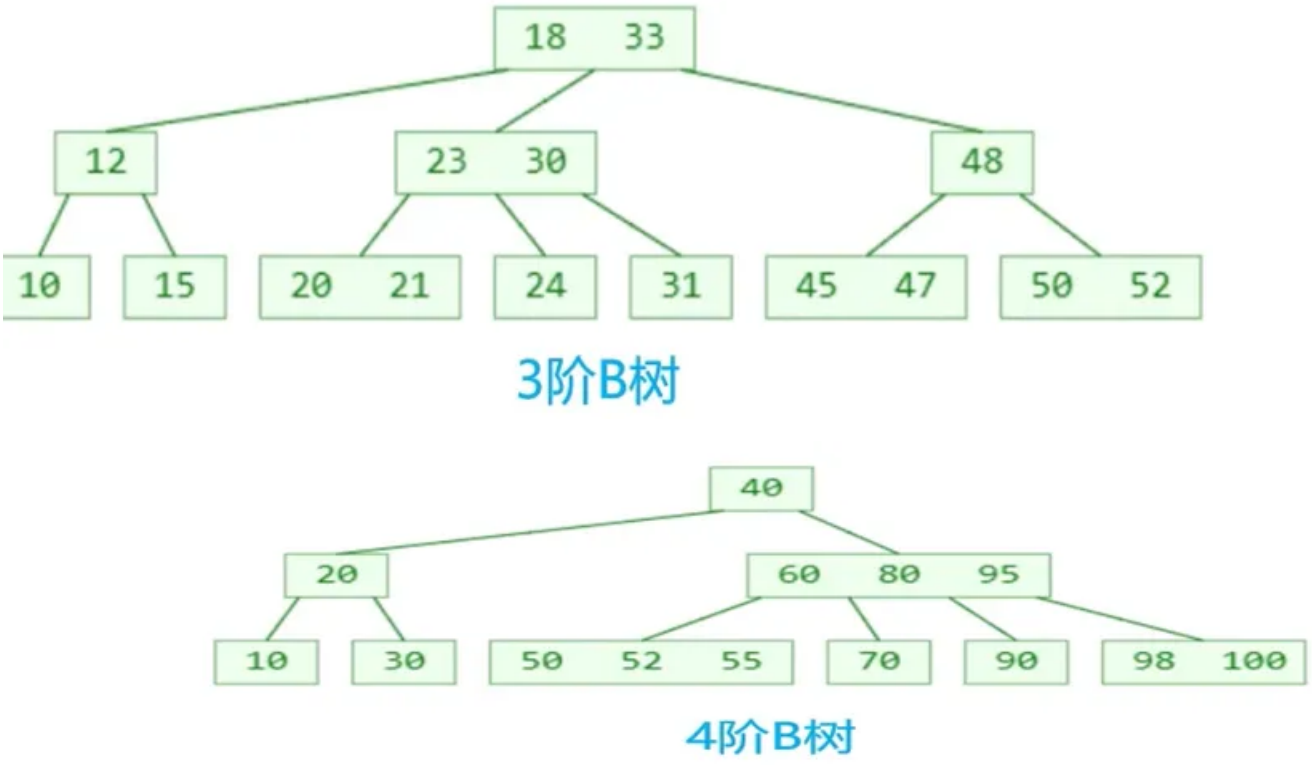
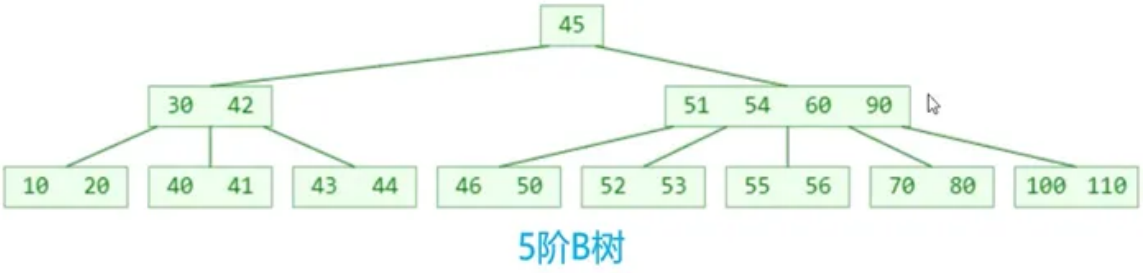
Btree索引

