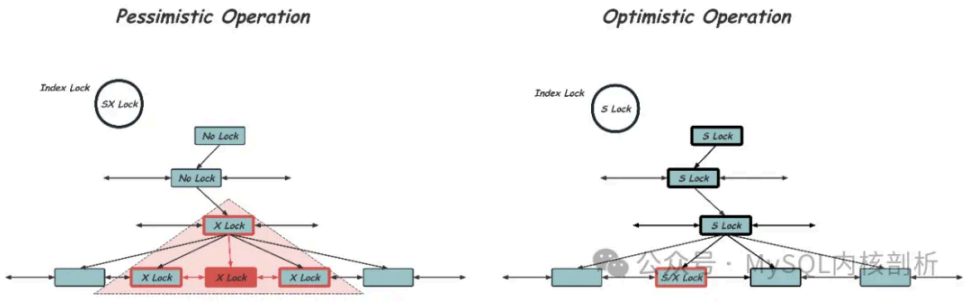
就是乐观操作对index->lock加slock，悲观操作对index->lock加xlock，然后无锁的搜索到叶子节点，之后根据读写类型对这个叶子结点加xlock或者slock即可。显然这种实现的瓶颈是很明显的，一次悲观操作带来的树结构变化，会阻塞整个B+Tree的读写。《B+Tree加锁历史中》提到的Tree Protcol(Latch Coupling)为解决这种树级别的加锁粒度问题提供了一种思路：利用B+Tree从上到下的搜索方式，持有上层节点的lock来对下层节点加lock，之后释放上层节点的lock并重复这个过程，就像两脚交替下梯子一样。InnoDB从5.7开始对这里也做了优化，但遗憾的是，可能由于工程实现上的复杂度，并没有完全的实现到Latch Coupling这一步。

从锁Tree到锁Subtree

InnoDB在5.7及以后的版本中，首先将悲观操作对index->lock的xlock改为sx lock，保留了悲观操作之间的互斥，同时允许SMO过程中持有slock的乐观操作并发访问，如此一来带来的问题就是，如何保证乐观操作不看到悲观操作导致的树结构变化的中间状态？InnoDB的答案是通过子树根节点的互斥访问实现锁子树，其实现过程如下：

1. 对于乐观操作，**btr_cur_search_to_nth_level**中会获取index->lock的slock，然后在向下搜索的过程中，对经过的每一个节点的block->lock获取slock，直到到达叶子节点获取对应的读锁或写锁，之后释放index->lock的slock，以及上层节点的slock。
2. 对应的悲观操作会先获得index->lock的sxlock，并且需要对所有发生树结构变化的节点加写锁，上一节中提到节点的分裂合并是可能级联向上的，因此这里可能需要对多层祖先节点加写锁。实现上这里有个矛盾：在开始做SMO之后才能知道会级联到哪一层的节点，但在做SMO之前就需要持有所有影响节点的block->lock的xlock。因此就需要对哪些Page会受影响提前做一个预估，InnoDB中这个实现在函数**btr_cur_will_modify_tree**中，悲观操作在**btr_cur_search_to_nth_level**搜索过程中会对每一个经过Page通过**btr_cur_will_modify_tree**做这个判断。其实在理论上在B+Tree上做这个判断是相对简单的，举个例子，当要做一次插入操作的时候，在经过祖先节点的时候，至少要保证这个祖先节点的剩余空间可以放下一个新的元素（InnoDB实现上由于存在分裂成三个节点的情况，这里需要保证两个元素的空间），否则就存在SMO最终会级联到当前这个节点的可能。可以看出，这里是一种很严格的判断方式，也就是本次操作只要有一丁点发生SMO的可能，就需要提前获取这个Page的xlock。为了尽量减少对乐观操作的影响，btr_cur_search_to_nth_level在搜索过程中并不会对中间节点加任何锁，而是在一个中间数据结构tree_blocks中记录所有**btr_cur_will_modify_tree**判断为true的节点，当到达level=1的节点，也就是叶子结点的父节点时，会对仍然在tree_blocks数组中的所有节点加xlock，并且在搜索到叶子结点后对叶子节点及前后兄弟节点加xlock。

