Frequently asked questions from the previous class survey

- Storage systems have gotten faster, how we moved to BSTs anywhere?
- Weakness of BSTs?
- Efficiency differences between recursive and iterative implementations of BSTs?
- Promoting child nodes?
Topics covered in this lecture

- B-Trees

BST: Predecessor and successors of a node

- Predecessor
  - Maximum value in its left subtree

- Successor
  - Minimum value in its right subtree
Wer Ordnung hält, ist nur zu faul zum Suchen. (If you keep things tidily ordered, you're just too lazy to go searching.) —German proverb

Databases and indices

- In order to efficiently find the value for a particular key in the database, we need a different data structure: an index.

- Key enabling idea
  - Keep some additional metadata on the side
  - Index acts as a signpost and helps you to locate the data you want

- If you want to search records in several different ways?
  - You may need several different indexes on different parts of the data
An index is an additional structure that is derived from the primary data

- Many databases allow you to add and remove indexes
  - This doesn’t affect the contents of the database; it only affects the performance of queries

- Maintaining additional structures incurs overheads during writes
  - For writes, it’s hard to beat the performance of simply appending to a file, because that’s the simplest possible write operation

- Any kind of index usually slows down writes
  - Because the index also needs to be updated every time data is written

Trade-off in storage systems

- Well-chosen indexes speed up read queries
  - but every index slows down writes

- For this reason, databases don’t usually index everything by default
  - Require you to choose indexes manually
    - Using your knowledge of the application’s typical query patterns
  - You can then choose the indexes that give your application the greatest benefit, without introducing more overhead than necessary
The most widely used indexing structure?

- The B-tree
  - Introduced in 1970 and called “ubiquitous” less than 10 years later
    - Inventors: Rudolf Bayer & Edward M. McCreight while at Boeing Research Labs
    - "B" was not defined: Could be for “balanced”, “broad”, “Boeing”?
  - B-trees have stood the test of time very well
  - They remain the standard index implementation in almost all relational databases, and many nonrelational databases use them too

Nodes in B-Trees

- Are usually also referred to as pages
- Very closely aligned with block-sizes on storage devices
B-Tree Nodes (or pages)

- The node contains several keys and references to child nodes
- Each child node is responsible for a **continuous range of keys**
  - The keys indicate where the boundaries between those ranges lie
- Most databases can fit into a B-tree that is **three or four levels deep**
  - So, you don’t need to follow many references to find the page you are looking for
  - A four-level tree of 4 KB pages with a branching factor (or fanout) of 500 can store up to 250 TB

---

B-Trees

We are braver than a bee, and a… longer than a tree...

Winnie the Pooh
How data is read from stable storage

- HDDs and SSDs address **blocks** rather than individual bytes
- Most operating systems have a block device abstraction
- When we’re reading a single word from an HDD or a SSD?
  - The whole block containing it is read

Primary limitation and design consideration for building efficient on-disk structures

- The **cost of disk access** itself
- The smallest unit of disk operation is a **block**
- To follow a pointer to the specific location within the block, we have to fetch an entire block
If we must always read a block?

- Why don’t we change the layout of the data structure to take advantage of it?

- Creating long dependency chains in on-disk structures greatly increases code and structure complexity
  - Much better to keep the number of pointers and their spans to a minimum

On-disk structures optimize for target storage specifics and optimize for fewer disk accesses

- Improving locality
- Optimizing the internal representation of the structure
- Reducing the number of out-of-page pointers
B-Trees combine these aforementioned ideas

- Account for storage characteristics (esp. the block construct)
- Increase node fanout
- Reduce tree height
- Reduce the number of node pointers
- Reduce frequency of balancing operations

B-Tree analogy: vast catalog room in the library

- You first have to pick the correct cabinet
- Then the correct shelf in that cabinet
- Then the correct drawer on the shelf, and
- Then browse through the cards in the drawer to find the one you're searching for

- Similarly, a B-Tree builds a hierarchy that helps to navigate and locate the searched items quickly
In most of the literature, **binary tree nodes are drawn as circles**
- Each node is responsible for just one key and splits the range into two parts
  - This level of detail is sufficient and intuitive