Btree的概述

B-tree 是一种自平衡的树数据结构，是多路搜索树用于文件系统，数据库的实现，维护有序数据。在数据库系统中，B-tree 索引通常用于加速数据检索操作，提高查询性能。通过在表的某一列上创建 B-tree 索引，可以使得数据库引擎快速定位到所需数据，从而加快查询和排序操作的执行速度。PostgreSQL 中默认使用 B-tree 索引，但也支持其他类型的索引，根据具体场景选择合适的索引类型可以最大限度地提升数据库性能。

下面来看下Btree的数据结构





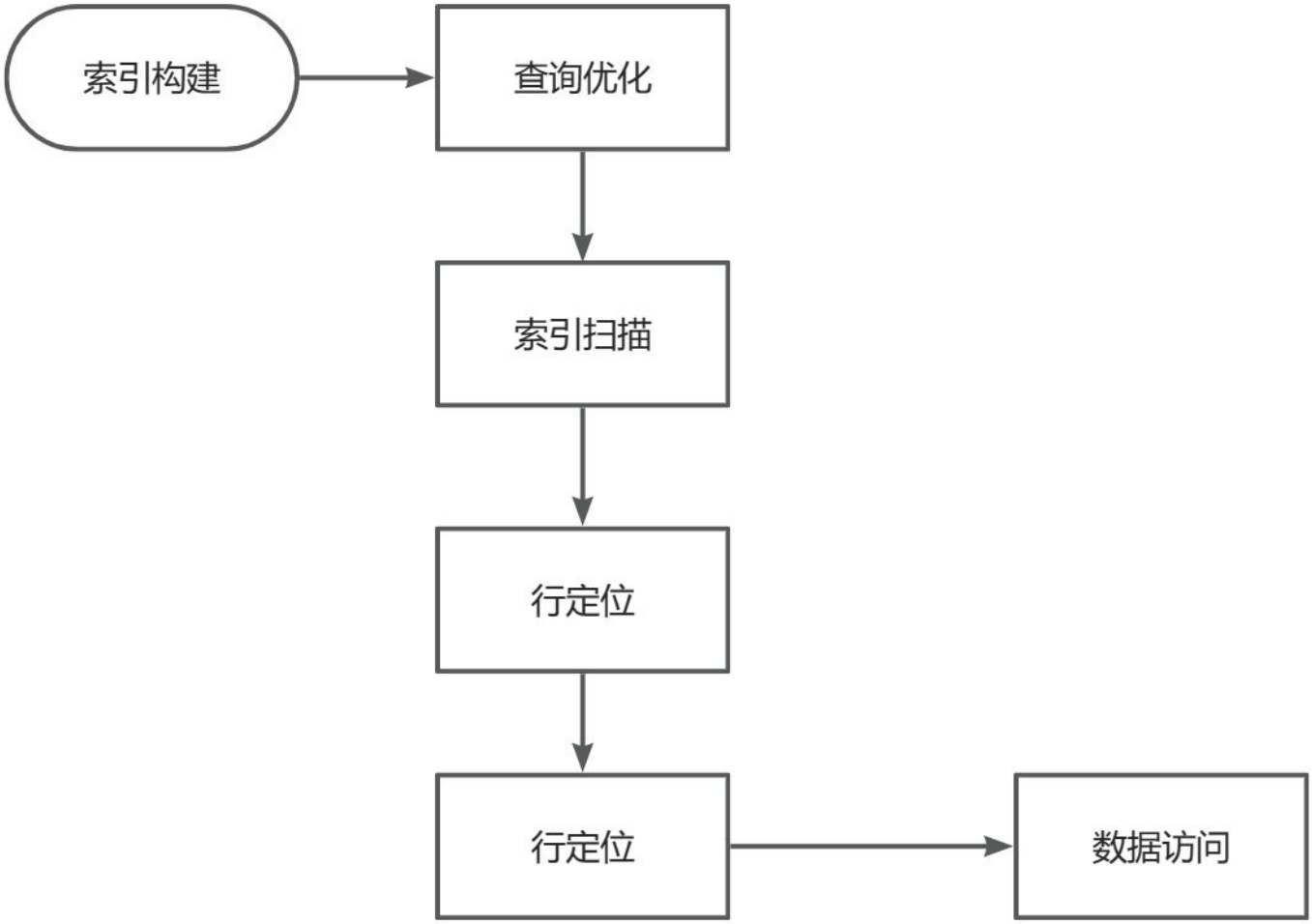
5阶B树

由此可以得出

B-tree 性质

- 假设一个节点存储的元素个数为 x
- 根节点：$1 \le x \le m - 1$
- 非根节点：$\lceil m/2 \rceil - 1 \le x \le m - 1$
- 如果有子节点，子节点个数 $y = x + 1$
  - ✓ 根节点：$2 \le y \le m$
  - ✓ 非根节点：$\lceil m/2 \rceil \le y \le m$
  - ➢ 比如 m = 3，$2 \le y \le 3$，因此可以称为（2, 3）树、2-3树
  - ➢ 比如 m = 4，$2 \le y \le 4$，因此可以称为（2, 4）树、2-3-4树
  - ➢ 比如 m = 5，$3 \le y \le 5$，因此可以称为（3, 5）树
  - ➢ 比如 m = 6，$3 \le y \le 6$，因此可以称为（3, 6）树
  - ➢ 比如 m = 7，$4 \le y \le 7$，因此可以称为（4, 7）树

Btree 索引的流程