综合上面对乐观操作和悲观操作的加锁处理，可以看出悲观操作和乐观操作其实是通过子树根节点上的读写锁实现整个子树的互斥访问，如下图所示：



当然这样的实现还是会有严重的瓶颈的。首先，悲观操作互相互斥，导致同一时刻无论多大的B+Tree都只能有一个SMO发生；其次，乐观操作和悲观操作之间，由于btr_cur_will_modify_tree必须做严格的判断，导致大多数情况下会block一个比需要更大的子树范围；最后，乐观操作持有了全搜索路径的s lock也限制了需要xlock的SMO悲观操作。这些其实都是Tree Protocol（Latch Coupling）可以迎刃而解的。那么有没有可能在InnoDB中做到这一点呢，答案是肯定得，阿里云的PolarDB中就做了相应的

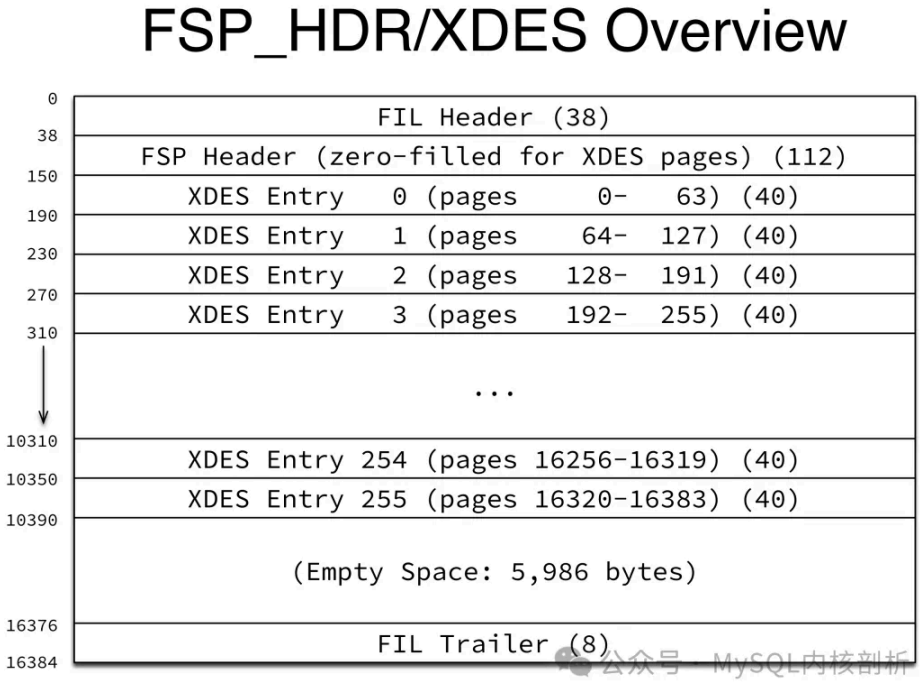
实现，并且更进一步将这种锁优化推进到接近只需要锁当前节点的Blink实现。《路在脚下, 从BTree 到Polar Index》[7]一文中中有详细的讨论，这里就不在赘述了。

文件组织

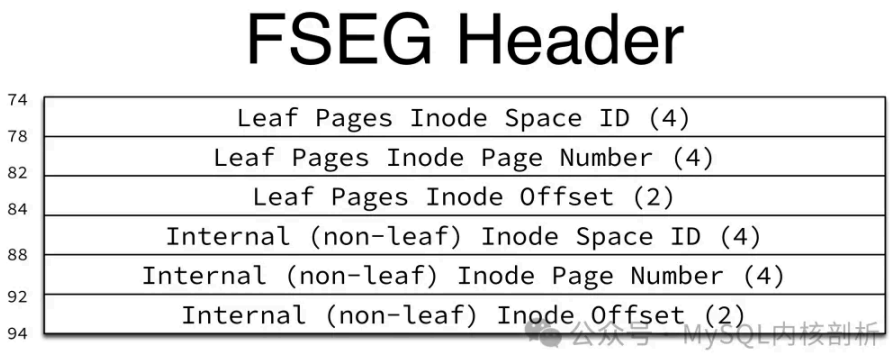
InnoDB数据最终是存储在磁盘上的文件中的，通常会设置innodb_file_per_table来让每个用户的表独占一个数据IBD文件，那么一张表中的聚簇索引B+Tree与所有可能存在的二级索引B+Tree都会共享这样一个文件，
我们上面讲到的B+Tree在创建删除、或者节点的分裂及合并的过程中，会不断的从文件中分配或者归还Page，IBD文件也会随着B+Tree中数据的增加而向后扩展。一个IBD文件内，不同的B+Tree，或者同一个B+Tree中的叶子或者非叶子节点，他们内部在访问上是有相关性的，比如一些Page在被访问的时候，很大概率意味着其兄弟节点也很快会被访问到，而磁盘的特性是顺序访问的速度远远的好于随机读写的速度，因此一种良好的设计思路是让逻辑上相关的Page在物理上也尽量的连续，这样有相关性的Page访问的时候就会有更大的概率以顺序读写的方式来进行IO，从而获得高效的读写性能。为了实现这一点，需要两部分的工作：