- B-Tree nodes are often drawn as **rectangles**
  - Pointer blocks are also shown explicitly to highlight the relationship between child nodes and separator keys
Keys inside the B-Tree nodes

- B-Trees are **sorted**
  - Keys inside the B-Tree nodes are stored in order
  - We can use an algorithm like *binary search* to locate a searched key

- This also implies that lookups in B-Trees have *logarithmic complexity*

Using B-Trees, we can efficiently execute both point and range queries

- **Point queries** locate a single item
  - Expressed by the equality (\(=\)) predicate in most query languages

- **Range queries** are used to query multiple data items in order
  - Expressed by comparison (\(<, >, \leq, \geq\)) predicates
The B-Tree Hierarchy comprises multiple nodes

- Each node holds up to N keys
  - And N + 1 pointers to the child nodes
- Nodes are logically grouped into three groups:
  - **Root** node, which is the top of the tree
  - **Leaf** nodes: Bottommost layer nodes that have no child nodes
  - **Internal** nodes: These are all other nodes with leaves
    - There is usually more than one level of internal nodes

B-Tree node hierarchy
Nomenclature

- Since B-Trees are a page organization technique
  - i.e., they are used to organize and navigate fixed-size pages
  - We often use terms node and page interchangeably
- The relation between the **node capacity** and the number of keys it actually holds is called **occupancy**

B-Trees are characterized by their fanout

- Fanout refers to the number of keys stored in each node
- **Higher fanout** helps to:
  - **Amortize** the cost of **structural changes** required to keep tree balanced
  - **Reduces the number of seeks** by storing keys and pointers to child nodes in a single block or multiple consecutive blocks
- Balancing operations (namely, splits and merges) are triggered when the nodes are full or nearly empty
Separator Keys

- Keys stored in B-Tree nodes are called index entries, separator keys, or divider cells

- **Split the tree into subtrees** (also called branches or subranges), holding corresponding key ranges
  - Keys are stored in sorted order to allow binary search

- A subtree is found by locating a key and following a corresponding pointer from the higher to the lower level

About pointers in a node

- The first pointer in the node?
  - Points to the subtree holding items less than the first key

- The last pointer in the node?
  - Points to the subtree holding items greater than or equal to the last key

- Other pointers?
  - Reference subtrees between the two keys: $K_{i-1} \leq K_s < K_i$, where $K$ is a set of keys, and $K_s$ is a key that belongs to the subtree.
Separator keys splitting a tree into subtrees

B-tree construction

- Rather than being built from top-to-bottom (as in BSTs), B-Trees are from bottom-to-top
- The number of leaf nodes grows, which increases the number of internal nodes and tree height
- B-Trees reserve extra space inside nodes for future insertions and updates
  - Tree storage utilization can get as low as 50%, but is usually much higher
  - Higher occupancy does not influence B-Tree performance negatively
B-Tree lookup complexity can be viewed from two standpoints

- The number of block transfers
- The number of comparisons done during the lookup

B-Tree lookup complexity: Number of block transfers

- In terms of number of transfers, the logarithm base is N (number of keys per node)
  - There are N times more nodes on each new level
    - Following a child pointer reduces the search space by the factor of N
  - During lookup, at most $\log_N M$ pages are addressed to find a target key
    - $M$ is the total number of items in the B-Tree
  - The number of child pointers that have to be followed on the root-to-leaf pass is also equal to the number of levels
    - In other words, the height $h$ of the tree
B-Tree lookup complexity: Number of block transfers

- B-Tree lookup complexity is generally referenced as $\log M$
- Logarithm base is generally not used in complexity analysis
  - Changing the base simply adds a constant factor
  - Multiplication by a constant factor does not change complexity
  - For example, given the nonzero constant factor $c$, $O(|c| \times n) = O(n)$

B-Tree lookup complexity: Number of comparisons

- From the perspective of number of comparisons
  - The logarithm base is 2
  - Since searching a key inside each node is done using binary search
- Every comparison halves the search space, so complexity is $\log_2 M$
Different ways to describe key and child offset counts [1/2]

