B+ Tree

- The most widely used index structure
- A balanced tree in which every path from the root to a leaf is of the same length.
  - Each non-roof has between $\lceil n/2 \rceil$ and $n$ children, where $n$ is a fixed value for a tree

- Supports equality and range searches, multiple attribute keys and partial key searches
- Either a secondary index (in a separate file) or the basis for an integrated storage structure
  - Responds to dynamic changes in the table
B⁺ Tree Structure

- Leaf level is a (sorted) linked list of index entries
- Sibling pointers support range searches in spite of
  - allocation and deallocation of leaf pages (but leaf pages might not be physically contiguous on disk)

Insertion and Deletion in B⁺ Tree

- Tree structure changes to handle row insertion and deletion – no overflow chains
- Tree remains balanced: all paths from root to index entries have same length
- Algorithm guarantees that the number of separator entries in an index page is between $\Phi/2$ and $\Phi$
  - Hence the maximum search cost is $\log_{\Phi/2} Q + 1$
  - Note - with ISAM search, the cost depends on length of overflow chain
**Handling Insertions - Example**

- Insert “vince”

![Diagram](image)

*FIGURE 9.19* Portion of the index of Figure 9.16 after insertion of an entry for vince.

---

**Handling Insertions (cont’d)**

- Insert “vera”: Since there is no room in leaf page:
  1. Create new leaf page, C
  2. Split index entries between B and C (but maintain sorted order)
  3. Add separator entry at parent level

![Diagram](image)

*FIGURE 9.19* Section of the index of Figure 9.10 after insertion of an entry for vince.

*FIGURE 9.20* Index sub-tree of Figure 9.19 after the insertion of vera has caused the split of a leaf page.
Handling Insertions (con’t)

- Insert “rob”. Since there is no room in leaf page A:
  1. Split A into A1 and A2 and divide index entries between the two (but maintain sorted order)
  2. Split D into D1 and D2 to make room for additional pointer
  3. Three separators are needed: “sol”, “tom” and “vince”

![Diagram](image1)

**FIGURE 9.21** Index subtree of Figure 9.20 after the insertion of rob has caused the split of a leaf page and of a separator page.

Handling Insertions (cont’d)

- When splitting a separator page, push a separator up
  - Repeat process at next level
  - * Might increase the height of tree by one

![Diagram](image2)

**FIGURE 9.22** B* tree that results from the insertion of vincent, vera, and rob into the index of Figure 9.16.
Handling Deletions

- Deletion can cause page to have fewer than $\varnothing/2$ entries
  - Entries can be redistributed over adjacent pages to maintain minimum occupancy requirement
  - Ultimately, adjacent pages must be merged, and if merge propagates up the tree, height might be reduced
  - See book

- In practice, tables generally grow, and merge algorithm is often not implemented
  - *Reconstruct tree to compact it*