1. 需要一种连续Page的元信息维护方式。
2. 需要一种按照逻辑相关性尽量分配连续Page的方法。

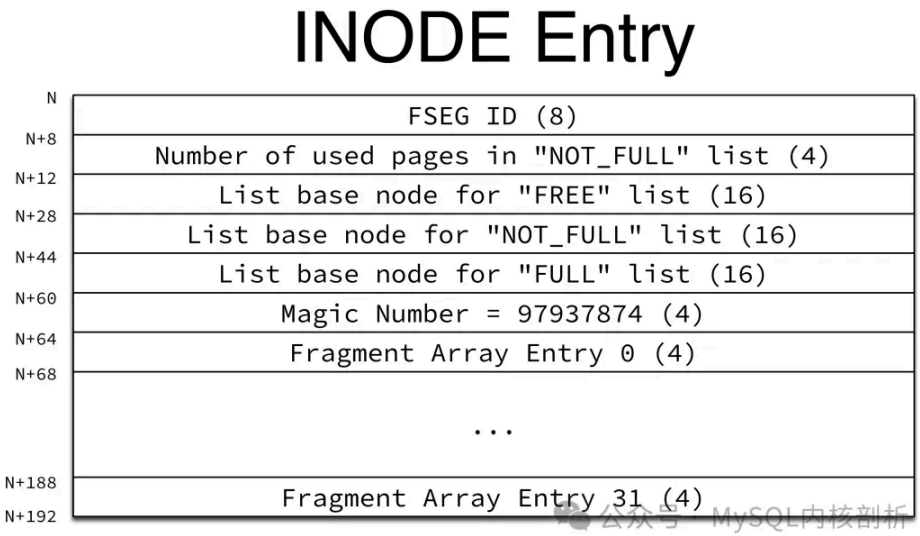
对于第一个问题，以16KB的Page大小为例，InnoDB中将连续的64个Page也就是连续1MB的空间划分为一个Extent，每个Extent都需要一个XDES Entry来维护一些元信息，其中主要包括这个Extend的是空闲、分配了部分还是全部分配的状态，以及其中每一个Page是不是被使用的Bitmap。这个元信息需要能方便的找到，并且随着文件的向后扩张，Page不断的增多，有足够的位置来进行存储。InnoDB的实现是每隔256MB就在这个固定位置放一个特殊的XDES Page，其中维护其向后256个Extent的元信息。这个Page的格式如下：



