B Trees, External Sorting and Hash Indices

Data/File Organizations -- Speeding Operations over Data

- Three general data organization techniques:
  - Indexing
  - Sorting
  - Hashing
- There is also the notion of a "heap", but that is data disorganization (or storage) rather than organization…
  - But, it is easy to maintain in the face of insertions and deletions. Just difficult to find things quickly.

Alternatives for Data Entry \( k^* \) in Index

- Three alternatives:
  1. Data record with key value \( k \)
     - Clustered \( \rightarrow \) fast lookup
     - Index is large; only 1 can exist
  2. \( \langle k, \text{rid of data record with search key value} \rangle \), OR
  3. \( \langle k, \text{list of rids of data records with search key} \rangle \)
     - Can have secondary indices
     - Smaller index may mean faster lookup
     - Often not clustered \( \rightarrow \) more expensive to use

- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value \( k \).

Algorithms & ‘Data’ Structures for DBMS file organization

- B-trees: multi-level index
  - Most commonly used database index structure today
- Hash index
  - ‘standard’ hash table concept
- External sorting algorithms
  - Sorting data residing on disk
  - Time complexity measured in terms of disk read/write
Recall: Indexing

- An index on a file speeds up selections on the search key attributes for the index (trade space for speed).
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of data entries, and supports efficient retrieval of all data entries \( k^* \) with a given key value \( k \).
  - Index record contains key \( k \) and a pointer (disk address) to the data record with that key value.

B-Trees and Indexing

- We have seen how multi-level indexes can improve search performance.
- One of the challenges in creating multi-level indexes is maintaining the index in the presence of inserts and deletes.
- We will learn B+-trees which are the most common form of index used in database systems today.

B-tree

- Class of balanced trees
- Forces all leaves to have same height
- Original motivation was search trees (not databases)
- Note: need to place some constraint that will force the tree to be balanced
  - This is accomplished by specifying the minimum and maximum number of entries at each node – the order \( d \) of tree
  - Alternately, can specify minimum and maximum number of children at each node – called the degree \( m \) of tree

B-trees Introduction

A B-tree is a search tree where each node has \( \geq n \) data values and \( \leq 2n \), where we chose \( n \) for our particular tree.

- Each key in a node is stored in a sorted array.
  - \( key[0] \) is the first key, \( key[1] \) is second key, …, \( key[2n-1] \) is the 2\( n \)-th key
- There is also an array of pointers to children nodes:
  - \( child[0], child[1], child[2], ..., child[2n] \)
  - Recursive definition: Each subtree pointed to by \( child[i] \) is also a B-tree.
- For any \( key[i] \):
  - 1) \( key[i] > \) all entries in subtree pointed to by \( child[i] \)
  - 2) \( key[i] \leq \) all entries in subtree pointed to by \( child[i+1] \)
- A node may not contain all key values.
  - \# of children = \# of keys + 1
- A B-tree is balanced as every leaf has the same depth.
**B-tree definition**

- Every tree node of B-tree of order \( d \) (\( d > 1 \)) must have:
  - (except the root) must have at least \( d \) key values (entries) sorted inside the node
  - Cannot have more than \( 2d \) key values
  - Has as many data pointers as key values
  - Has one more tree pointer than the number of key values; i.e., between \( d+1 \) and \( 2d+1 \)
  - Is either a leaf or an internal node

**B-tree Alternate definition**

- Every tree node of B-tree of degree \( m \) (\( m > 1 \)) must have:
  - (except the root) must have at least \( m-1 \) key values (entries) sorted inside the node
  - Cannot have more than \( 2m-1 \) key values
  - Has as many data pointers as key values
  - Has one more tree pointer than the number of key values; i.e., between \( m \) and \( 2m \)
  - Is either a leaf or an internal node

**B-trees Example**

- Programming View
  - Searching a B Tree: Example #1
    - B tree order 2 (1 or 2 keys at node)
      - Find 34
Searching a BTree Example #2

Height of B-tree of \( n \) nodes?