1.索引构建： postgres中如果不指定索引构建方式，则默认Btree方式构建因为它适用于大多数情况下的查询，并且具有良好的性能。如果不特别指定索引类型。可以使用一下语法进行索引构建。

```sql
CREATE INDEX idx_name ON example_table USING btree (name);
```

2.查询优化： 当执行查询时，数据库系统首先会对查询进行优化，确定最佳的执行计划。如果查询涉及到了已经创建了 B-tree 索引的列，数据库系统会选择使用索引来加速查询。

3.索引扫描： 如果查询涉及到了 B-tree 索引的列，数据库系统会根据查询条件在索引树上进行搜索。搜索的过程类似于在一颗树上进行二分查找，根据比较结果决定向左子树或右子树移动，直到找到符合条件的数据或者搜索到叶子节点为止。

4.行定位： 一旦在索引树上找到了符合查询条件的索引条目，数据库系统会获取该条目对应的指针（通常是行号或行的物理地址），用于定位实际的数据行。

5.数据访问： 最后，数据库系统利用获取到的指针从表中检索出相应的数据行，并返回给用户。

性能评估

对于Btree，我们可以通过一些简单的sql来查看和评估btree的具体性能。

查询优化：在查询语句中使用 EXPLAIN 命令可以分析查询的执行计划，查看是否使用了 B-tree 索引。

```sql
EXPLAIN SELECT * FROM example_table WHERE name = 'John';
```

强制索引使用：在查询语句中使用 SET enable_indexscan = OFF; 可以禁用索引扫描，强制查询器选择顺序扫描表而不是使用索引。这有助于评估索引对查询性能的影响。

```sql
SET enable_indexscan = OFF;
SELECT * FROM example_table WHERE name = 'John';
```

