初期调研报告

姓名：洪宇
学号：2022080901022
项目名称：数据库B+树搜索引擎
难度等级：4

调研内容

历史

B树

要论及B+树到诞生，就必须要先提到它的前身——B树。B树由 Rudolf Bayer 和 Edward M. McCreight[1] 在他们的论文《Organization and maintenance of large ordered indices》中提出，最初是为了解决如何更高效地管理大量随机访问文件（random-access files，即可以直接访问而不需要其他文件先被读取）的索引页问题。容易联想到，这里根据索引来查找文件的需求和用一种高效的数据结构来存储键值对，再根据键来查找值的情景很相似，而随后我们将看到这也正是B+树的用武之地。

因此要理解B树的解决方案，首先便要理解在大量键值对中按键查值的困难。首先用一维结构存储并且线性访问的方案肯定是不允许的，因为时间开销过大；然而二叉树同样难以满足要求：首先，一个内结点不可能存储大量数据（或是中间索引），因为结点内的数据过多的话就退化成一维结构存储的查找效率了；其次，既然结点无法存储过多数据，那么结点数必然很大，如果用二叉树存储的话就会面临两个问题——一是深度过深导致查找效率不高，二是无法自平衡所以难以保证存储效率。

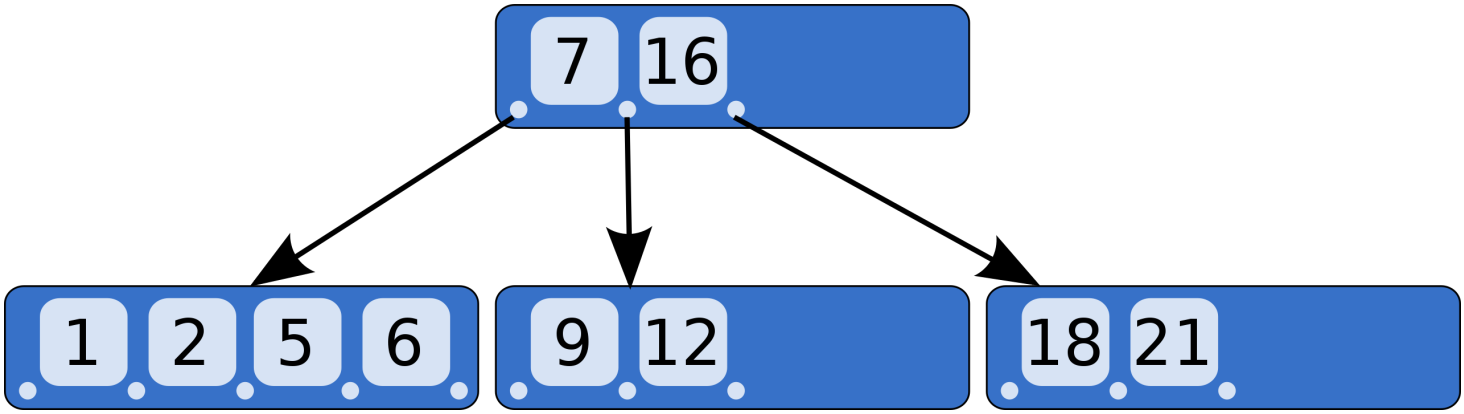
在介绍B树的解决方案之前，需要简要理解一下“深度深”会导致“查找效率不高”的原因。一个处理大数据量的程序和一般程序的最大不同之处在于：由于数据存储在外存（如磁盘），因此它与数据的交互并不局限于内存，而是有着与内存IO相比耗时会多上几个数量级的磁盘IO。然而程序需要将数据从磁盘读入内存后，再根据具体数据来进行接下来的操作。举个例子，可以想象每个结点所对应的数据是磁盘上的一个“块”（block），并且对于内结点而言，指向孩子结点的指针（也即子结点对应的块相对于该结点的偏移量，或者是子结点的地址）也同样存储在这个块中。那么要想通过一个个内结点查找到叶子结点的数据，就必须要重复这样的操作：

1. 把内结点块的数据读到内存中（1次磁盘IO）；
2. 根据读到的指针信息，找到孩子结点在外存中的地址，再把上面的数据读到内存中（1次磁盘IO）。

这也就意味着，深度越深，需要的内结点越多，磁盘IO就越多，存取效率也就越低。因此B树的核心思想便是要增加内结点存数据的能力、降低树的高度，同时又要尽可能地保持平衡。这里引用 Donald Ervin Knuth 对于B树的定义[2]：

> A B-tree of order m is a tree which satisfies the following properties:
>
> 1. Every node has at most m children.
> 2. Every internal node has at least ⌈m/2⌉ children.
> 3. The root node has at least two children unless it is a leaf.
> 4. All leaves appear on the same level.
> 5. A non-leaf node with k children contains k−1 keys.

其中，每个结点中的数据都是有序的，如下图：



每个数据（索引）实际上就是一个块。举个例子，要查询索引5对应的块，先在根结点中比较，随后读入根结点第一个指针所指向的4个块，接着查找（可以使用二分）到5对应的块，便可以对其进行删改查的操作了。同理，如果要查询的是索引7，那么在根结点中找到以后便可以直接返回。

B+树

既然已经有了B树，为什么还需要B+树呢？这表明B树仍然存在一些问题：

1. 由于内结点除了索引还存储了该索引对应的块的数据（或是指向块的指针），因此一个内结点存储索引的能力仍显不足；
2. 最好的情况下在根结点就可以读到数据直接返回，然而最坏的情况却要一直查询到根结点，导致查询效率不够稳定；
3. 由于不同的数据存储在B树的不同层之中，导致如果要进行范围查找，则只能对B树进行中序遍历。然而在之后对SQL的介绍中可以看到，范围查找是一种非常常见的操作。

有关B+树的起源并没有确切的说法，事实上，人们在B树提出之后便不断地讨论把所有数据存储在叶子结点的可能性[3]。 Douglas Comer 在他一篇关于B树的调查中提到B+树被使用于 IBM 的 VSAM（一个对存储文件的访问软件），并引用了 IBM 在1973年的一篇论文[4]。B+树与B树最

大的不同之处有两点：

1. B+树将所有的实际数据（或指向块的指针）都存在叶子结点中，这也就意味着内结点的索引和叶子结点对应了实际数据的索引是可以重复的；
2. 每个叶结点都有指向下一个叶结点的指针（如果有下一个叶结点）。

由于将实际数据都存储在了叶子中，使得存储内结点的一个块可以存放更多的索引，于是经过一次磁盘IO之后所获取的索引数也就更多，这也就意味着可以通过更少的磁盘IO来更快地定位到需要的数据位置。当然，这个与B树相比之下的“更快”也是就平均而言，毕竟B树可以在中途查询到数据后就直接返回，自然在一些情况下可能会比B+树更快。然而，在这个意义上，由于B+树将所有实际数据都存储在叶子，因此所有数据的查询所需要的磁盘IO数都是相同的——这表明它的查询效率比B树要稳定得多。

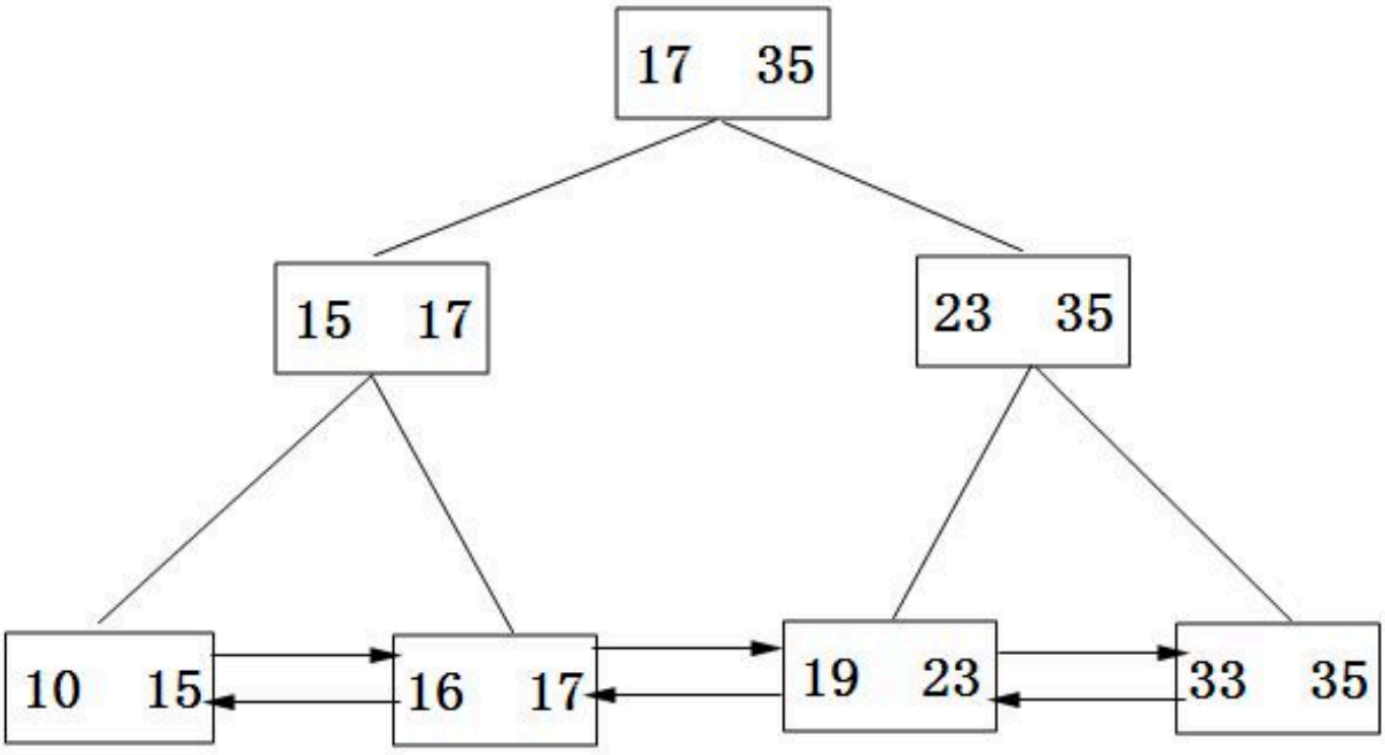
另一方面，通过把数据都放在叶子层并且形成一个链表，B+树范围查找的效率相比B树有了显著的提升（只需要遍历链表的一部分即可），这对于数据库的查询而言是至关重要的。