Pseudocode for Insertion

A pseudocode rendering of the B* tree insertion algorithm. An entry in the index has the form $(k, P)$ where $k$ is a search key value and $P$ is a pointer, and the tree nodes have the form $(P, k_1, P_1, ..., k_n, P_n)$. In the figure, $\varnothing$ or $\#$ denotes deintersection of the actual tree nodes pointed to by the pointer $p$ and $\varnothing$ denotes the address of node.

```plaintext
proc insert(subtree, new, pushup)
    // Insert entry into subtree with root page "subtree" (new and subtree are pointers to nodes).
    The maximum number of separators in a page is $\varnothing$ (assumed to be even). pushup is null initially and upon return unless the node pointed to by subtree is split. In the latter case it contains a pointer to the entry that must be pushed up the tree. If the number of levels in the tree increases, "subtree" is the new root page when the outer level of recursion returns.

    if subtree is a non-leaf node
        (let's denote it $N = (P_0, k_0, P_1, ..., k_n, P_n)$)
        then
            let $n$ be the number of separators in $N$
            let $i$ be such that $k_i \leq$ (search-key value of new) $< k_{i+1}$
            or $i=0$ if (search-key value of new) $< k_1$
            or $i=n$ if (search-key value of new) $> k_n$
            insert($P_i$, new, pushup);
            if pushup is null return;
        else // then subtree has the form < key, pr >
            if $N$ has fewer than $\varnothing$ entries then // recall $N$ = "subtree"
                insert(subtree in $N$ in sorted order);
                pushup := null,
                return;
            else // $N$ has $\varnothing$ entries
                add pushup to a list of the entries in $N$ in sorted order
                split $N$: first $\varnothing/2 + 1$ entries stay in $N$,
                last $\varnothing/2$ entries are placed in new page, $N'$;
                pushup := $\#$; (smallest key value in $N'$, $\#$);
                if $N$ was the root of the entire B* tree then
                    create a new root-page $N''$ containing $\langle k(N), pushup \rangle$;
                subtree := $\#(N'')$;
                return;
    ```
Pseudocode for Insertion (Cont’d)

if subtrie is a leaf page (denoted L) then
  if L has fewer than Φ entries then
    insert new in L in sorted order;
    pushup := null;
    return;
  else  // L has Φ entries
    add &new to a list of the entries in L in sorted order
    split L: first (Φ/2) + 1 entries stay in L;
    the remaining Φ/2 entries placed in a new page, L’;
    pushup := &(<smallest key value in L’, &L’>);
    set sibling pointers in L, L’, and in the leaf page following L’;
  return;
endproc

Pseudocode for Deletion

FIGURE 9.24 A pseudocode rendering of the first part of the B^n tree deletion algorithm. An entry in the index has the form (<k, P>, where K is a search-key value and P is a pointer, and the tree nodes have the form (<k_0, P_0>, <k_1, P_1>, ..., <k_n, P_n>) in the figure, *ptr denotes dereferencing of ptr and &inode denotes the address of node.

proc delete(parentptr, subtree, oldkey, removeleaf)
// Delete oldkey from subtree with root *subtree.
// The minimum number of separators in a page is Φ/2 (Φ is assumed to be even).
// parentptr is null initially.
// But contains a pointer to the current index page of the caller thereafter.
// removeleaf is null initially and upon return, unless a child page has been
// deleted. In that case, removeleaf is a pointer to that deleted child.
// On return, *subtree is the (possibly new) root of the tree.

if subtree is a non-leaf node
  (henceforth denoted N = (P_0, <k_1, P_1>, ..., <k_n, P_n>) then
    let m be the number of separators in N;
    let l be such that k_l ≤ oldkey < k_{l+1};
    or i:=0 if oldkey < k_1 or i:=n if k_n ≤ oldkey;
    delete(subtree, P_i, oldkey, removeleaf); // a child page has been deleted
    if removeleaf is null then return; // no pages deleted in the process
  else  // a child page has been deleted
    remove separator containing removeleaf from N;
    if N is the root of the entire B^n tree then
      if N is not empty then return;
      else  // delete not root
        discard N;
        subtree := P_1;
        return;
    // N is not the root
Pseudocode for Deletion (Cont’d)

FIGURE 9.24 (continued)

If (number of entries in N) ≥ θ/2 then
  removedptr := null;
  return;
else  // N has fewer than θ/2 entries
  use parentptr to locate siblings of N;
  if N has a sibling, S, with more than θ/2 entries then
    if S is a right sibling of N then
      redistributefilel(parentptr, subtree, &S);  // recall that here *subtree = N
      removedptr := null;
      return;
    else  // S is a left sibling of N
      redistributeright(parentptr, &S, subtree);  // recall that here *subtree = N
      removedptr := null;
      return;
  else  // merge N and a sibling S
    choose a sibling, S;
    let M1 be the leftmost of the nodes N and S, and M2 the rightmost;
    removedptr := &M2;
    move all entries from M2 to M1;
    discard M2;
    return;

Pseudocode for Deletion (Cont’d)

FIGURE 9.25 A continuation of the pseudocode from Figure 9.24 for the procedure delete().

If *subtree is a leaf node (denoted L) then
  if oldkey is not in L then return;
  if (number of entries in L) > θ/2 then
    delete entry containing oldkey from L;
    removedptr := null;
    return;
else  // L has θ/2 entries
  delete entry containing oldkey from L;
  use sibling pointers to locate siblings of L;
  if L has a sibling, S, with more than θ/2 entries then
    if S is a right sibling of L then
      redistributefilel(parentptr, subtree, &S);  // here *subtree = L
      removedptr := null;
      return;
    else  // S is a left sibling of L
      redistributeright(parentptr, &S, subtree);  // here *subtree = L
      removedptr := null;
      return;
  else  // merge L and a sibling S
    choose a sibling, S;
    let M1 be the leftmost of the nodes L and S, and M2 the rightmost;
    removedptr := &M2;
    move all entries from M2 to M1;
    discard M2;
    adjust sibling pointers;
    return;
endproc
Pseudocode for Deletion (Cont’d)

FIGURE 9.26 A pseudocode for the procedure redistributeleft() called by the procedure delete() in Figure 9.24. The procedure moves a key from *rightptr to *parentptr and from *parentptr to *leftptr.