- Compute worst case height of B-tree of degree \( m \) (order \( m-1 \)), with total \( n \) nodes and height \( h \)
  - Root at level 0 has only 1 node
  - Level 1 has 2 nodes, each has at least \( m \) children
  - Level 2 has at least \( 2m \) nodes, each has at least \( m \) children
  - Level 3 has at least \( 2m^2 \) nodes, each has at least \( m \) children
  - ...
  - Level \( h \) has \( 2m^{(h-1)} \) nodes

Therefore \( n = 1 + 2 + 2m + 2m^2 + \ldots + 2m^{(h-1)} \)

\[ n = 1 + 2m - 1 + m \]

Therefore, \( h = O(\log_m n) \)

B-trees as External Data Structures

- Now that we understand how a B-tree works as a data structure, we will investigate how it can be used for an index.
- A regular B-tree can be used as an index by:
  - Each node in the B-tree stores not only keys, but also a record pointer for each key so the actual data being stored.
  - Could also potentially store the record in the B-tree node itself.
  - To find the data you want, search the B-tree using the key, and then use the pointer to retrieve the data.
  - No additional disk access is required if the record is stored in the node.
B-tree nodes & Index records

- B-tree is collection of blocks/nodes that contain
  - Search-key (index) values
  - Data pointers to data records
  - Tree pointers to next/children node
  - Some info local to block
- Each B-tree node resides on one disk block
  - When tree is traversed, relevant nodes/blocks are brought into main memory
- Given this description, how might we calculate the best B-tree order.
  - Depends on disk block and record size.
  - We want a node to occupy an entire block.

B-tree node

B-tree nodes

Nodes in B-tree arranged in in-order
each node has many entries

B-tree structure

(a) Node in B-tree with q-1 keys; (b) B tree with max 2 at node
Size of B-tree node

- What is the maximum number of bytes needed by a B-tree node of order \( d \)?
  - Tree pointers, data pointers, key fields, byte specific info
- What is the maximum number of bytes you can use for the node?

Picking the order of a B-tree

- Best "packing": when entire node fits into one disk block with minimum space wasted
- Assume \( P_d \) bytes needed to specify address of data pointer
- Assume \( P_t \) bytes needed for address of tree pointer
- Assume key field requires \( K \) bytes
- Assume disk block size = \( B \) bytes
- Assume \( x \) bytes needed for block-specific information

Order of B-tree

- Degree \( d \) tree has at most \( 2d+1 \) tree pointers, \( 2d \) data pointers and \( 2d \) key values
- \( x + 2d((K + P_d) + (2d+1)P_t) \leq B \)

Example

- \( B = 512 \) bytes, \( x = 2 \), \( K = 20 \), \( P_t = P_d = 4 \)
- \( 2 + 2d(20+4) + (2d+1)4 \leq 512 \)
  - \( 56d + 6 \leq 512 \)
  - \( d \leq 9 \)
Search in B-tree

Search for key=6

Inserting Data into a B Tree

- Find correct leaf L: search $O(\log N)$
- Put data entry into L.
- If L has enough space, done!
- Else, must split L (into L and a new node L2)
  - Redistribute entries evenly, push up middle key.
  - Insert index entry pointing to L2 into parent of L.
- This can happen recursively
  - Splits "grow" tree; root split increases height.
    - Tree growth: gets wider or one level taller at top.

Inserting 8 Example: $d=2$

- In-order search
  - Time complexity?
  - Height of tree $= \log(N)$

Want to insert here; no room, so split & push up:

Entry to be inserted in parent node.
Multilevel Index

B-tree and Multilevel Index

- Same?
- Different?

### B+ Tree: The DB World’s Favorite Index

- Insert/delete at log \_ N cost
  - (F = fanout, N = # leaf pages)
- Keep tree height-balanced
- Minimum 50% occupancy (except for root).
- Each node contains \( d \leq m \leq 2d \) entries. \( d \) is called the order of the tree.
- Supports equality and range searches efficiently.