争议

值得注意的是，B+树的具体定义并没有完全统一的说法。假设一个内结点的分支（孩子）数为m，则有以下两种主流定义：

以最大键分割

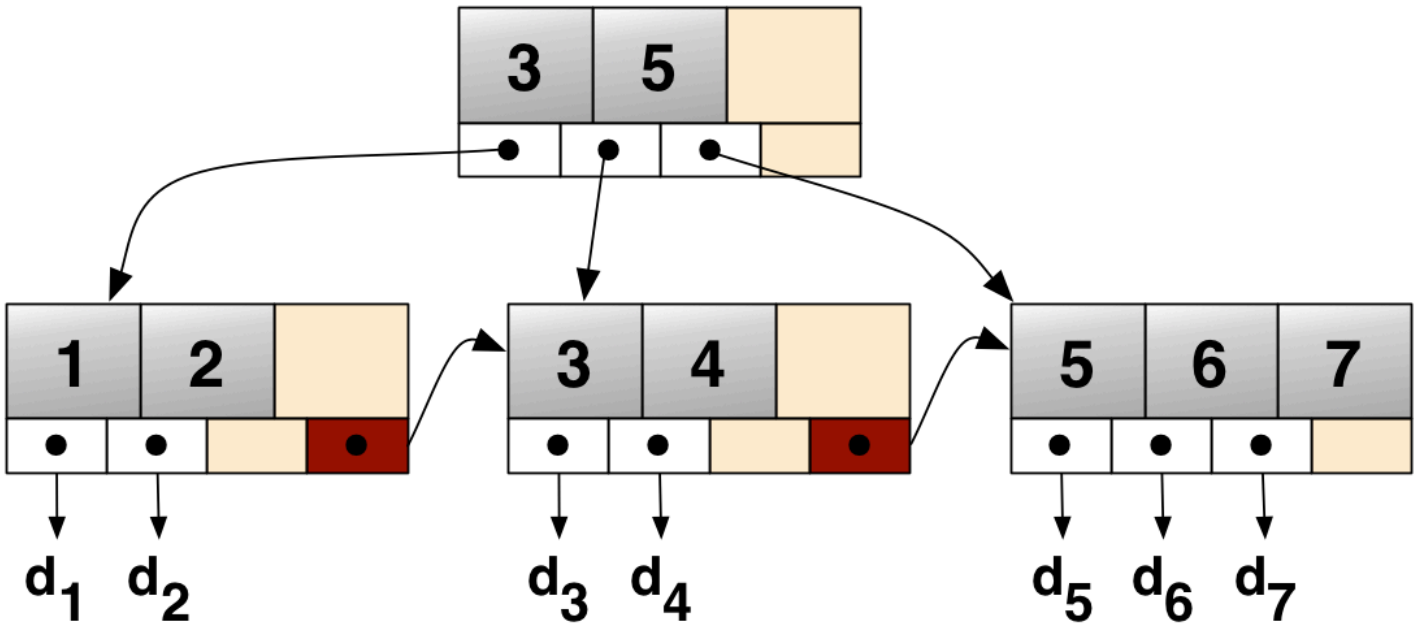
第一种定义也是老师给出的PPT中的定义，如下图：



1. 每个结点所持有的键数也为m；
2. 一个键左边的指针所指向的孩子结点所持有的键都小于等于这个键。

以最小键分割

第二种定义则延续了 Knuth 对B树的定义，如下图：



1. 每个结点所持有的键数为m-1；
2. 一个键右边的指针所指向的孩子结点所持有的键都大于等于这个键。

两种定义的主要区别有二：一是内结点所存的键，于前者而言是孩子结点的最大键，于后者则是孩子结点的最小键；二是分支数与阶数（一个结点所能持有的最大键数），前者分支数与阶数相等，后者则分支数比阶数多一，这是因为后者利用了一个结点最右边的空位并使其再存储一个指向子结点的指针。

我最初的实现采用了定义一，然而定义一最大的不方便之处便是总要判断要插入或删除的键是否是最大键（无论局部还是整体），并且一旦修改了最大键，则整条路径上的索引都要修改。为了达到这条要求，势必要在每次操作前增加对是否是最大键的判断，这让代码的美观度和整体性都被削减了，所以最后采用了定义二，并且加上了一些自己的修改。不过具体实现不在初期调研的范围内，在此不继续提及。

常用领域

对于大量数据的增删改查以及按键查值的操作在数据库和文件系统中尤为频繁，另外，数据的持久化以及对持久化数据的查询也是非常常见的需求，因此B+树被广泛地应用于数据库中，最著名的例子便是MySQL的InnoDB存储引擎。

从数据处理类型的角度来看，B+树则广泛应用于OLTP（Online Transaction Processing）中。OLTP在传统意义上是指并发执行的众多在线交易事务，如网上银行交易、购物、订单条目的处理、发送信息等等[5]。不过随着互联网的普及，OLTP的含义逐步扩大到了各种以网络为媒介的行为，如下载网页文件、观看在线视频、发送社交评论等。

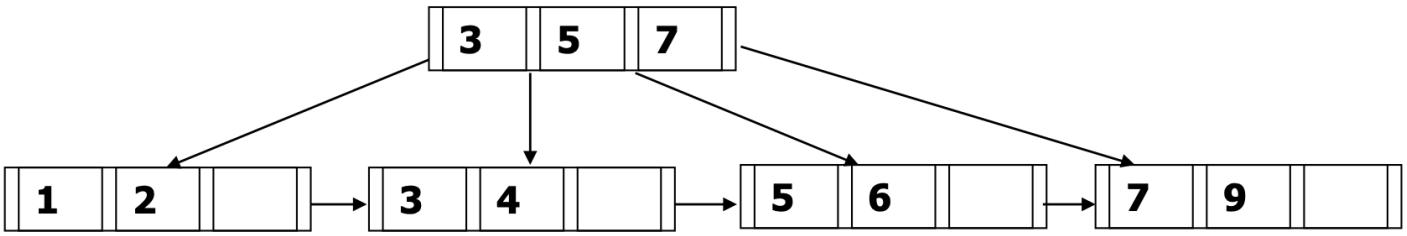
相关技术

CRUD

在维持B+树的自平衡上，无论如何优化（比如针对并发的效率而延迟父结点更新），其基本操作都是不变的。其核心思想便是控制每个结点的键数在一定范围内，如果超出范围则将键在相邻结点间重新分配。注意，这里采取的B+树定义是按最小键分割。

查询

一个典型的小型B+树如下图：



由于每个结点内的键是有序的，因此操作与二分查找以及二叉树的查找类似，具体为（设查找的键为k，每个结点的键值数组为key）：

1. 从根结点出发，二分查找到第一个下标i，满足key[i] ≤ k；
2. 查找key[i]所对应的子结点；
3. 在子结点的key数组中二分查找；
4. 重复2-3过程直到叶子；
5. 在叶子中查找k，若无键为k则表明不存在，有则返回k对应的值。

插入

由于对于B+树阶和分支的关系没有统一的定义，因此这里采取我的实现中的定义：

设最大分支数为M，则一个结点所能拥有的键数最大为M-1，每个结点所含键数范围为[M / 2, M - 1]，这里的除法为向下取整且为闭区间。则插入的具体过程为：

1. 与查询方法类似，首先找到应该插入的叶子；
2. 将要插入的键值对插入叶子中；
3. 若叶结点所含键数 ≥ M，则将其“分裂”，具体为：
    i. 创建一个新的叶结点，将较大的一半的键值对转移到新结点；
    ii. 将新结点的最小键插入到父结点的正确位置并更新父结点的指针。
4. 在回溯到根结点的过程中重复第3步。

通过不断调整各结点的键数在范围之内，B+树的平衡性得以保证。

删除

首先需要找到待删除的键值对所在的叶结点，若该键不存在则返回，否则：

1. 删除该键值对；
2. 若该结点为根结点，则返回（此时该树只有一个根结点）；
3. 若叶结点的键数比 M/2 要小，则找到其相邻的一个兄弟结点sib（由于该结点不为根，则必然有兄弟结点）；
4. 若sib的键数 > M/2，则将sib的一个键值对转移到该结点；
5. 若sib的键数恰好等于 M/2，则将该结点与sib合并，并更新父结点中的键和指针（注意，此时该结点有 M/2 - 1个键，sib有 M/2个，则合并后键数最大只能为 M-1，因此不会超出范围）；
6. 在回溯的过程中重复第3-4步。

范围查找

范围查找便是体现B+树比B树的优越之处的时候了：

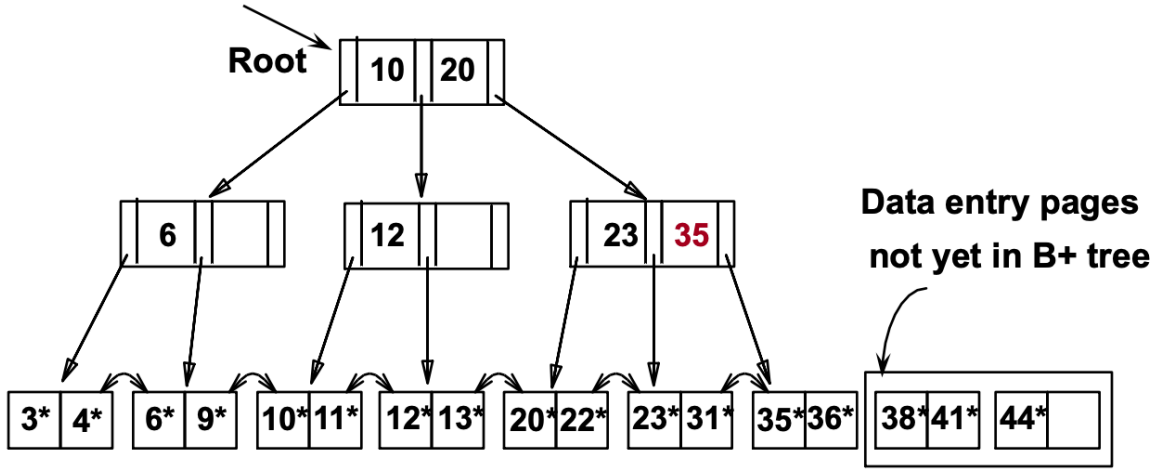
1. 与查找相同，找到左边界的键所在的位置；
2. 利用叶结点所构成的链表，直接向后遍历各个叶子；
3. 当叶结点的键 ≥ 右边界的键时结束。

