基本的 B+ 树数据结构

杨其臻

Jun 25, 2025

在数据库系统与文件系统的核心层，传统数据结构面临着严峻挑战。当数据规模超出内存容量时，二叉搜索树可能退化为 $O(n)$ 性能的链表结构，而哈希表则无法高效支持范围查询。B+ 树正是为解决这些问题而生的多路平衡搜索树，其设计目标直指「最小化磁盘 I/O 次数」与「优化范围查询性能」两大核心需求。作为 B 树家族的重要成员，B+ 树通过独特的数据分离结构，在 MySQL InnoDB、Ext4 文件系统等关键基础设施中承担索引重任。理解 B+ 树不仅是对经典算法的学习，更是掌握现代存储引擎设计原理的必经之路。

1 B+ 树的核心概念

B+ 树的架构设计围绕「磁盘友好」原则展开。其内部节点仅存储键值用于路由导航，所有实际数据记录都存储在叶节点层，这种「数据分离」特性显著降低了树的高度。通过设定阶数 m 控制节点容量，B+ 树确保每个节点至少包含 $\lceil m/2 \rceil - 1$ 个键值，至多 $m-1$ 个键值，从而维持「多路平衡」特性。最精妙的设计在于叶节点间通过双向指针连接成有序链表，这使得范围查询如 `SELECT * FROM table WHERE id BETWEEN 100 AND 200` 无需回溯上层结构即可高效完成。所有叶节点位于完全相同的深度，形成「绝对平衡」状态，保证任何查询的路径长度稳定为 $O(\log_m n)$。

```
class BPlusTreeNode:
    def __init__(self, order, is_leaf=False):
        self.order = order # 节点阶数，决定键值容量
        self.is_leaf = is_leaf # 标识节点类型
        self.keys = [] # 有序键值列表
        self.children = [] # 子节点指针（内部节点）或数据指针（叶节点）
        self.next = None # 叶节点专用：指向下一叶节点的指针
```

该节点类定义了 B+ 树的基础构件。`order` 参数决定树的阶数，直接影响节点容量上限 $m-1$ 和下限 $\lceil m/2 \rceil - 1$。核心区别在于 `is_leaf` 标志：内部节点的 `children` 存储子节点引用，形成树形导航结构；叶节点的 `children` 则关联实际数据，并通过 `next` 指针串联为链表。键值列表 `keys` 始终保持有序，这是高效二分查找的基础。

2 B+ 树的详细结构剖析

内部节点与叶节点承担截然不同的角色。内部节点作为「路由枢纽」，其键值 k_i 表示子树 $child_i$ 中所有键值的上界。例如键值 `[15, 30]` 意味着：第一个子树包含 $(-\infty, 15)$ 的键，第二个子树包含 $[15, 30)$ 的键，第三个子树包含 $[30, +\infty)$ 的键。叶节点则是「数据终端」，存储完整键值及其关联的数据指针（或内联数据）。特别需要

注意的是根节点的特殊性 —— 作为初始节点，其键值数量可低于 $\lceil m/2 \rceil - 1$，这是树生长过程中的过渡状态。结构约束规则确保树的平衡性。当节点键值数量达到上限 $m - 1$ 时触发分裂，低于下限 $\lceil m/2 \rceil - 1$ 时触发合并（根节点除外）。叶节点链表在范围查询中发挥关键作用：当定位到起始键所在叶节点后，只需沿 next 指针遍历链表即可获取连续数据块，避免了传统二叉树的中序遍历回溯。

3 B+ 树操作原理解析

查找操作从根节点开始，通过二分查找定位下一个子节点，直至叶节点。时间复杂度 $O(\log_m n)$ 看似与二叉树相同，但由于 m 值通常达数百（由磁盘页大小决定），实际高度远低于二叉树。例如存储十亿条数据时，二叉树高度约 30 层，而 $m = 256$ 的 B+ 树仅需 4 层，将磁盘 I/O 次数从 30 次降至 4 次。

```
def search(node, key):
    # 递归终止：到达叶节点
    if node.is_leaf:
        idx = bisect.bisect_left(node.keys, key)
        if idx < len(node.keys) and node.keys[idx] == key:
            return node.children[idx] # 返回数据指针
        return None # 键不存在

    # 内部节点：二分查找子节点位置
    idx = bisect.bisect_right(node.keys, key)
    return search(node.children[idx], key)
```

该搜索实现展示递归查找过程。bisect.bisect_right 返回键值应插入位置，对于内部节点即对应子节点索引。叶节点使用 bisect.bisect_left 精确匹配键值，返回关联的数据指针。

插入操作需维持节点容量约束。当叶节点溢出时，分裂为两个节点并提升中间键至父节点。例如阶数 $m = 4$ 的节点键值 [5,10,15,20] 插入 18 后溢出，分裂为 [5,10] 和 [15,18,20]，并将中间键 15 提升至父节点。若父节点因此溢出，则递归向上分裂。特殊情况下，根节点分裂会使树增高一层。

```
def split_leaf(leaf_node):
    mid = leaf_node.order // 2
    new_leaf = Node(leaf_node.order, is_leaf=True)

    # 分裂键值与数据
    new_leaf.keys = leaf_node.keys[mid:]
    new_leaf.children = leaf_node.children[mid:]
    leaf_node.keys = leaf_node.keys[:mid]
    leaf_node.children = leaf_node.children[:mid]

    # 更新叶链表
    new_leaf.next = leaf_node.next
    leaf_node.next = new_leaf
```

```
    # 返回提升键值和新节点引用
    return new_leaf.keys[0], new_leaf
```