内核分析

Btree 索引查找的函数流程 （伪代码）

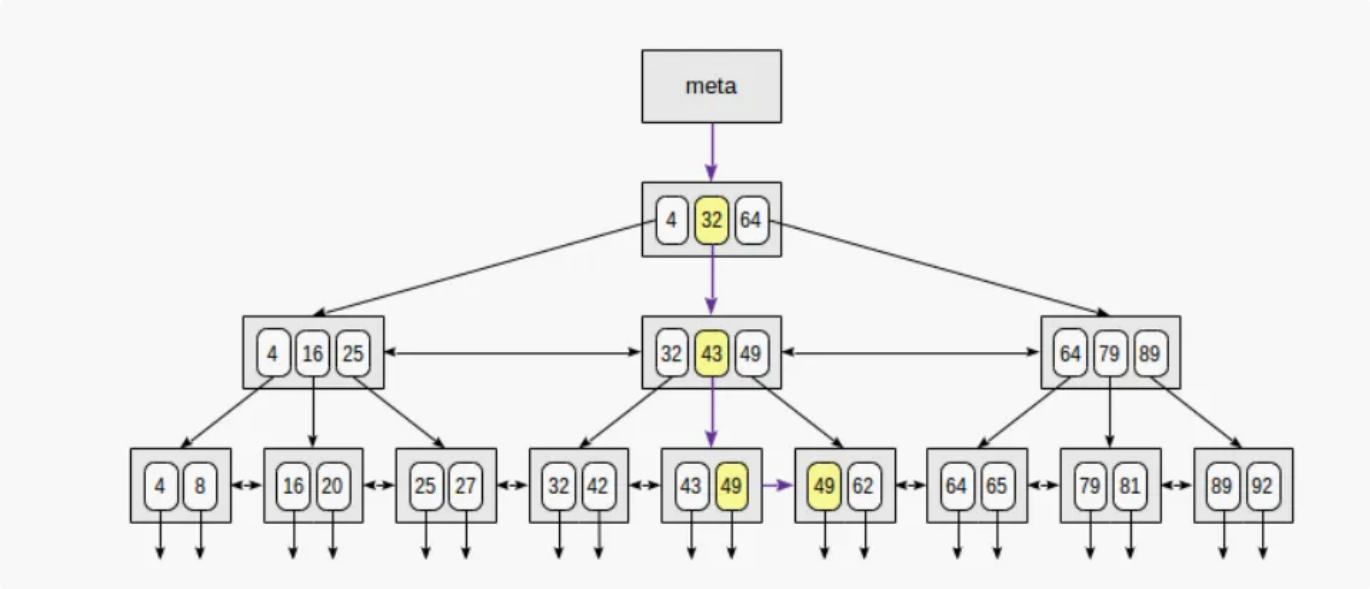
Btree 索引查找的函数流程

```
ExecScan         //索引扫描的流程
  ExecScanFetch
    IndexNext
       【1】index_beginscan:  (scandesc = index_beginscan 生成scandesc可以)
        index_beginscan_internal[getr]   //初始化准备函数，准备数据结构，相关参数
          btbeginscan: 初始化IndexScanDesc和BTScanOpaque
            IndexScanDesc scan = ...
              BTScanOpaque so = ...
              scan->opaque = so
            return scan
       【2】index_rescan      //如果需要重新扫描索引，那么就重新扫描
          btrescan

       【3】index_getnext(scandesc, direction)
        index_getnext_tid : 开始索引遍历，拿到一个符合要求的ctid (found = ...->amgettuple(scan, direction)
                            实际执行btgettuple返回true说明找到了符合条件的ctid)
          btgettuple      : 调用_bt_first拿到第一个符合条件的，再调用_bt_next顺序扫描后面符合条件的
            _bt_first[setr]:【重点分析】
        index_fetch_heap

       【3】index_getnext(scandesc, direction)
        index_getnext_tid : 开始索引遍历，拿到一个符合要求的ctid (found = ...->amgettuple(scan, direction)
                            实际执行btgettuple返回true说明找到了符合条件的ctid)
          btgettuple      : 调用_bt_first拿到第一个符合条件的，再调用_bt_next顺序扫描后面符合条件的
            _bt_next[setr]:【重点分析】
              _bt_steppage
                _bt_readnextpage
index_endscan
```

下面我们来看下关键函数之一　index_getnext_tid


meta
4
32
64
4
16
25
32
43
49
64
79
89
4
8
16
20
25
27
32
42
43
49
49
62
64
65
79
81
89
92

index_getnext_tid

```c
// ItemPointer: 用于获取下一个符合查询条件的数据行的位置标识符（TID）
// index_getnext_tid: 该函数用于在B树索引中获取下一个符合查询条件的数据行的位置标识符（TID）
// 参数:
//    - scan: 索引扫描描述符
        // -indexRelation：指向正在扫描的索引的指针，通过它可以获取索引的元数据信息。
        // -xs_ctup：当前索引条目的元组数据结构，包括位置标识符（TID）等信息。
        // -xs_cbuf：如果需要访问堆页以获取数据行，则可能会用到的缓冲区。
//    - direction: 扫描方向，指示向前还是向后扫描
// 返回值:
//    - 如果找到了符合条件的数据行，则返回该数据行的位置标识符（TID）；如果没有找到，则返回NULL

index_getnext_tid(IndexScanDesc scan, ScanDirection direction)
{
    bool found; // 用于标识是否找到符合条件的数据行

    // 调用索引访问方法（AM）的amgettuple函数，尝试获取下一个数据行
    found = scan->indexRelation->rd_amroutine->amgettuple(scan, direction);

    // 如果未找到符合条件的数据行，则执行以下操作
    // 索引扫描已经结束  or   查询条件不满足
    if (!found)
    {
        // 如果当前扫描缓冲区仍然有效，则释放该缓冲区并将其标记为无效
        // 这个操作的目的是清理掉不再需要的缓冲区，避免内存泄漏，并为后续的操作释放
        // 内存空间。因为在索引查找过程中，可能会多次切换和使用不同的缓冲区，
        // 释放不再需要的缓冲区是一种良好的资源管理实践。
        if (BufferIsValid(scan->xs_cbuf))
        {
            ReleaseBuffer(scan->xs_cbuf); // 释放当前扫描缓冲区
            scan->xs_cbuf = InvalidBuffer; // 将当前扫描缓冲区标记为无效
        }
        return NULL; // 返回空指针，表示未找到符合条件的数据行
    }

    // 如果找到了符合条件的数据行，则执行以下操作

    // 返回指向找到数据行位置标识符（TID）的指针
    return &scan->xs_ctup.t_self;
}
```