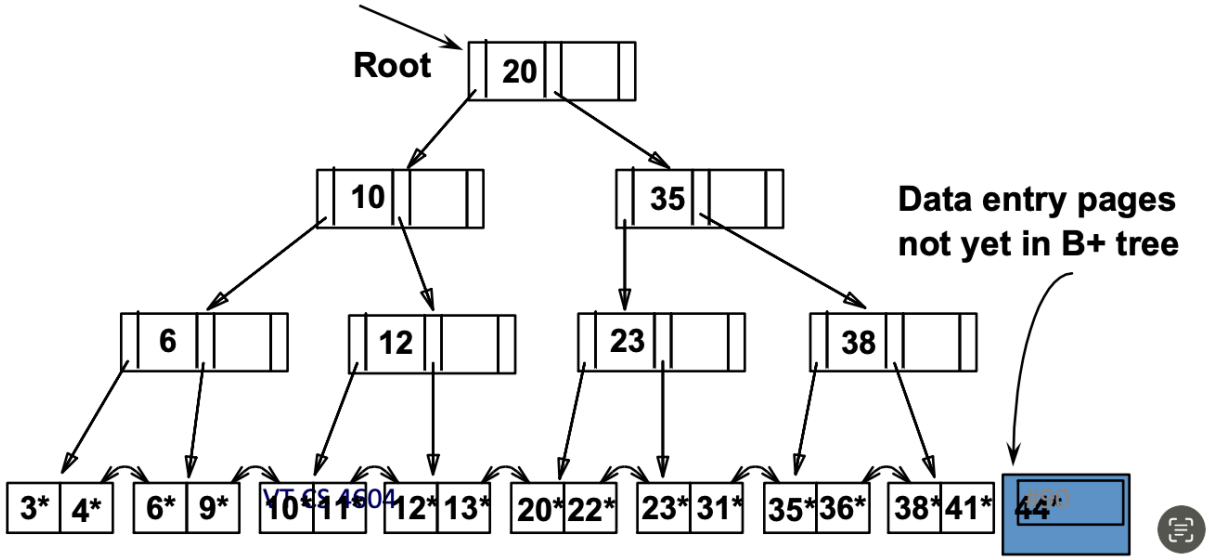
Bulk Loading

假设有大量的键值对，然而B+树还未构建，那么如何才能快速地由这些键值对构造出来一棵B+树呢？根据前面介绍的操作，容易想到可以一个键一个键地插入到树当中。然而由于B+树的结构特点是已知的（如阶数），而且现在又知道了所有叶子结点，那么我们容易反推出来插入完的B+树应该是什么形态。另外，由于在插入的过程中会不断分裂来保持平衡，因此一个个插入的方法会多上许多不必要的开销。

插入完成的形态是很容易反推的：只需要自底向上地建树，每当键数达到最大值就在其上构造一个父结点，不断递归便可以获得平衡的B+树，如下图：

Bulk Loading of a B+ Tree





优化

注意到上面的Bulk Loading实际上将每个结点都填充满了。如果之后的操作以查询为主，这样生成的树高度更低，自然查询效率更高；然而更可能的情况是之后还有许多插入、删除的操作，那么每更新一个结点就很可能会导致多个结点的分裂、合并，可能会造成更大的开销。

因此可以预先设置一个填充度。举个例子，当设置填充度为75%时，假设阶为4，则每插入3个键就在其上生成一个父结点，如此一来随后的插入操作便可以引起更少的大范围更新了。

Latch Crabbing

前面提到，B+树广泛应用于OLTP中，因此对并发的高效支持是必然要求。在并发中，对共享变量最一般的保护措施便是加锁。一个锁可以与一个对象相关联，并且在同一时间内只能被一个线程持有，这样便能保证这个线程操作的“原子性”。所谓原子性，即该操作“不可再分”，这意味着它要不没有开始，要不一定完成，不会中途被干扰打断。举个例子，假如在程序设计中每

个结点便是一个对象（一个实例），并且每个对象关联一个锁，那么当我要读写一个结点时，便先尝试获取它的锁，如果此时该锁被其他线程持有便只好等待，拿到锁之后进行读写操作，操作完毕之后再将锁释放，使得其他排队等待的线程可以读写这个结点。

然而上面的操作存在一个问题：如果在线程A读写结点a的过程中，线程B更新了a的父结点b，那么由于A、B都无法访问对方持有的结点，最后的结点关系不就可能不正确了吗？举个例子，由于插入操作，a分裂并且需要将一个键推到父结点中作为指向新结点的索引；同时b也分裂了，并且按照正确的键值排列a是b分裂出的新结点的孩子。然而由于B没能修改a结点的父指针，因而a便会将新的索引推给错误的父结点。有许多类似的错误，在此不一一举出，不过根源都是相同的——那便是如果只给当前在读写的结点加锁，那么这个操作实际上不是原子的，因为其他的共享变量（即其他的结点）与这个结点是有关联的。

最简单粗暴的解决方案便是每当进行读写操作时就给整棵树加锁。然而在实际的OLTP中，这样的操作是绝对不可容许的，因为它会造成巨量的线程阻塞，进而造成巨大的经济损失。那么有没有一种加锁协议，可以使得加锁的粒度（可以理解为操作时锁住的结点数量）尽可能地小呢?

这便是 Latch Crabbing 的由来。它的具体内容如下：

首先给出“安全结点”的概念。我们称一个结点是“安全”的，当且仅当：

- 若在删除中，则删除一个键后不会合并，即键数 >= M/2 + 2
- 若在插入中，则插入后不会分裂，即键数 <= M - 1

接着介绍读写锁。容易想到，当我们只需要读一个结点时，可以允许其他线程同时读取这个结点，因为它们的操作是互不影响的。因此，当一个结点被加上读锁时：

- 其他线程也可以给它加上读锁
- 所有线程都不可以给它加上写锁

而当一个结点被加上写锁时，其他线程不能再加锁。由于实际情况中，读比修改操作占比要大得多，因此读写锁分离的设计让并发效率有了显著提升。

接下来先给出 Latch Crabbing 的操作步骤，再谈一点自己的理解。

操作步骤

查询

重复执行：

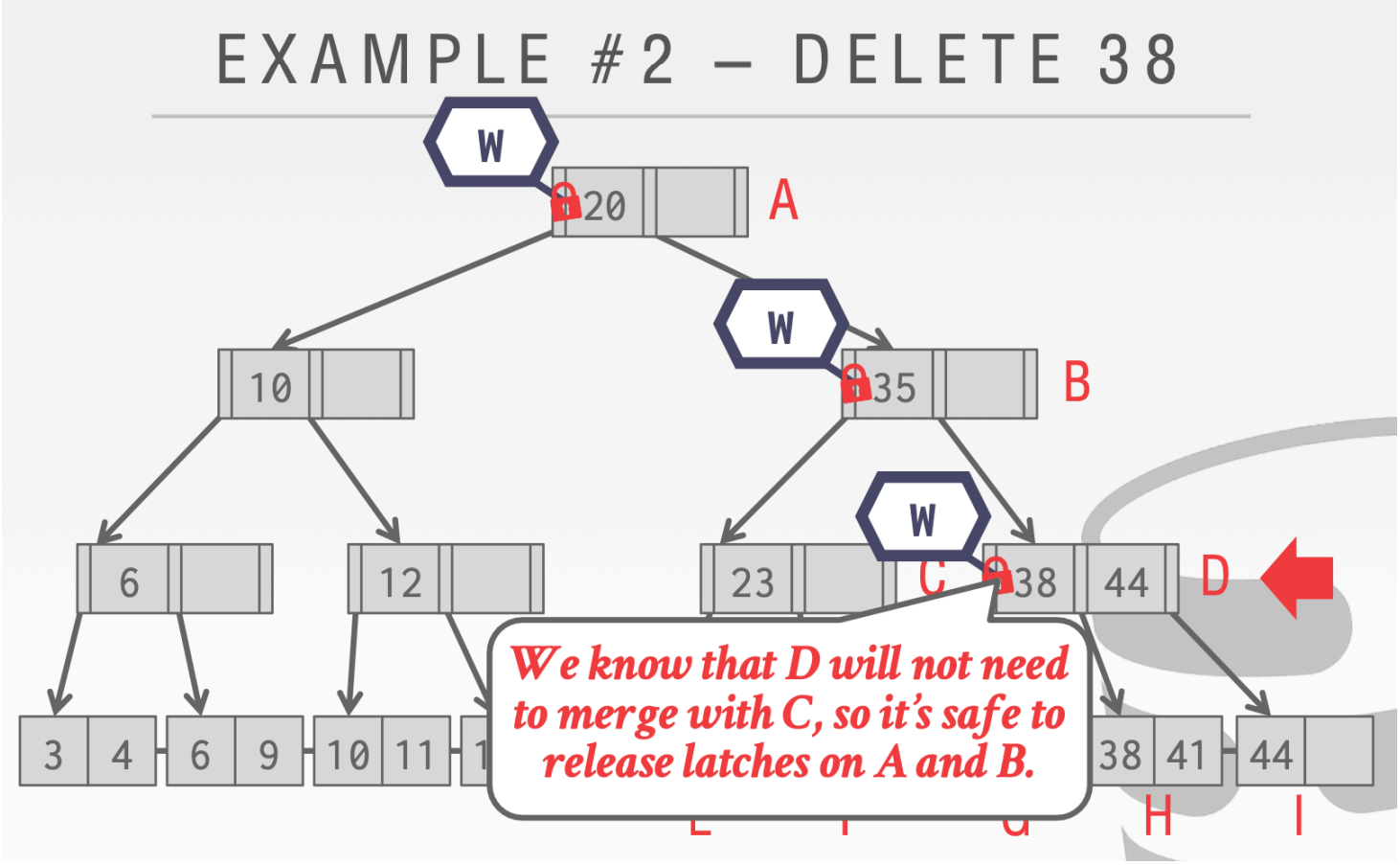
1. 获得该结点的读锁
2. 获得孩子的读锁
3. 释放该结点的读锁

插入/删除

重复执行：

1. 获取结点的写锁
2. 获取孩子的写锁
3. 若孩子是安全的，释放孩子的所有祖先的写锁

一个简单的示意图如下：



范围查询

在遍历叶子时，重复执行：

1. 获取叶子读锁
2. 获取下一个叶子的读锁
3. 释放该叶子的读锁

个人理解

那么，Latch Crabbing 协议中为何能放心地释放祖先的锁呢？

首先，为什么离开了祖先结点还需要保留加在上面的锁呢？加锁的目的有二，一是保证自己对该结点的操作不会受其他线程干扰，二是自己接下来的操作不会使其他线程对祖先结点的操作受到影响。

因此，如果已知自己接下来的操作都不会影响到祖先结点，那么意味着：1.由于自己不会再操作祖先，所以本线程接下来的操作就祖先结点而言，不再受其他线程干扰，从而满足条件一；2.本线程不再操作祖先结点，则不影响其他线程对祖先的操作，从而满足条件二。