叶节点分裂时，原节点保留前半部分键值，新节点获得后半部分。提升的键值为新节点的首个键值（非中间值），这是因 B+ 树内部节点键值始终代表右子树的最小边界。链表指针的更新确保顺序遍历不受分裂影响。

删除操作需处理下溢问题。当叶节点键值数低于 $\lceil m/2 \rceil - 1$ 时，优先向相邻兄弟节点借键。若兄弟节点无多余键值，则触发节点合并。例如删除导致节点键值为 [10,20]（$m = 4$ 时下限为 1），若左兄弟为 [5,6,8] 可借出最大值 8，调整后左兄弟 [5,6]，当前节点 [8,10,20]。合并操作将两个节点与父节点对应键合并，可能引发递归合并直至根节点。

```
def borrow_from_sibling(node, parent, idx):
    # 尝试从左兄弟借键
    if idx > 0:
        left_sib = parent.children[idx-1]
        if len(left_sib.keys) > min_keys:
            borrowed_key = left_sib.keys.pop()
            borrowed_child = left_sib.children.pop()
            node.keys.insert(0, parent.keys[idx-1])
            node.children.insert(0, borrowed_child)
            parent.keys[idx-1] = borrowed_key
            return True

    # 尝试从右兄弟借键（类似逻辑）
    ...
```

借键操作需同步更新父节点键值。向左兄弟借键时，父节点对应键值需更新为借出键值，因该键值代表左子树的新上界。这种同步更新机制容易出错，是 B+ 树实现中的常见陷阱点。

4 代码实现关键模块

节点类作为基础容器，需严格控制键值与子节点的对应关系。对于内部节点，keys 长度始终为 len(children) - 1，因为 n 个键值需要 $n + 1$ 个子节点指针。叶节点则保持 len(keys) == len(children)，每个键值对应一个数据项。

```
def insert(node, key, data):
    if node.is_leaf:
        # 叶节点插入
        idx = bisect.bisect_left(node.keys, key)
        node.keys.insert(idx, key)
        node.children.insert(idx, data)
```

```
        if len(node.keys) > node.order - 1: # 溢出检测
            new_key, new_node = split_leaf(node)
            if node.is_root: # 根节点特殊处理
                create_new_root(node, new_key, new_node)
            else:
                return new_key, new_node # 向上传递分裂
    else:
        # 内部节点路由
        idx = bisect.bisect_right(node.keys, key)
        child = node.children[idx]
        split_result = insert(child, key, data)

        if split_result: # 子节点发生分裂
            new_key, new_node = split_result
            node.keys.insert(idx, new_key)
            node.children.insert(idx+1, new_node)

            if len(node.keys) > node.order - 1:
                ... # 递归处理溢出
```

插入流程通过递归实现层次化处理。叶节点直接插入后检查溢出，分裂后若当前为根节点则创建新根。内部节点根据子节点分裂结果插入新键和指针，并递归检查自身溢出。这种「自底向上」的处理方式确保树始终保持平衡。

5 实战：B+ 树 vs B 树

B+ 树与 B 树的核心差异在于数据存储位置。B 树的内部节点存储实际数据，导致节点体积增大，降低缓存效率。而 B+ 树通过「数据集中于叶节点」的设计，使内部节点更紧凑，相同内存容量可缓存更多节点，显著减少磁盘访问。在范围查询场景，B+ 树的叶节点链表实现 $O(1)$ 跨节点遍历，而 B 树需复杂的中序遍历。此外，B+ 树所有查询路径长度严格相等，提供更稳定的性能表现。

6 应用案例：数据库索引

MySQL InnoDB 存储引擎采用 B+ 树实现聚簇索引，其叶节点直接包含完整数据行。这种设计使得主键查询只需一次树遍历即可获取数据。辅助索引（非聚簇索引）同样使用 B+ 树，但其叶节点存储主键值而非数据指针，通过二次查找获取数据。B+ 树的「高扇出」特性使得亿级数据表索引仅需 3-4 层深度，而叶节点链表结构使全表扫描转化为高效顺序 I/O 操作，这正是 `SELECT * FROM table` 语句的性能保障。

7 实现优化与常见陷阱

「批量加载」技术可大幅提升初始化效率。通过预先排序数据并自底向上构建树，避免频繁分裂，速度可提升 10 倍以上。并发控制需考虑锁粒度 —— 节点级锁虽简单但易死锁，B-link 树等变种通过「右指针」实现无锁读取。

常见实现陷阱包括：叶节点链表断裂（分裂/合并时未更新指针）、键值范围失效（删除后未更新父节点边界）、忽略根节点特殊规则（允许单键存在）等。边界测试需特别关注最小阶数（$m = 3$）和重复键值场景。

B+ 树以「空间换时间」的核心思想，通过多路平衡与数据分离的架构创新，成为大容量存储系统的基石。其价值在于同时优化点查询与范围查询，这在传统数据结构中难以兼得。理解 B+ 树不仅需要掌握分裂/合并等机械操作，更要领会其「面向磁盘」的设计哲学。随着新型存储硬件发展，LSM-Tree 等结构在某些场景展现优势，但 B+ 树在更新频繁、强一致性要求的系统中仍不可替代。