```plaintext
proc redistributeleft(parentptr, leftptr, rightptr)
    // Let e1 be entry in *parentptr containing rightptr; e1 = < k1, rightptr >
    // Let e2 be smallest entry in *rightptr; e2 = < k2, ptr >
    // Let p0 be the leftmost pointer in *rightptr
    add < k1, p0 > to *leftptr;
    delete e1 from *parentptr;
    add < k2, rightptr > to *parentptr;
    delete e2 from *rightptr;
    set the leftmost pointer in *rightptr to ptr;
endproc
```

Hash Index

- Index entries partitioned into buckets in accordance with a hash function, \( h(v) \), where \( v \) ranges over search key values
  - Each bucket is identified by an address, \( a \)
  - Bucket at address \( a \) contains all index entries with search key \( v \) such that \( h(v) = a \)
- Each bucket is stored in a page (with possible overflow chain)
- If index entries contain rows, set of buckets forms an integrated storage structure; else set of buckets forms an (unclustered) secondary index
Equality Search with Hash Index

- Given v:
  1. Compute $h(v)$
  2. Fetch bucket at $h(v)$
  3. Search bucket

- Cost = number of pages in bucket (cheaper than $B^+$ tree, if no overflow chains)

Choosing a Hash Function

- Goal of $h$: map search key values randomly
  - Occupancy of each bucket roughly same for an average instance of indexed table
- Example: $h(v) = (c_1 \ast v + c_2) \mod M$
  - $M$ must be large enough to minimize the occurrence of overflow chains
  - $M$ must not be so large that bucket occupancy is small and too much space is wasted
Hash Indices – Problems

- Does not support range search
  - Since adjacent elements in range might hash to different buckets, there is no efficient way to scan buckets to locate all search key values \( v \) between \( v_1 \) and \( v_2 \)

- Although it supports multi-attribute keys, it does not support partial key search
  - Entire value of \( v \) must be provided to \( h \)

- Dynamically growing files produce overflow chains, which negate the efficiency of the algorithm

Extendable Hashing

- Eliminates overflow chains by splitting a bucket when it overflows

- Range of hash function has to be extended to accommodate additional buckets

- **Example**: family of hash functions based on \( h \):
  - \( h_k(v) = h(v) \mod 2^k \) (use the last \( k \) bits of \( h(v) \))
  - At any given time a unique hash, \( h_k \), is used depending on the number of times buckets have been split
Extendable Hashing – Example

- Extendable hashing uses a directory (level of indirection) to accommodate family of hash functions
- Suppose next action is to insert sol, where \( h(sol) = 10001 \).
- **Problem:** This causes overflow in \( B_1 \).

![Diagram showing extendable hashing example](image)

---

Extendable Hashing – Example (Cont’d)

- **Solution:**
  1. Switch to \( h_3 \)
  2. Concatenate copy of old directory to new directory
  3. Split overflowed bucket, \( B \), into \( B \) and \( B' \); dividing entries in \( B \) between the two using \( h_3 \)
  4. Pointer to \( B \) in directory copy replaced by pointer to \( B' \)

![Diagram showing extendable hashing example (cont’d)](image)

Note: Except for \( B' \), pointers in directory copy refer to original buckets. The current hash identifies current hash function.
Extendable Hashing – Example (Cont’d)

- Next action:
  - Insert judy, where \( h(\text{judy}) = 00110 \)
  - \( B_2 \) overflows, but directory need not be extended

Problem:
When \( B_i \) overflows, we need a mechanism for deciding whether the directory has to be doubled

Solution:
\( \text{bucket\_level}[i] \) records the number of times \( B_i \) has been split. If \( \text{current\_hash} > \text{bucket\_level}[i] \), do not enlarge directory

FIGURE 9.31 Bucket \( B_2 \) of Figure 9.30 is split, without enlargement

Extendable Hashing

- Deficiencies:
  - Extra space for directory
  - Cost of added level of indirection:
    - If directory cannot be accommodated in main memory, an additional page transfer is necessary.
Choosing An Index

- An index should support a query of the application that has a significant impact on performance
  - Choice based on frequency of invocation, execution time, acquired locks, table size

Example 1: `SELECT E.Id
FROM Employee E
WHERE E.Salary < :upper AND E.Salary > :lower`

  - This is a range search on Salary.
  - Since the primary key is Id, it is likely that there is a clustered, main index on that attribute that is of no use for this query.
  - Choose a secondary, B^+ tree index with search key Salary

Choosing An Index (cont’d)

Example 2: `SELECT T.StudId
FROM Transcript T
WHERE T.Grade = :grade`

  - This is an equality search on Grade.
  - Since the primary key is (StudId, Semester, CrsCode) it is likely that there is a main, clustered index on these attributes that is of no use for this query.
  - Choose a secondary, B^+ tree or hash index with search key Grade.
Choosing an Index (cont’d)

Example 3:

```
SELECT T.CrsCode, T.Grade
FROM Transcript T
WHERE T.StudId = :id AND T.Semester = ‘F2000’
```

- Equality search on StudId and Semester.
- If the primary key is (StudId, Semester, CrsCode) it is likely that there is a main, clustered index on this sequence of attributes.
- If the main index is a B+ tree it can be used for this search.
- If the main index is a hash it cannot be used for this search. Choose B+ tree or hash with search key StudId (since Semester is not as selective as StudId) or (StudId, Semester)

- Suppose Transcript has primary key (CrsCode, StudId, Semester). Then the main index is of no use (independent of whether it is a hash or B+ tree). -- the order matters here.

Indexing on Flash Memory

- All discussions so far are about data stored on magnetic disks
- Should indexing techniques be modified for flash drive?
  - Flash memory support fast block load for index lookups
    - Takes microseconds instead of milliseconds to seek and read a random block.
    - So.... B+ tree node size can be smaller.
  - Drawback?
    - Data should be replaced logically instead of physical level
    - Every update is “copy + write” of an entire flash-memory block
      - Block erase time ~ 1 millisecond
      - So.... Issues are reducing the number of block erases.