简而言之，在插入/删除中一旦确定孩子是安全的，那么相当于祖先结点与自己接下来的操作都无关了，于是就可以放心地释放其所有祖先的写锁了。

Latching 优化

既然说到优化，则上面的协议仍然有性能瓶颈。简单观察一下可以发现，有个结点对于所有的操作而言都非常重要，然而它又总是被第一个加上写锁，并且很可能迟迟不被放锁。我们知道，一旦一个结点被加上了写锁，那么其他线程连读取操作都无法进行了。这个结点是谁呢？它便是位于所有操作最开头的根。

另外一个观察结果便是：由于实际的数据库所使用的B+树往往有着很大的阶数，因此大多数的插入/删除并不会导致结点的分裂或合并，这也就意味着大多数情况下叶子都是“安全的”。基于这样的假设，可以先尝试不加写锁地到达叶结点，假设叶结点是安全的，那么直接插入/删除即可。如此一来便避免了总是给根加上写锁，导致众多线程阻塞的尴尬局面。

优化的具体操作如下（当然，这只针对插入和删除）：

假设叶子是安全的，则从根结点开始：

1. 获取本结点读锁
2. 获取孩子读锁
3. 释放本结点读锁
4. 重复1-3直到到达叶子
5. 叶子加上写锁
6. 若叶子安全，执行操作并释放写锁

延迟父结点更新

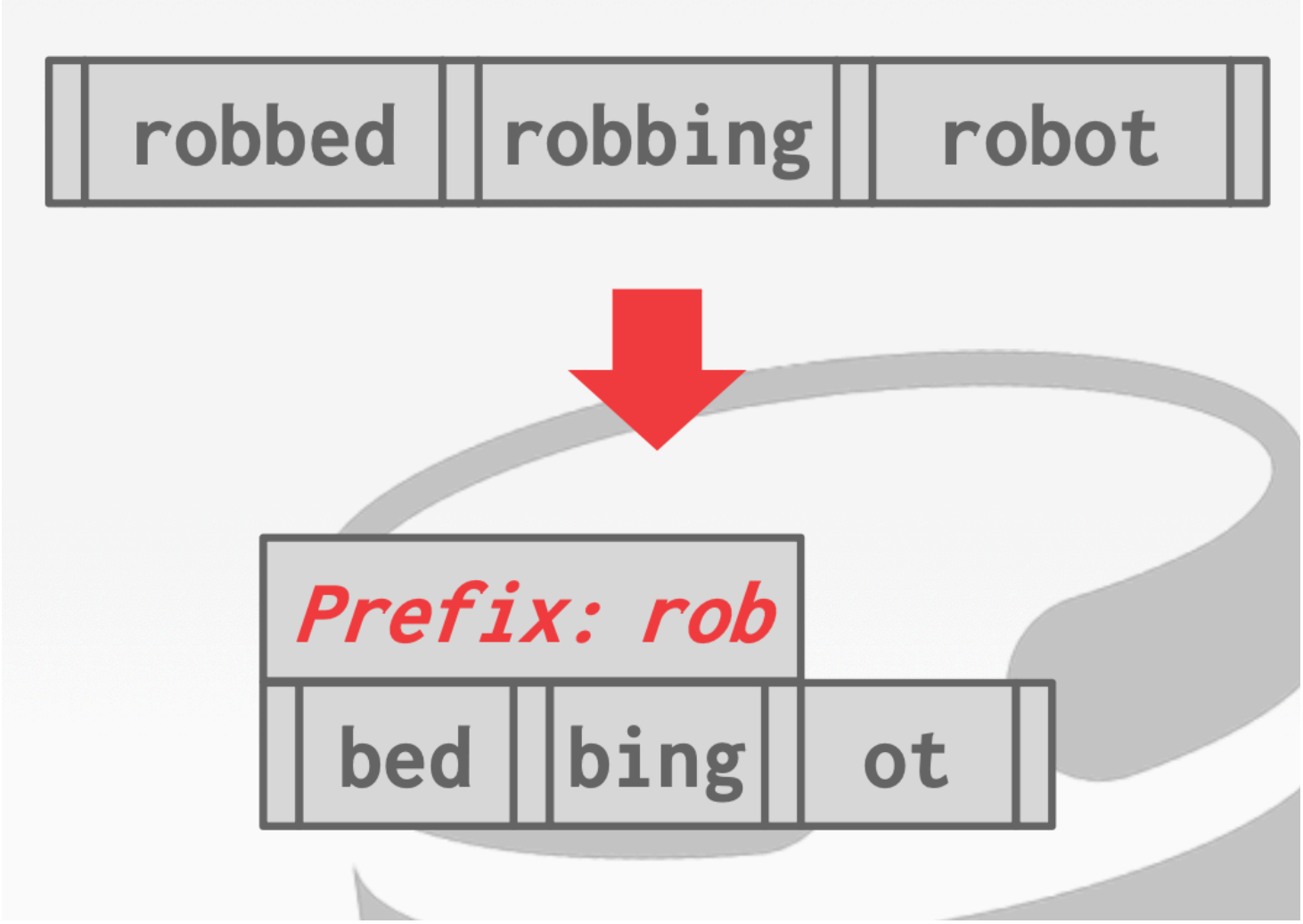
从前面的 Latching 优化可以看出，非必要时不加写锁往往可以获得可观的并发性能提升。延迟更新的核心思想，便是当一个叶子分裂时，我们不更新父结点，而是留在下一次需要给父结点加上写锁时再更新它的键，这样一来便可以减少一次给父结点加写锁的操作。

那么为什么可以延迟更新呢？我的理解是，由于叶子分裂后键仍然是有序的，并且仍然可以通过父结点的指针来找到新的叶子（只不过会稍微增加一点在叶子和新叶中二分查找的时间），因此并不会影响读取操作的正确性。

Prefix Compression

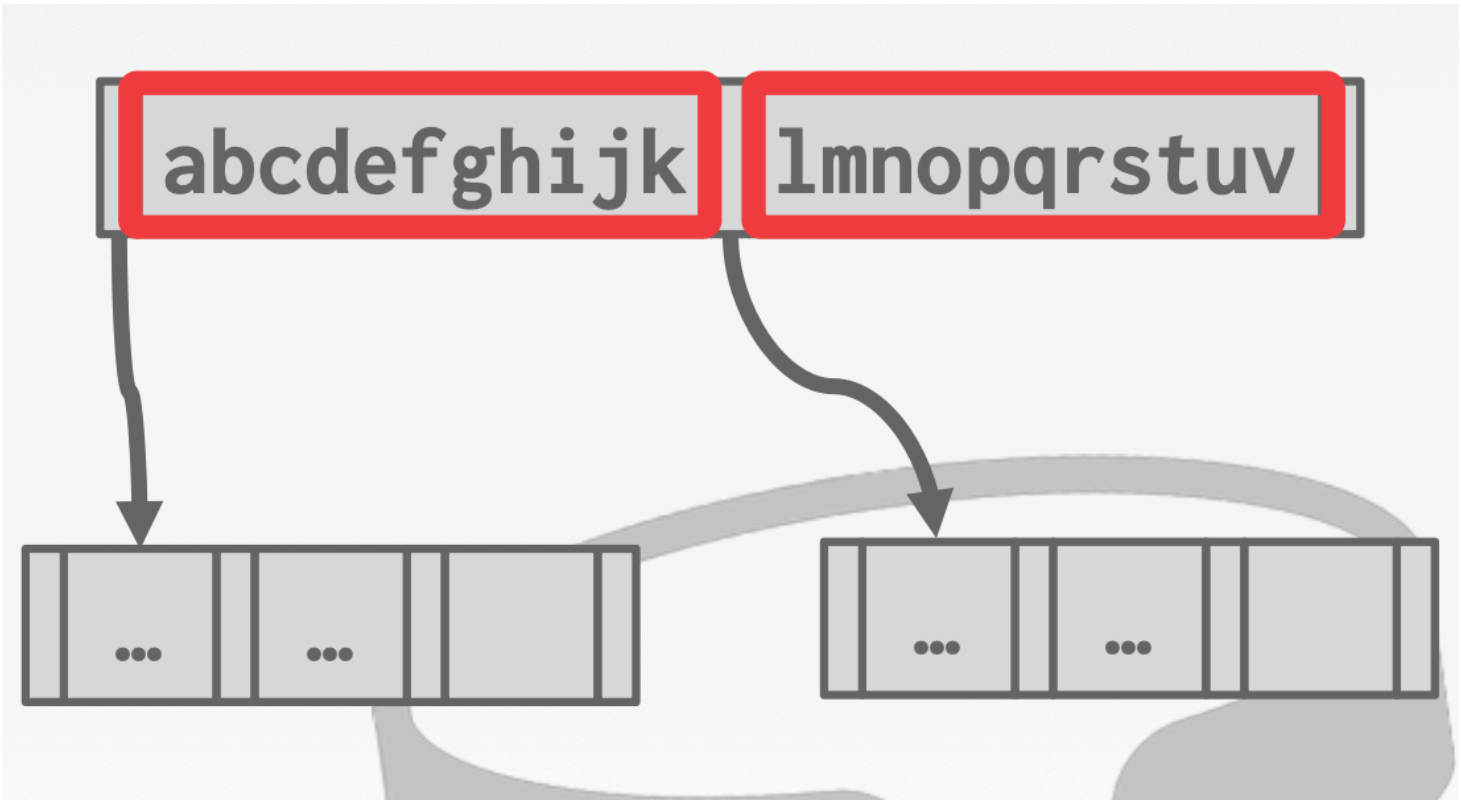
直译过来便是“压缩前缀”。它的思想是这样的：由于键都是有序的（注意键并不只有数字类型，现实情况中往往是字符串，比如人名），因此在数据量十分庞大的时候，很有可能相邻的多个键都有共同的前缀。因此，可以不把每个键都完整地存储下来，而是仅存储一次它们的共同前缀，从而节省空间。