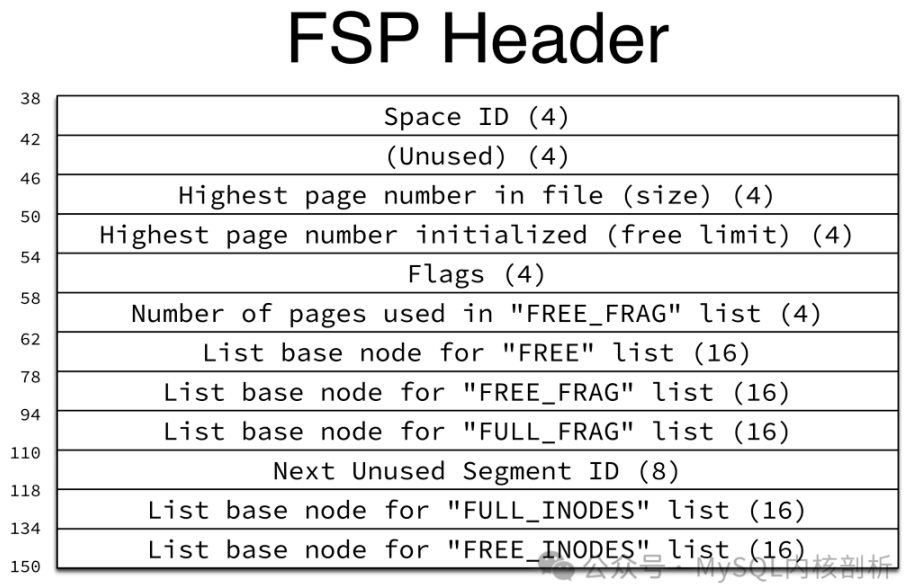
对于第二个逻辑相关的问题，在InnoDB中维护了Segment的概念，这是一个逻辑概念，对应的是一组逻辑上相关的Page集合，每个Segment会尽量的获取一些连续的Page（Extent）并持有，当这个Segment中得节点想要分配Page的时候，就优先从这些连续的Page中来获得。目前的InnoDB实现里，对每一个B+Tree维护了两个Segment，一个负责叶子节点，一个负责非叶子节点。这部分信息维护在B+Tree的Root Page中，上面介绍B+Tree的Page结构的时候，遗留的最后一段FSEG Header就是这个作用，只是这个信息只有在Root Page上有用，非Root节点上是空闲的，Root Page多出来的FSEG Header中就维护了当前的B+Tree所持有的的两个Segment信息：



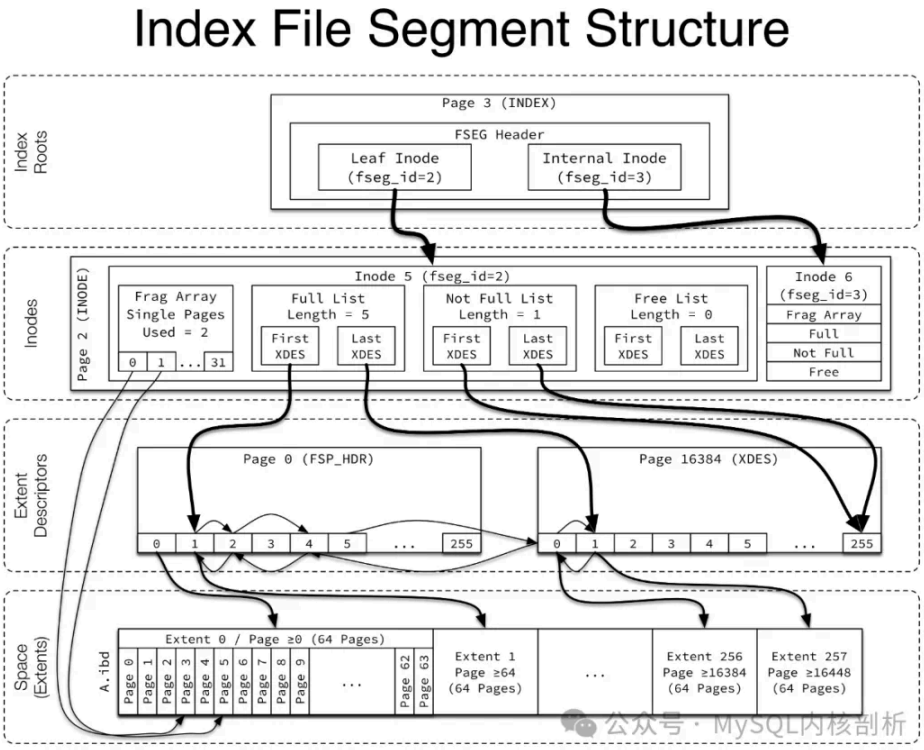
可以看出，这里对每个Segment其实都是用SpaceID + PageNO + Offset唯一锁定了一个叫做Inode Entry的文件偏移，通常一个IBD文件的第三个Page就是Inode Page，其中维护了最多84个Inode Entry，也就是最多84个Segment，绝大多数情况下都是足够的，当然如果不足就需要动态分配新的Inode Page。Inode Entry如下图所示：