### B+ Trees – Definition

- Similar structure to B-tree
- Distinguish between internal nodes and leaf nodes
  - Leaf nodes contain all search keys inserted into tree and connected by linked list in sorted order
    - Each leaf node:
      - Has no tree pointers
      - Has only search keys and data pointers
      - Has linked list pointer to next leaf node
    - Each internal node
      - Has search keys
      - Has tree pointers
      - Has no data pointers
  - Search keys in internal nodes used only for navigation
**B+ trees**
- Search keys in internal nodes are repeated in leaves
  - Similar to multilevel index
- Every search key does not occur as internal value
- To get the data record, we have to get to the leaf level
  - All searches will take O(h) where h is height of tree

**B+ trees: Definition**
- B+-tree of order d
  - Each internal node or leaf node must contain at least d keys
  - Each internal or leaf node can contain at most 2d keys
  - Each internal node can contain at most 2d+1 pointers to next node in tree
  - Each leaf node must contain as many data pointers as there are keys in the node
  - Each leaf node has pointer to next leaf node in linked list

---

**Example B+ Tree**
- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24*

Root

13 17 24 30

12 13 14 15 16 17 20 21 22 23 24 25 26 30 31 32 33

- Based on the search for 15*, we know it is not in the tree!

**Picking the order of a B+-tree**
- Same logic as for B-tree:
- Best “packing”: when entire node fits into one disk block with minimum space wasted
- Assume P bytes needed to specify address of data/tree pointer
- Assume key field requires K bytes
- Assume disk block size = B bytes
- Assume x bytes needed for block-specific information
Size of B+-tree node

- What is the maximum number of bytes needed by a B+-tree node of order \( d \)?
- Pointers, key fields, byte specific info
- What is the maximum number of bytes you can use for the node?

Order of B-tree

- Order \( d \) tree has at most \( 2d \) key values and \( 2d+1 \) pointers (data pointers and linked list pointer for leaf node, and tree pointers for internal node)
- \[ x + 2d(K) + (2d+1)P_t \leq B \]

Example

- \( B = 512 \) bytes, \( x = 2 \), \( K = 20 \), \( P_t = 4 \)
- \[ 2 + 2d(20) + (2d+1)4 \leq 512 \]
- \[ 48d + 6 \leq 512 \]
- \( d \leq 10.54 \)
- Higher than degree of B-tree!

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
- Average fanout = 133
- Typical capacities:
  - Height 4: \( 133^4 = 312,900,700 \) records
  - Height 3: \( 133^3 = 2,352,637 \) records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
**B+ trees**

- Insertion
- Deletion

**Inserting Data into a B+ Tree**

- Find correct leaf $L$.
- Put data entry onto $L$.
  - If $L$ has enough space, done!
  - Else, must split $L$ (into $L_1$ and a new node $L_2$)
    - Distribute entries evenly, copy up middle key.
    - Insert index entry pointing to $L_2$ into parent of $L$.
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
  - Splits “grow” tree; root split increases height.
  - Tree grows: gets wider or one level taller at top.

---

**Inserting $8^*$ into Example B+ Tree**

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Recall that all data items are in leaves, and partition values for keys are in intermediate nodes.
  - Note difference between copy-up and push-up.

---

**Inserting $8^*$ Example: Copy up**

- Want to insert here; no room, so split & copy up:
- Entry to be inserted in parent node. (Note that it is copied up and continues to appear in the leaf.)
**Inserting 8* Example: Push up**

Need to split node & push up

```
<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>14</th>
<th>16*</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>17</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>
```

Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)

**Deleting Data from a B+ Tree**

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has only d-1 entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

**B+ Tree Summary**

B+ tree and other indices ideal for range searches, good for equality searches.
- High fanout (F) means depth rarely more than 3 or 4.
- Almost always better than maintaining a sorted file.
- Typically, 67% occupancy on average.
- Note: Order (d) concept replaced by physical space criterion in practice ("at least half-full").
  - Records may be variable sized
  - Index pages typically hold more entries than leaves

**Other Kinds of Indices**

- Multidimensional indices
  - R-trees, kD-trees, …
- Text indices
- Inverted indices
- Structural indices
  - Object indices: access support relations, path indices
  - XML and graph indices: dataguides, 1-indices, d(k) indices
    - Describe parent-child, path relationships
Speeding Operations over Data

- Three general data organization techniques:
  - Indexing
  - Sorting
  - Hashing

Why Sorting?