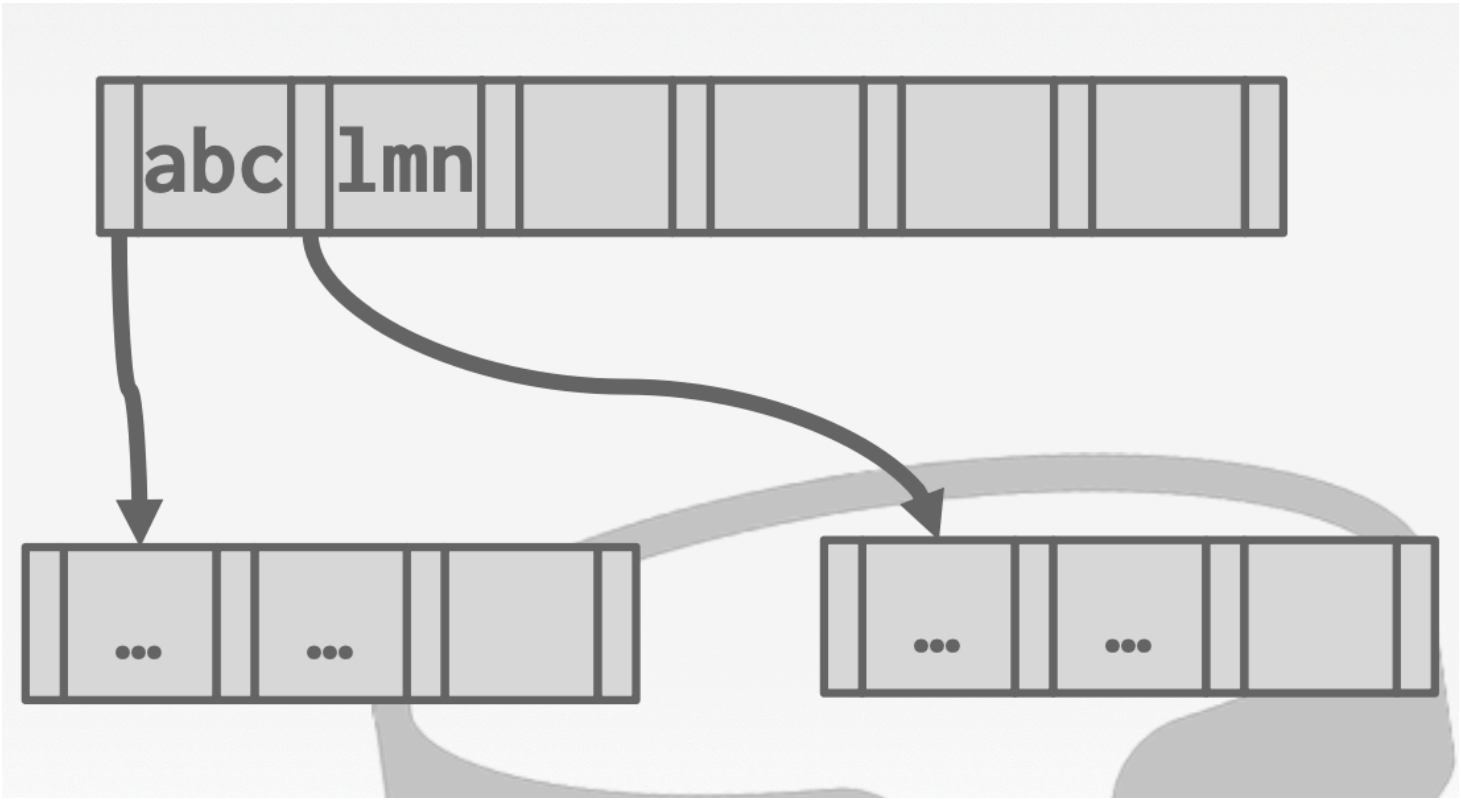
如下图所示：



Suffix Truncation

与压缩前缀的节省空间的目的相同，不过操作的部分相反，这种优化技术的思想是：由于内结点的键所具有的唯一作用便是将所有孩子结点分割成有序的几块，并“告知”每一个操作应该沿着哪条路径走到下一个结点，因此并不一定整个键都是具有区分效力的。如下图所示：





这两张图非常简洁明了地表明了：实际上我们只需要存储“abc”“lmn”这两个前缀部分，就足以完成这两个键的“引路”职责了。

Storage Manager

从这项开始，后面的技术就和数据库本身而非B+树更相关了。由于本项目的重点在于B+树搜索引擎的实现，因而基础要求里并没有指定数据库的存储方式，所以我大体上有两种选择：

1. 仅存储在一个Java程序所申请到的内存中。这表明每次程序启动时会重新建树，并且程序

终止时对象销毁、内存释放，因而存储的数据的生命周期等同于这个Java程序的生命周期；

2. 持久化到磁盘上，从而数据可以实现“永久”存储，并且数据库的大小不再受程序申请的内存大小的限制。

此处介绍的 Storage Manager 便是第二种情况下可以用到的技术，当然它也是任何一个“真正的”数据库所必需拥有的组件。

所谓 Storage Manager 即存储管理，它实际上是许多技术的总和，在我的理解中它主要表明了数据库的存储逻辑。数据其实仍然以文件的形式存储在数据库中，既可以只用一个文件存储（如Sqlite），也可以分文件存储（如MySQL）不同类型的数据，比如实际数据和元数据（如表列的类型等）。那么在这些文件中，数据又以什么形式存储呢?

记录

首先，关系型数据库基本都是按行存储的。可以简单地理解一个关系型数据库为多个表的集合，而每张表可以看成一张二维的表格，每一行对应一个条目，而每个列则是这个条目的各个属性。如果以OOP的结构来比喻，那么每行就是一个对象实例，每个列定义了它所拥有的字段。比如下图：

这里的记录头信息（header）包含了非常多的额外信息[6]，比如下一条记录的指针（从而各条记录可以串成一个单链表）等等。下图便是一个例子：

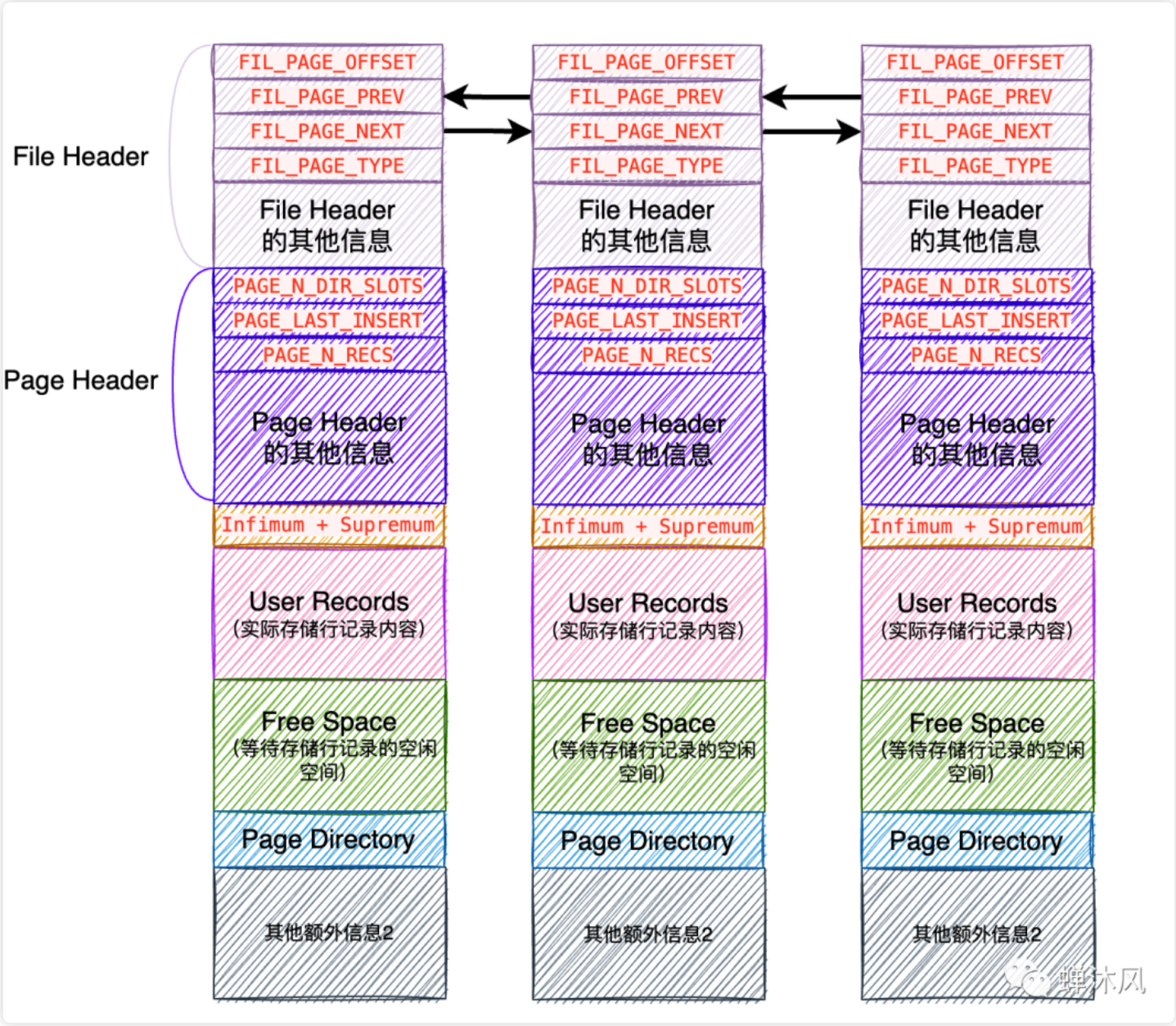
名称	占用bit数	描述
预留位1	1	暂时未使用到
预留位2	1	暂时未使用到
delete_mask	1	标记该记录是否被删除
min_rec_mask	1	B+树的每层非叶子节点中的最小记录都会添加该标记
n_owned	4	当前槽拥有的记录数
heap_no	13	当前记录在记录堆的位置
record_type	3	表示当前记录的类型，0表示普通记录，1表示B+树非叶子节点记录，2表示最小记录，3表示最大记录
next_record	16	下一条记录的相对位置

页

前面提到，为了获取一个数据的信息，需要将其从磁盘加载到内存中。然而如果每次只加载一行的话无疑效率极低，因此每次加载数据量的大小有一个固定的值，并且这些条目所构成的集合被称为“页”（page），因而各种数据在磁盘上便是按页来组织的。不同的存储引擎有不同大小的页，比如SQL Server是8 KB，MySQL是16 KB。对于InnoDB而言，这意味着默认情况下，一次最少从磁盘中读取16KB的数据到内存中，一次最少把内存中16KB的内容刷新到磁盘上。

简单而言，页其实便是固定大小的数据块，页中存储了数据库中所有相关的数据，例如元组（tuple，即一条记录）数据、元数据、索引还有日志记录等等。一般而言，我们会尽量将内容存储在单个页中，从而一个页便是自包含的，即关于如何解释和理解页内容，所需要的所有信息都已经存储在页本身中。这样，即使丢失了任何一页，也不会影响其他任何一页的解析和使用。反之，如果把元数据与元组数据分开存储在不同的页，那么一旦元数据页丢失或者损坏了，元组数据页也就无法解析了。