可以看出，其中维护了一组其持有的Extent的链表，包括完全空闲、部分分配，或全部已分配三个链表，除此之外，还有32个碎片Page的位置，这个其实是一种分配策略上的权衡，一个连续Extent有64个Page，如果每个Segment最少都分配一个Extent，显然会有极大的空间浪费，因此对于小的Segment会优先按照单个Page的方式进行分配，直到这个Segment得Inode Entry里的32个碎片Page的槽位已经用满。而文件的第一个Page，FSP_HDR作为一个特殊的XDES Page，除了XDES Entry外，还会维护一些文件相关的FSP_HEADER信息，如下图所示：



其中除了Space ID，当前文件大小，以及完成初始化的文件大小等常规信息外，还包括了文件内全局空闲、部分分配或全部分配的Extent链表，以及按照碎片Page分配的Extent链表，属于全局的Extent及Page资源，会在需要的时候按需分配给不同的Segment，并允许其持有，Segment中归还的Extent及Page资源也会加入到FSP Header上的全局链表中，来满足后续需求。全局空间资源不足的时候，也会通过文件的向后扩张以及初始化来获得新的Extent及Page资源。



上图是一个B+Tree文件空间组织的示意图，Page 3作为一个B+Tree的Root Page，在FSEG Header位置记录了其持有的两个Segment的Inode Entry位置，比如叶子节点在Segment ID为2的Segment上，这个Segment的信息记录在Page 2，也就Inode Page上的一个Inode Entry中，包括32个碎片Page的位置，已经用满的Extent链表，还没有用满的Extent链表，以及空闲的Extent链表，这些Extent通过对应的XDES Entry上的指针相互串联起来，图中Page 0是FSP Header，Page 16384是另一个XDES Page，

这两个Page中都包含256个XDES Entry，管理着IBD上的每一个64 Page组成的Extent。

综上所述，为了高效地满足IBD文件内多个B+Tree所需空间的分配及释放需求，InnoDB首先将连续64个16KB的Page组织成一个Extent，从Page 0开始每隔256MB的空间就会放一个特殊XDES Page，每个XDES Page会有256个40B的XDES Entry来维护对应Extent中的Page分配情况。同时，还为逻辑上相关的Page设计了Segment的概念，每个B+Tree的叶子和非叶子节点分别对应一个Segment，IBD可以按照Extent或者Page为单位为Segment分配需要的节点空间，每个Segment会指向一个Inode Page上的192字节的Inode Entry，作为其当前持有的Page或Extent空间的元信息。

总结

本文首先简单介绍了理论上B+Tree的背景；紧接着从数据索引、并发控制及故障恢复三个方向介绍了B+Tree在InnoDB中的位置以及其不可或缺的作用；之后从B+Tree上维护的行数据入手，从Record到Page，再到B+Tree的从小到大顺序介绍了B+Tree中的数据组织；然后介绍了B+Tree上对数据的查询、修改、插入及删除等操作的基本流程，以及随之带来的树结构的变化，节点分裂合并及树整体的升高降低实现。有了这个准备后，又详细的分析了B+Tree上的并发控制实现及发展过程；最后将B+Tree放回到文件中，介绍B+Tree是如何从文件中申请及释放Page空间的，以及其中的一些实现考虑。

参考

[1] Bayer R, McCreight E M. Organization and Maintenance of Large Ordered Indices[J]. Acta Informatica, 1972, 1: 173–189.

[2] 浅析数据库并发控制机制

[3] 庖丁解InnoDB之Undo LOG

[4] B+树数据库加锁历史

[5] Mohan C. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes[M]. IBM Thomas J. Watson Research Division, 1989.

[6] B+树数据库故障恢复概述

[7] 路在脚下, 从BTree 到Polar Index

[8] MySQL Source Code

[9] POLARDB · B+树并发控制机制的前世今生

[10] InnoDB btree latch 优化历程

[11] Innodb 中的 Btree 实现 (一)・引言 & insert 篇

[12] Innodb 中的 Btree 实现 (二)・select 篇

[13] innodb_diagrams

[14] Jeremy Cole Blog

MySQL 11 InnoDB 6 PolarDB 12

MySQL・目录

上一篇・AWS re:Invent2024 Aurora 发布了啥 — DSQL 篇

阅读原文