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing gpa order
- Sort-merge join algorithm involves sorting.
- Internal sorting
  - Quicksort, heapsort, etc.
- Sorting Problem considered here: sort data much larger than main memory
  - Example: sort 1Gb of data with 1Mb of RAM.
    - Cannot hold all data in main memory – cannot use internal sort

Sorting Algorithms

- Quicksort, Heapsort
- Complexity?
- Use the same algorithm to sort our data files?

External Sorting

- Data to be sorted = data file stored on disk
- Sorted file must be stored back on disk

- External sorting problem:
  - Data file too large to fit in memory
  - Data must be sorted in pieces
  - Data usually heap size
  - Desired result: sorted file (sorted on some key field)
Mergesort: Recall

- Merge two sorted lists
  - (2, 5, 10, 19)
  - (4, 6, 7, 20)

- If we have two sorted files of length n/2 then binary merge gives sorted file of length n

External Mergesort

- File is set of pages/blocks

- Read one block at a time from disk into main memory
  - Assume 2 pages to hold a page from each input file and 1 page for output sorted block
Finally...

Observations

- How many disk I/Os? (how many read and write operations)
- Length of sorted file vs length of input file(s)?
External Sorting: Key ideas

- Individual blocks/pages can be easily sorted
  - Read them into memory and use internal sort algorithm
- Create runs
  - A run is a group/set of sorted blocks (i.e., sorted piece of a file)
  - Runs can be created by merging data from several blocks
- Merge shorter runs into longer runs
- Merge runs to get sorted file

External Mergesort

- What is the unsorted input?
  - Unsorted file of n blocks is n/2 groups of 2 blocks each
    - Each group has two blocks to be sorted
    - Sorting each group gives n/2 groups of runs, each run is 2 blocks
- What if we merge n/4 groups, each group of 2 sorted blocks?
  - n/4 groups, each of run 4 sorted blocks
  - How many disk I/Os?
  - What if we merge n/8 groups, each group of size 4 (i.e., 4 sorted blocks)
    - n/8 groups, each of size 8 blocks
    - How many disk I/Os?
  - What if we merge 2 groups, each group is run of size n/2
    - We get sorted file of n blocks
  - How many phases to get to this last step?

External Sorting

- Pass 1: Read a page, sort it, write it.
  - Only one buffer page is used
- Pass 2, 3, ..., etc.:
  - Three buffer pages used.

Two-Way External Merge Sort

- Input file
  - 1-page runs
  - 2-page runs
  - 4-page runs
  - 8-page runs
  - Disk
  - Main memory buffers
  - Disk
Two-Way External Merge Sort

- Each pass we read, write each page in file.
- N pages in the file => the number of passes = ⌈log₂ N⌉ + 1
- Total cost is:
  \[ 2N\left(\left\lceil \log_2 N \right\rceil + 1\right) \]
- Idea: Divide and conquer: sort subfiles and merge

General External Merge Sort

- How can we utilize more than 3 buffer pages?
- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce \( \frac{N}{B} \) sorted runs of B pages each.
  - Pass 2, ... etc.: merge \( B-1 \) runs.