下图表示了3个典型的数据页以及它们之间的连接[7]：



可以看到，除了实际存储的用户记录，以及为了以后的记录插入而预留出的空间之外，还有着大量的额外信息。下面简要介绍其中一些关键组分的含义：

- FIL_PAGE_OFFSET
  数据页地址的偏移量。由于唯一标识了这个页，因此可以作为找到它的依据。

- FIL_PAGE_PREV，FIL_PAGE_NEXT
  指向前后两页的指针，因而数据页间构成了一个双链表。

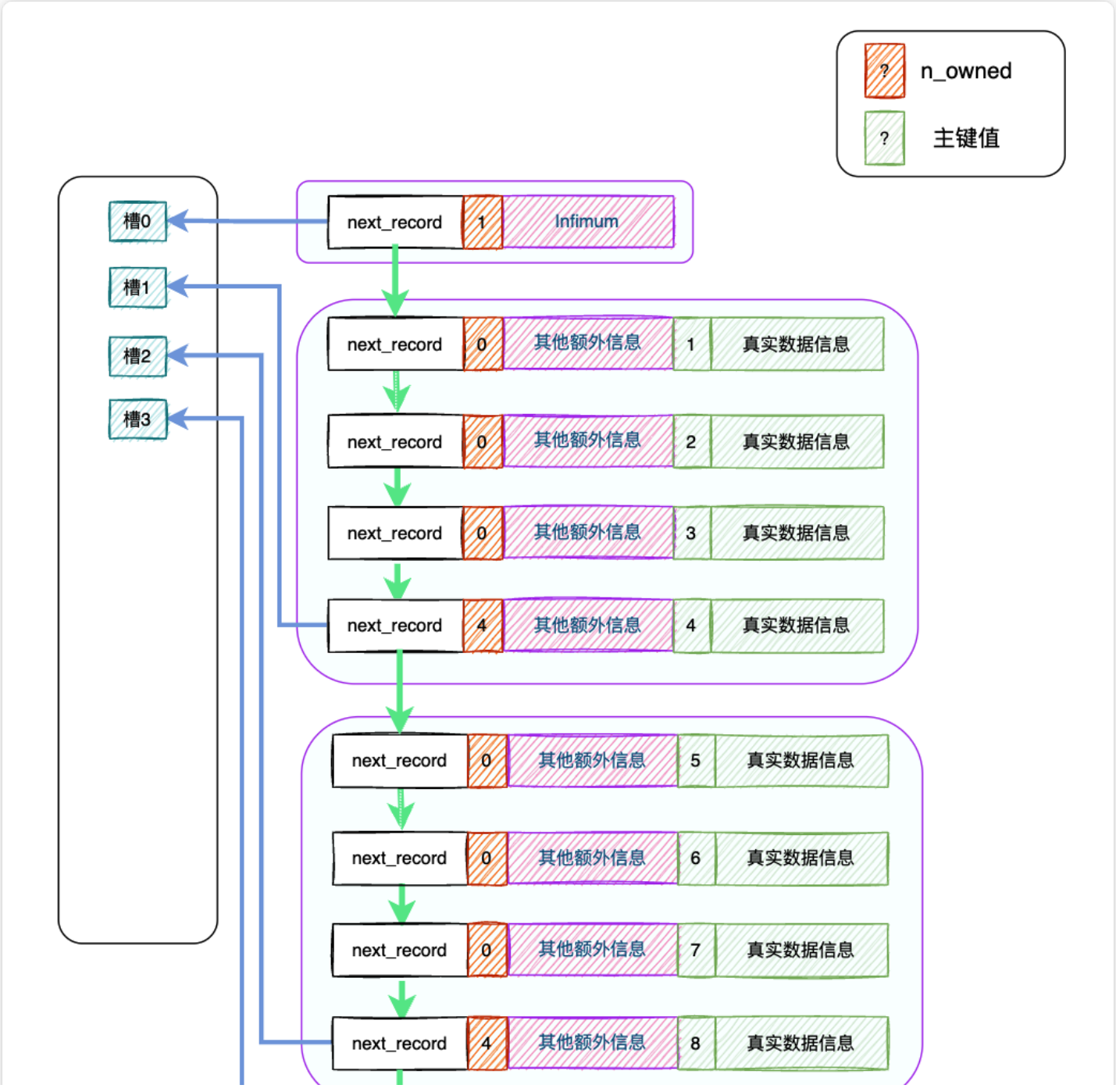
- FIL_PAGE_TYPE
  前面提到，一个数据页应该是自包含的，那么为什么还会有类别区分呢？这是因为除了存储实际数据之外，还有如存放“undo日志”信息的页，这是为了应对诸如还未将数据从缓冲区冲回磁盘便断电的情况而产生的。有关日志技术会在下文中提及。

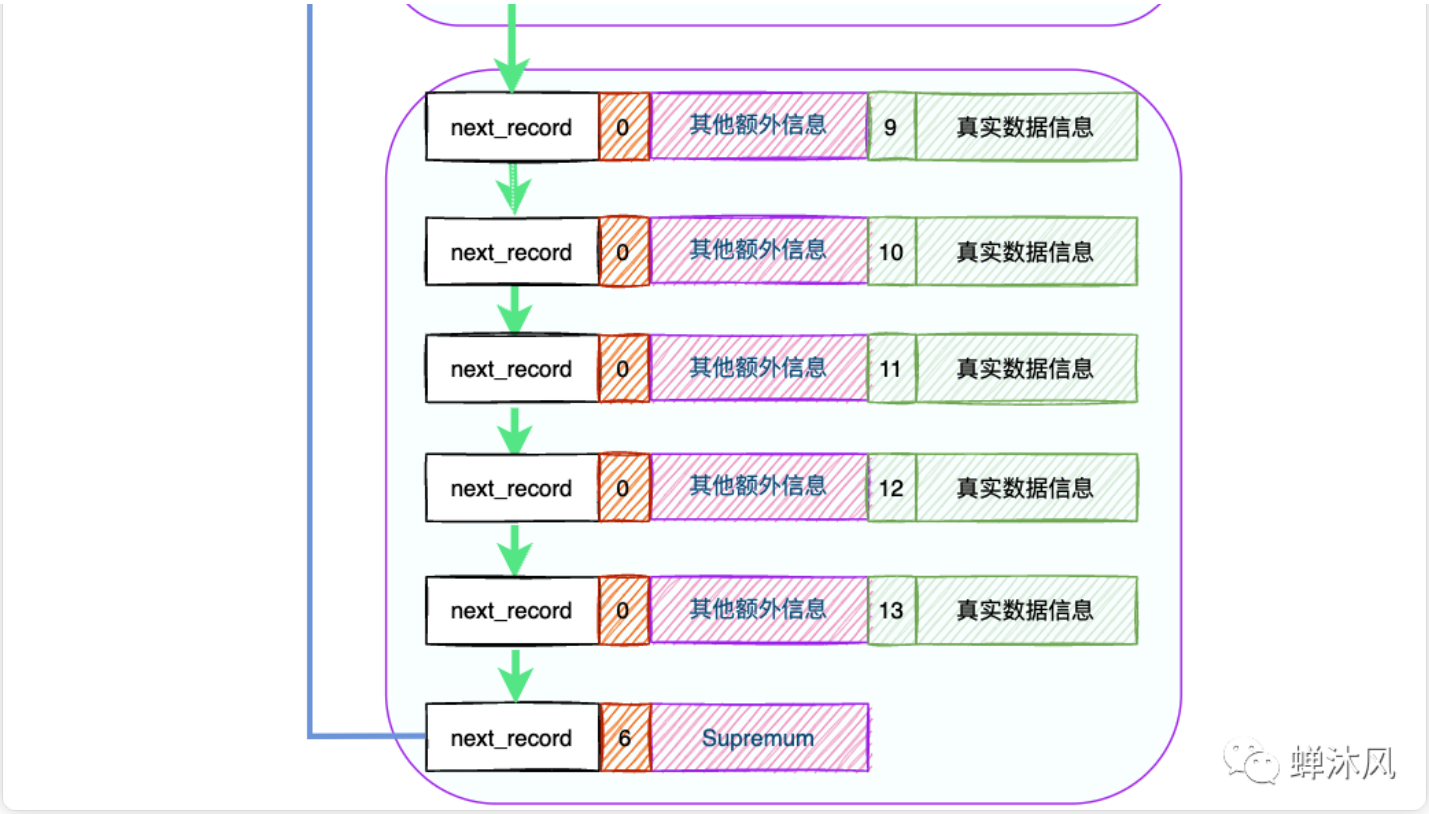
- PAGE_N_DIR_SLOTS
  所谓slot，直译过来即是“槽”，可以想到它应该是一个组织数据页的结构。在我的理解中，

一个页实际上便是一个B+树的叶结点。当我们在B+树中查询一条记录时，我们会先找到包含该键的叶子，接着在叶子中二分查找到对应的记录（也就是前面介绍B+树时提到的键值对）。然而，前面对B+树的元素没有具体的定义，只是很抽象地简单认为一个叶子中的各条记录已经构成了一个数组（或是一个链表），因而只需定义一个记录之间比较大小的方式，我们就可以容易地二分查找了。

如果要和前面的抽象完全吻合的话，我们需要将数据页读入之后再将其生成为一个Java对象，需要读入内存的数据页都被处理为对象后再由它们生成一个暂存在Java程序的内存中的B+树。可是，这样一来，我们就无法忽视将数据页构造成一个Java对象，以及将这些对象构造成一个树的开销，显然在真正的数据库中是不会如此操作的，否则每次打开数据库都要进行长时间的初始化。那么，首当其冲的问题便是：如何在叶子（一个页）中高效查找？难道只能顺序遍历吗？

槽便是为了解决这个问题而诞生的。一个典型的由槽组织的数据页如下图：





可以看到，图中对各条记录进行了分组，并且每组都对应一个槽。

需要注意，记录之间是有序的。因此槽所对应的存储结构很类似于B+树里一个内结点和它的子结点的关系（也有点类似于查找表）。槽记录了组中最大记录的键值，以及它对应的地址，于是遍历一个数据页时只需先遍历槽即可。另外，由于槽与槽间在物理空间上是连续的，因此遍历槽非常快。