btgettuple C

```c

// btgettuple: 该函数用于在B树索引中获取下一个符合查询条件的数据行，并返回是否成功获取到数据行的布尔值
// 参数:
//   - scan: 索引扫描描述符
//   - dir: 扫描方向，指示向前还是向后扫描
// 返回值:
//   - 如果成功获取到符合条件的数据行，则返回true；如果没有找到或者出现错误，则返回false

btgettuple(IndexScanDesc scan, ScanDirection dir)
{
    // 获取B树索引扫描描述符中的BTScanOpaque数据结构
    BTScanOpaque so = (BTScanOpaque) scan->opaque;
    bool res; // 用于存储函数执行结果的布尔值

    // 进入循环，尝试获取下一个符合条件的数据行
    do
    {
        // 如果当前扫描位置无效，则调用_bt_first函数获取第一个符合条件的数据行
        if (!BTScanPosIsValid(so->currPos))
            res = _bt_first(scan, dir);
        else
        {
            // 否则调用_bt_next函数获取下一个符合条件的数据行
            res = _bt_next(scan, dir);
        }

        // 如果成功获取到了数据行，则跳出循环
        if (res)
            break;
        // 如果未获取到数据行，则检查是否还有更多的数组键需要处理
    } while (so->numArrayKeys && _bt_advance_array_keys(scan, dir));

    // 返回获取数据行的结果，即是否成功获取到了数据行的布尔值
    return res;
}

```

bt_first

```c
// _bt_first: 该函数用于在B树索引中定位到第一个符合查询条件的数据行，并返回是否成功定位到的布尔值
// 参数:
//    - scan: 索引扫描描述符
         // -indexRelation：指向正在扫描的索引的指针，通过它可以获取索引的元数据信息。
         // -xs_ctup：当前索引条目的元组数据结构，包括位置标识符（TID）等信息。
         // -xs_cbuf：如果需要访问堆页以获取数据行，则可能会用到的缓冲区。
//    - dir: 扫描方向，指示向前还是向后扫描
// 返回值:
//    - 如果成功定位到第一个符合条件的数据行，则返回true；如果未定位到或者出现错误，则返回false

_bt_first(IndexScanDesc scan, ScanDirection dir)
{
    Relation rel = scan->indexRelation; // 获取索引对应的关系对象
    BTScanOpaque so = (BTScanOpaque) scan->opaque; // 获取B树索引扫描描述符中的BTScanOpaque数据结构
    Buffer buf; // 用于存储数据块的缓冲区
    BTStack stack; // 用于存储B树中的路径栈
    OffsetNumber offnum; // 用于存储数据项（索引项）的偏移量
    StrategyNumber strat; // 用于存储扫描策略
    bool nextkey; // 用于标识是否需要获取下一个键
    bool goback; // 用于标识是否需要向前回溯
    ScanKey startKeys[INDEX_MAX_KEYS]; // 用于存储起始键
    ScanKeyData scankeys[INDEX_MAX_KEYS]; // 用于存储扫描键
    ScanKeyData notnullkeys[INDEX_MAX_KEYS]; // 用于存储非空键
    int keysCount = 0; // 用于计算扫描键的数量
    int i; // 循环变量
    bool status = true; // 用于存储函数执行结果的布尔值
    StrategyNumber strat_total; // 用于存储总体扫描策略
    BTScanPosItem *currItem; // 用于存储当前扫描位置的项
    BlockNumber blkno; // 用于存储数据块的编号

    // 断言当前扫描位置无效
    Assert(!BTScanPosIsValid(so->currPos));

    // 更新索引扫描统计信息
    pgstat_count_index_scan(rel);

    /*
     * Examine the scan keys and eliminate any redundant keys; also mark the
     * keys that must be matched to continue the scan.
     */
```

```
        _bt_preprocess_keys(scan);
    }
```

Question?

1.如何理解 B 树（B-tree）数据结构在 PostgreSQL 中的实现以及其对索引扫描性能的影响?

2.在index_getnext_tid这个函数中，为什么当found是false的时候要清空缓冲区

3.

第1题的回答：PostgreSQL 中的 B 树索引是一种平衡树，每个节点包含多个键值对，按照键值排序。树的高度相对较低，使得查找、插入和删除操作的时间复杂度为 O(log n)。节点分为内部节点和叶子节点，内部节点包含键和指向子节点的指针，叶子节点包含键和对应的数据指针。B 树的平衡性保证了树的高度相对较低，从而保证了高效的索引查找。