M-way External Mergesort

- Divide file into groups of M blocks each
- Merge each group into a sorted run: N/M runs
- Create groups of N/M runs, merge each group into single run
- Example: N=400 block file, M=4 input buffers
  - Phase 1: create groups of 4 blocks
    - 100 groups of 4 blocks each = 100 runs of 4 blocks each
  - Phase 2: create groups of 4 runs; \( 100/4 = 25 \) groups
    - 25 groups of 16 blocks each = 25 runs of 16 blocks each
  - Phase 3: create groups of 32 runs; \( 25/4 = 7 \) groups
    - 7 groups of up to 32 blocks each = 7 runs of 32 blocks
  - Phase 4: groups of 64 runs; \( 7/4 = 2 \) groups
    - 2 groups, up to 64 blocks = 2 groups of 64 blocks
  - Phase 5: group of 128 runs; \( 2/4 = 1 \) group
    - 1 group, up to 128 blocks; 4-way merge gives 400 blocks
Cost of External Merge Sort

- Number of passes: $1 + \left\lfloor \log_2 N \right\rfloor$
- Cost = $2N \times (\# \text{ of passes})$

Speeding Operations over Data

- Three general data organization techniques:
  - Indexing
  - Sorting
  - Hashing

Technique 3: Hashing

- A familiar idea:
  - Requires “good” hash function (may depend on data)
  - Distribute data across buckets
  - Often multiple items in same bucket (buckets might overflow)
- Types of hash tables:
  - Static
  - Extendible (requires directory to buckets; can split)
  - Linear (two levels, rotate through + split; bad with skew)

Hash Files

- Basic idea: while domain is large, actual range of values is much smaller
  - If last name is 12 characters long, then domain is $12^{26}$ but actual number of names is much smaller
- In hashed file, records are distributed among a number of B buckets
- Hash function $h$, takes value as input and maps to one of B buckets
  - $h(k) = x, x \in \{0, \ldots, B-1\}$
Hashing as indexing technique

- Key idea: partition records among $B$ buckets

Hashing

- Hashing function $h$ applied to each record
  - Select “key” field (name, SSN, etc.) to hash
  - Each integer associated with a bucket number
    - $h(x) = 20$ means $x$ is in bucket 20
  - Initially a disk block to each bucket
  - As bucket gets large, add more blocks to bucket
  - Chain the blocks in each bucket
  - Hash table itself is stored on disk block(s)

Hashing: Buckets

Hash Functions

- Ideally, distribute the records evenly across buckets
  - As buckets get long, there are long overflow chains and it resembles heap
  - Dynamic hashing methods handle non-uniformity among buckets
  - Preserving order in hash function extremely difficult
    - If $x < y$, then $h(x) < h(y)$ --- not possible
### Operations on Hash files

- **Insert:** record with key field value $x$
  - Compute $h(x)$ to find bucket, and insert into bucket
    - If no space, then get new block
- **Delete:**
  - Compute $h(x)$ to find bucket, search for record, and delete
- **Search**
  - Compute $h(x)$ to find bucket, and search in the bucket (all blocks)

### Time analysis of Hash files

- Assume $B$ buckets, $n$ records
- Assume uniform distribution across buckets
- Number of blocks in each bucket is $n/Bp$
  - $p$ is blocking factor – no. of records per block
- Search/Lookup: compute $h(x)$ and search in bucket
  - Bucket is a Heap file of size $(n/Bp)$
  - Average time $\frac{1}{2}(n/Bp)$; worst case $(n/Bp)$
- Insert: compute $h(x)$ and insert into bucket in one disk access
  - 1 disk access
- Delete: search and delete

### Example:

- 1 million records
- 1000 buckets, $B=1000$
- $n/B = 1000$ records per bucket
- Blocking factor $p = 20$ (record size 200, block size 4096)
- Bucket size = $(1000/20) = 50$ blocks
- **Lookup time:**
  - Average time = 25
  - Worst case = 50
- **Insert time:** 1
- **Delete time:** same as lookup time

### Hashing and Sorting

- **Sorted files:**
  - Search on sorted field is fast
  - Insertion slow
  - Periodic reorganization needed
- **Hashing:**
  - Fast equality search
  - Bad for range search
  - Quick insert time
  - Challenge of finding a good hash function
    - Performance depends on how well records are divided amongst buckets