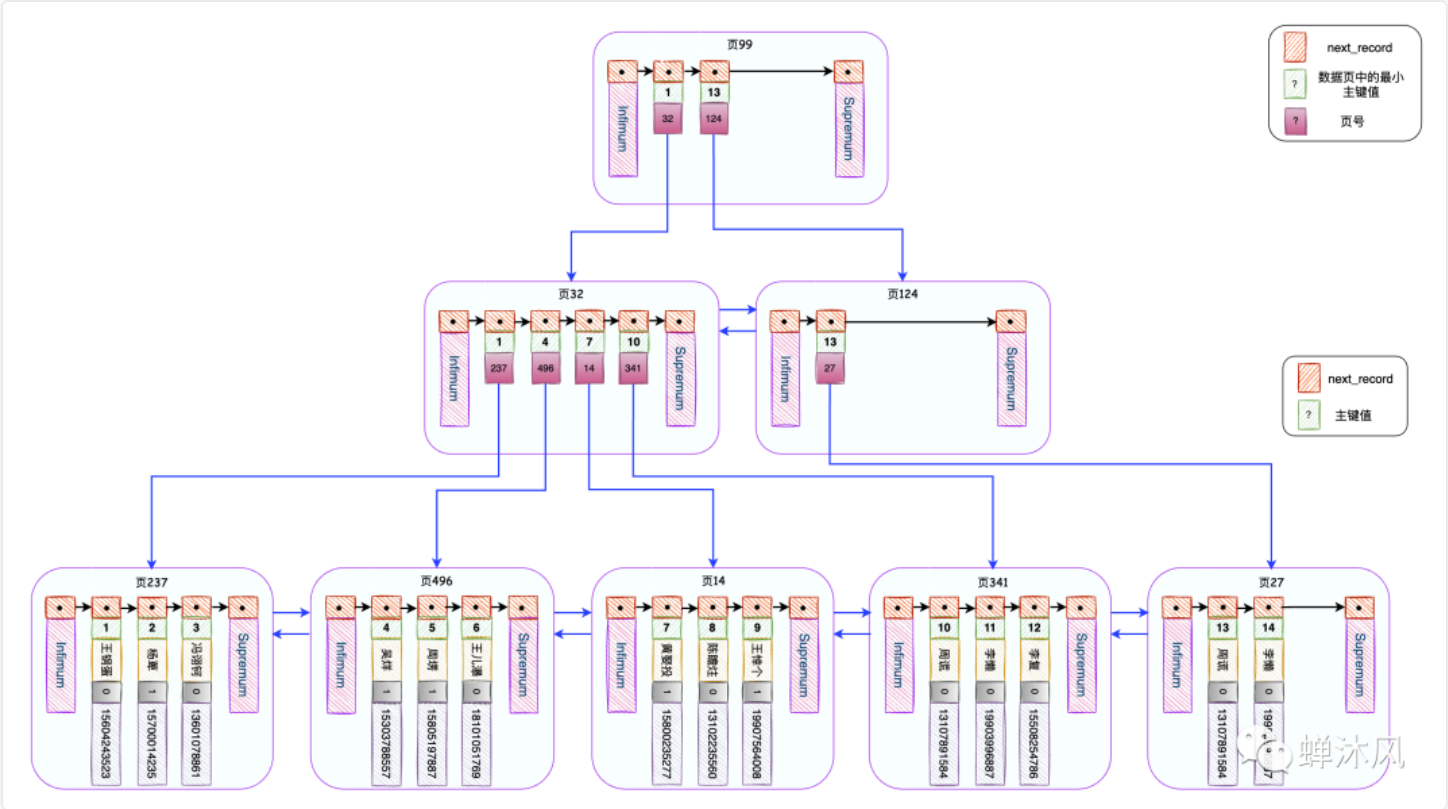
索引

主键索引

我们已经从一条记录上升到了一个数据页。那么当数据页越来越多时，又该如何快速定位到正确的数据页呢?

从前面数据页连接的图中可以看到，页与页间构成了一个双链表，因此最简单粗暴的方式便是遍历链表。显然线性时间在真正的数据库中是不能容许的，那么如何才能更好地组织起数据页呢？或者说，在已知一个页可以对应到B+树的一个叶结点的情况下，如何组织起一棵B+树?

很简单，既然已经提示一个页可以是一个叶结点了，那么再使用一些页来作为内结点不就可以了么！因此，一个典型的组织方式如下图：



内结点页相当于一个目录，它的每条记录便是叶子数据页的最小主键值和对应的数据页号（FIL_PAGE_OFFSET）。至此便能很容易地对应到一棵B+树的结构了。

普通索引

上文提到的索引是以没有重复键（都为主键）为前提的。然而，在SQL的操作中，我们检索的依据往往是“列”（即不同记录的某个属性）而不是“行”（即一个主键所对应的一条记录）。最容易想到的解决方案是：我们为每个副键（也就是列）单独建一棵B+树，并且允许重复键就可以了。然而，它的弊端也很明显——这样一来，就需要重复多次存储相同的数据了。

解决方案也很简单：既然查询记录时总需要以主键为媒介，那么只需要以副键为键、主键为值构成键值对，再建一棵B+树就可以了，如此便能减少空间的开支。另外，由于值只有主键，因此内结点也可以同时存储值，这样当我们在内结点就遇到需要查找的键时，就可以直接把值取走然后返回了。拿到主键之后，我们再查询主键对应的B+树即可（这有个专门的名称，叫做“回表”）。

联合索引

在SQL建立索引或建表的语句中，并不一定指指定单个列为索引，也有可能指定多个列构成联合索引（如 `CONSTRAINT key PRIMARY KEY (row1, row2)` 便会根据 row1 和 row2 两个列生成名称为 key 的主键索引）。解决方案与普通索引完全类似，只不过多了一个比较——当两个联合索引 k1 和 k2 的 row1 部分相同时，接着比较 row2，以此类推。

Buffer Pool Manager

前面提到的 Storage Manager 是对数据库在磁盘上存取逻辑的具体实现。如果数据量小，我们一次性把数据读到缓冲区（Buffer Pool）就可以了。然而，实际的情况往往是：磁盘中存储的文件比内存要大得多，那么一次性就只能读取一定量的页。这就涉及到多个问题，如：

- 每次读取多少页？应该总是把缓冲区填满吗？还是应该需要多少就读多少？
- 如果要查找的页不在读入的页中怎么办？
- 何时把页冲回磁盘？
- 如何在多个线程需要读写一个页的情况下保证并发安全？

Pre-Fetching

第一个问题其实很好判断。前面提到，数据库操作的性能瓶颈在于磁盘IO。因此，预读取（Pre-Fetching）的思想便是当我们需要将磁盘中的页加载到内存中时，不是仅仅将需要的页加载进来，而是通过一些预读取策略，多加载一些页到内存之中，从而减少以后的磁盘IO。

有关“预读取策略”，最容易使用的依据便是当查询语句显式地指出还会读取其他页时，比如对叶子结点的顺序扫描，或者对索引的扫描。我的另一个猜想是依据局部性原则，即即使查询没有显式指出还会读取其他页时，也读取一定量的邻近页。当然，还有一些更高端的算法[8]，在此先不做太深入的研究。

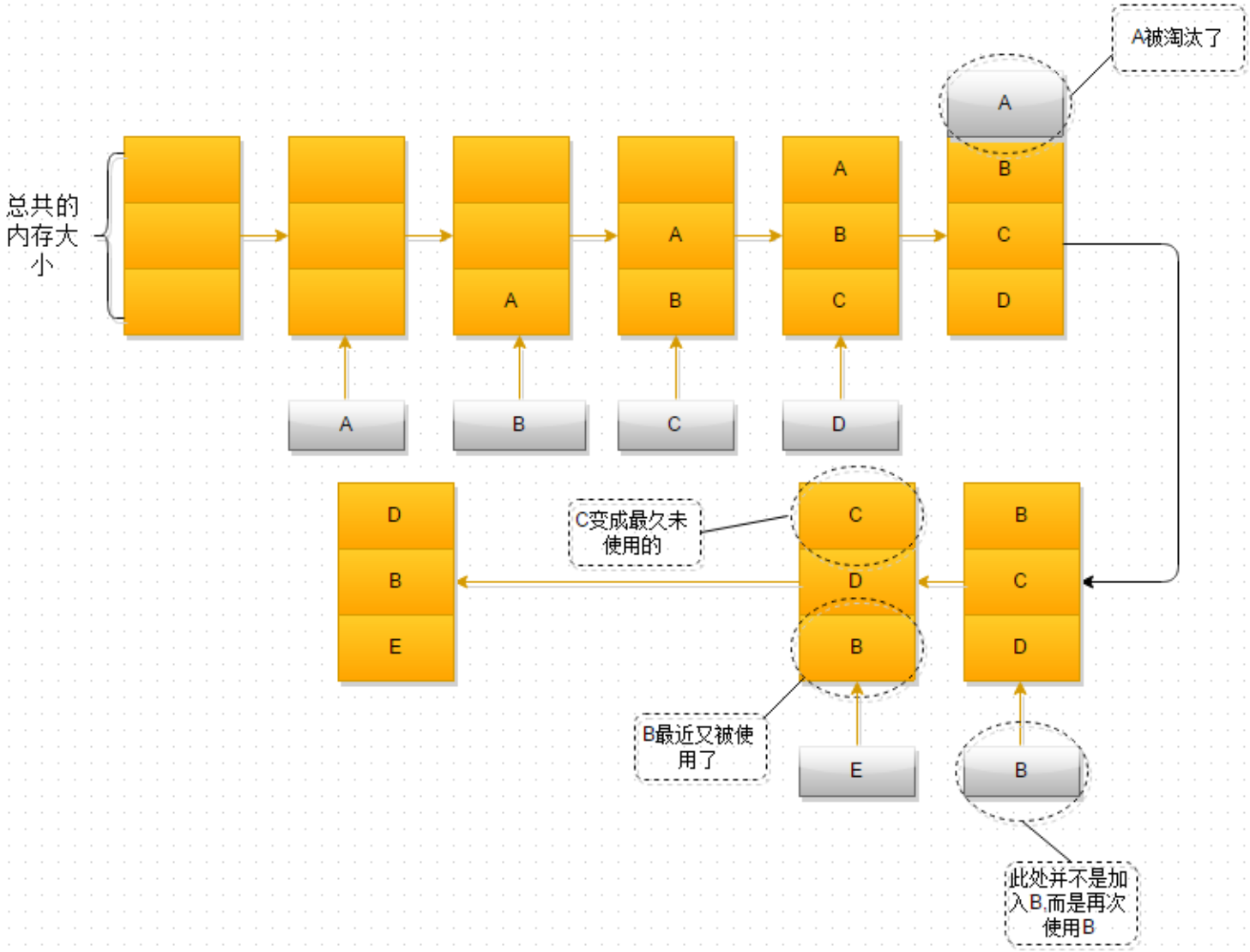
LRU

由于内存大小是有限的，我们不可能将读入的页总是留在内存中，则必然要采取一种策略来将不用的页冲回。最简单的一种缓存淘汰算法算法便是LRU（Least Recently Used）。LRU的核心思想是：如果数据最近被访问过，那么将来被访问的几率也更高。它的具体算法如下[9]：

首先维护一个存储了各个页的表示（如指针）的链表，重复执行：

1. 将新页插入到链表头部；
2. 每当缓存命中（即在内存中该页被访问），则将页移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

如下图：



优化

假设这样一种情况：页A是个被频繁访问的热点页，但是它在某一段时间里没被访问，尽管它在之后很有可能继续被频繁访问。如果在这段时间中有新读入的页，那么A就有可能会被冲回磁盘。这种缓存命中率下降的情况叫做“缓存污染”。

为了减少这种情况的出现，可以提升页访问的次数上限，当达到 k（k ≥ 2）次时才能够替换其他的页。因而普通的 LRU 便是 k=1 的情形。

Multiple Buffer Pools

顾名思义，便是将一个内存开辟为多个缓冲区。它的好处主要在于减少不同线程对锁的竞争，因为不同线程的操作由于缓冲区不同而减少了互相干扰。具体的分区方案是比较灵活的，可以依据页的类型或是不同的数据库来划分。

Scan Sharing

字面含义便是共享同一个扫描的结果，它的核心思想是：如果一个线程的查询需要扫描一个表，同时另一个线程恰好也要扫描这张表，那么没有必要两个线程同时进行相同的操作。反

之，我们可以只保留一个线程的操作，并共享扫描结果（比如页数据）。

脏页

将要淘汰一个页时，判断一个页是否被修改了显得十分重要。假如一个页没被修改，那么只需要简单地将其从缓冲区中删除即可；如果被修改了，那么就要冲回磁盘，将新的数据持久化。

为了解决这个问题，引入了“脏页”（dirty page）的概念。每个缓冲区中的页都有一个脏页标志位，并且有一个单独的线程重复地对缓冲区中的各个页进行扫描（当然，每个周期的时长是需要合适地设置的）。当它扫描到脏页时，可以有2种选择：

1. 将脏页冲回磁盘；
2. 重置脏页标志位并保留在内存中。

API

通过对 Buffer Pool Manager API 的详细定义[10]，可以更好地了解它的功能：

```
class BufferPoolManager {
    public:
    /* Fetch requested page from the buffer pool. */
    Page *FetchPage(page_id_t page_id) {
    // 1.   Search page table for requested page P;
    // 1.1  If P exists, pin it and return;
    // 1.2  Otherwise, find a replacement page R from the free list or the
replacer;
    // 2.   If R is dirty, flush it to disk;
    // 3.   Delete R from the page table and insert P;
    // 4.   Read P from the disk into the buffer pool, pin the page, and
return pointer to P.
    }

    /* Unpin the page from the buffer pool. */
    bool UnpinPage(page_id_t page_id, bool is_dirty) {
    // 1.  If the requested page P is not in buffer pool, immediately return;
    // 2.  If is_dirty=true, mark the page as dirty;
    // 3.  Decrements the pin count; If pin count becomes 0, call
replacer.Unpin(P).
    }

    /* Flush the page to disk. */
    bool FlushPage(page_id_t page_id) {
    // 1.  Check the requested P is not in buffer pool, immediately return;
    // 2.  Ask disk manager to flush the page.
    }
```

```
    /* Creates a new page in the buffer pool. */
    Page *NewPage(page_id_t *page_id) {
    // 1.  Ask disk manager to allocate a new page P;
    // 2.  If all pages in the buffer pool is pinned, return nullptr;
    // 3.  Pick a victim page V from the free list or the replacer. Evict V;
    // 4.  Zero out V's correspoinding frame in memory, pin the page, and add
P to page table.
    }

    /* Deletes a page from the buffer pool. */
    bool DeletePage(page_id_t page_id) {
    // 1.   Ask disk manager to deallocate the page;
    // 2.   Search the page table for the requestd page P;
    // 2.1  If P doesn't exist, return true;
    // 2.2  If P exists, but has positive pin-count, return false;
    // 2.3  Otherwise, remove P from the page table. Add its frame to the
free list.
    }

    private:
    std::mutex latch;
    size_t pool_size_;
    Page *pages_;
    DiskManager *disk_manager_;
    std::unordered_map<page_id_t, frame_id_t> page_table_; // frame_id
indexes into pages_
    Replacer *replacer;
    std::list<frame_id_t> free_list;
}
```

项目分析

功能分析

首先，对最基本功能的要求是明确的。它包含下面几个方面：

1. 在Java程序中建立一棵B+树；
2. B+树需要支持 CRUD；
3. 可以多个线程同时操作这颗B+树（即并发地 CRUD）；
4. 可以读取最基础的 SQL 输入（如 CREATE SCHEMA、CREATE TABLE、INSERT INTO、SELECT），并将其解析为对B+树的操作。

以上已经符合挑战难度4，不过只是第一阶段。它还有几个缺陷：

1. 没有将数据库持久化到磁盘；
2. 每次都需要重新建树，并且B+树的生命周期仅等同于Java程序的生命周期；
3. 没有提供对 SQL 更复杂的支持。

因此，第二阶段的改进便是：

1. 将数据库建在磁盘上；
2. 更复杂的 SQL 语法支持（如 JOIN、WHERE、AND、ALTER TABLE）。

为什么没有包含第2点缺陷呢？因为将B+树的生命周期延长意味着完全改变每个数据的存储方式。在第一、二阶段中，我不需要将 Storage Manager 的概念应用上去，磁盘文件只需要存储各个记录就可以了。然而如果要持久化B+树，那么就要完整实现数据页的各个部分（如槽和额外信息）。另外非常重要的一点便是：前两阶段写的Java程序将有许多会作废。为什么这么说呢?

在前两阶段里，程序的处理逻辑是这样的：

1. 将磁盘上的各个数据文件读取进内存，为每个数据块生成一个叶结点对象；
2. 由这些叶结点建树，指针（指向子结点的指针以及叶结点之间构成链表的指针）是Java对象所具有的字段；
3. 读取 SQL，解析后调用B+树的对应方法；
4. 获取返回结果，打印到控制台（或命令行界面）；
5. 重复执行3-4步直到用户使程序结束。

然而，引入 Storage Manager 之后，指针已经存储在了每一页的“额外信息”中，这意味各个页已经是以B+树的结构存储在磁盘上的了。因此第1-2步的程序（也是前两阶段里比重相当大的程序部分）就无用了。实际上，本项目作为数据结构的实验课，重点便是在第1-2步（尽管第3-4步的麻烦程度完全不低于1-2，因为即使调用第三方库，SQL 的解析以及转换为对应的方法调用以及合适的数据结构也相当麻烦，因为已经不是简单的CRUD了）。所以引入第三阶段（也即Storage Manager）后，相当于重写了整个实验，因此对整个项目是颠覆性的改变。

第四阶段则是面向实际情形的最终阶段。所谓“实际情形”，就是要模拟大规模的并发和数据，因此会涉及到频繁的磁盘IO，或者说内存（缓冲区）与外存（磁盘）的交互，因而除了 Storage Manager 之外还需要引入 Buffer Pool Manager 来完善交互的逻辑。当实验走到这一步时，可以说B+树实现所占的比重仅是较小的一部分了。

项目计划

实际上，当我写到这里时，已经处在第二阶段的开发中。但是看着数据页那复杂的设计，再加上Java在操控硬件上天然的劣势，不禁感到能否实现第三阶段都是个未知数。举个例子，由于指针（也即下个记录的地址）存储在每个数据页上，因此程序必然要直接使用指针（或者是地

址偏移量）来寻址（即找到下一个数据页）。使用C的话，可以直接操纵指针的值（也即地址），然而Java便没有这么底层了。

那为什么最初选择了Java来开发呢？一是对于较大型的项目而言，还是OOP组织起来更为清晰；二是大一上时曾经历过C程序极难调试的痛苦；三是Java有好用的标准库，而C还要自己造轮子。因此，如果没有更好的方法的话，有可能第三阶段需要用C来重写，那么工程量就翻倍了。

综上，项目的目标是在良好完成第二阶段的基础上，尝试完成第三阶段；如果完成了第三阶段并且尚有余力和时间，则向第四阶段冲刺。至于具体的技术，已经确定会使用的有：

1. CRUD；
2. Bulk Loading（尽管经过咨询，这个项目的基础要求并不包含Bulk Loading，不过考虑到快速建树的效率，还是很有可能会选择实现）；
3. Latch Crabbing（支持并发所必需的性能优化——毕竟直接锁整棵树是不应被容许的）；
4. Latching 优化；
5. SQL 解析；
6. 基本 SQL 命令；
7. 主键索引；
8. 普通索引（需要支持回表，否则难以高效完成 SQL 的基本功能）；
9. 持久化数据库到磁盘。

结语

（温馨提示：该部分并非初期调研报告的正式内容）

说是结语，实则吐槽。本来不打算写太多，毕竟文书工作真的是累啊……结果还是几乎完全手敲了一万多字，泪目。技术介绍部分是报告的大头，不过虽然说是介绍技术，其实更多的还是融合了自己对它们的一些理解。比如 Latch Crabbing 究竟为什么可行，B+树的优势又究竟在哪里，等等。

虽然写得累，不过也算是有了点收获。其实在写报告之前，我就已经有点迫不及待地把B+树基本操作以及 Latch Crabbing 给写了。结果到把数据库存储到磁盘上就开始懵逼了——存储格式是啥？SQL 要怎么解析？数据要怎么组织？B+树的位置又给放到哪了？不仅 SQL 的解析（即使有第三方库可以调）写得很痛苦（IO这种事情真的是细节多得让人无奈），而且一想到 Storage Manager 要实现的话基本上整个项目要从底层重来就感到@#¥%……&*

不过在开始介绍 Storage Manager 时，终于初步理了理数据在磁盘上存储的逻辑。记录->页->槽->树，整个结构显得稍微清晰些了，也算是小有收获吧。

参考文献

1. https://en.wikipedia.org/wiki/B-tree ↩
2. Knuth, Donald (1998), Sorting and Searching, The Art of Computer Programming, vol. 3 (Second ed.), Addison-Wesley, ISBN 0-201-89685-0 ↩
3. https://en.wikipedia.org/wiki/B%2B_tree ↩
4. Comer, Douglas (1979). "Ubiquitous B-Tree". ACM Computing Surveys. 11 (2): 121–137. doi:10.1145/356770.356776. S2CID 101673 ↩
5. https://www.oracle.com/hk/database/what-is-oltp/#:~:text=What%20is%20OLTP%3F-,OLTP%20defined,sending%20text%20messages%2C%20for%20example. ↩
6. https://juejin.cn/post/6844903970989670414#comment ↩
7. https://mp.weixin.qq.com/s?__biz=MzI1MDU0MTc2MQ==&mid=2247484235&idx=1&sn=d0a8ae41c280de196f8f439561928f3f&chksm=e981e0e5def669f386ca16ee67c3658df8f2650bc74e1ee56d46ae0afc42a8cf8ae9d0b09f76#rd ↩
8. https://dl.acm.org/doi/pdf/10.1145/320263.320276 ↩
9. https://zhuanlan.zhihu.com/p/34989978 ↩
10. https://yinfredyue.github.io/database/3-buffer-pool-manager/ ↩