- The original paper refers to device-dependent natural number $k$
  - Nodes, in this case, can hold between $k$ and $2k$ keys, but can be partially filled
  - Hold at least $k + 1$ and at most $2k + 1$ pointers to child nodes
- The root page can hold between 1 and 2$k$ keys
  - Later, a number $l$ is introduced, and it is said that any nonleaf page can have $l + 1$ keys

Different ways to describe key and child offset counts [2/2]

- Other sources, describe nodes that can hold up to $N$ separator keys and $N + 1$ pointers, with otherwise similar semantics and invariants
- Both approaches bring us to the same result
  - Differences are only used to emphasize the contents of each source
  - We stick to $N$ for clarity
**B-TREE LOOKUPS**

Separator keys splitting a tree into subtrees

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>31</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_s &lt; 12$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$12 \leq K_s &lt; 31$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$31 \leq K_s &lt; 45$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_s \geq 45$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Separator keys** splitting a tree into subtrees

- $K_1 \leq K_5 \leq K_2 \leq K_3$
- $K_5 < K_1$
- $K_2 \geq K_3$

---

**B-Tree: Example**

- Tree with keys: 4, 6, 7, 9, 17, 20, 24, 25, 28, 29, 30, 33, 37, 38, 40, 53, 55, 30
- Key 30 split into 25 and 38, 40, 55
- Key 30 split into 25 and 38, 40, 55
- Key 25 split into 20 and 28
- Key 25 split into 20 and 28
- Key 17 split into 9 and 20
- Key 17 split into 9 and 20
B-tree Lookup Algorithm: [1/4]

- To find an item in a B-Tree, we perform a single **traversal from root to leaf**
- The objective of this search is to find the key or its predecessor
  - Finding an exact match is used for point queries, updates, and deletions
  - Finding its predecessor is useful for range scans and inserts

B-tree Lookup Algorithm: [2/4]

- Index keys split the tree into subtrees
  - With boundaries between two neighboring keys
- The algorithm starts from the root and performs a binary search
  - This locates a subtree
- As soon as we find the subtree?
  - Follow the pointer that corresponds to it
  - Repeat search
B-tree Lookup Algorithm: [3/4]

- On each level, we get a more detailed view of the tree:
  - We start on the most coarse-grained level (the root of the tree)
  - Descend to the next level where keys represent more precise, detailed ranges
  - Until we finally reach leaves, where the data records are located

B-tree Lookup Algorithm: [4/4]

- During the point query
  - Search completes after finding (or failing to find) the target key

- During the range scan
  - Sibling pointers are followed until the end of the range is reached or the range predicate is exhausted
To insert the value into a B-Tree?

- First locate the target leaf and find the insertion point
- If the target node doesn’t have enough room available?
  - We say that the node has overflowed
  - Should be split in two to fit the new data
More precisely, the node is split if the following conditions hold

- For leaf nodes:
  - If the node can hold up to $N$ key-value pairs?
    - Inserting one more key-value pair brings it over its maximum capacity $N$

- For nonleaf nodes:
  - If the node can hold up to $N+1$ pointers?
    - Inserting one more pointer brings it over its maximum capacity $N+1$

Mechanics of splits

- Splits are done by allocating a new sibling node
  - Transferring half the elements from the splitting node to it
  - Adding its first key and pointer to the parent node
  - We say that the key is promoted

- The index at which the split is performed is called the split point
  - All elements after the split point (including split point in the case of leaf node split) are transferred to the newly created sibling node
  - The rest of the elements remain in the splitting node
Mechanics of splits

- If the parent node is full?
  - The parent has to be split as well
  - This operation might propagate recursively all the way to the root

Node splits are done in **four steps**

- Allocate a **new sibling node**
- **Transfer** half the elements from *splitting node* to the new *sibling node*
- **Place the new element** into either the new sibling or the splitting node
- At the parent of the split node?
  - Add a separator key and a pointer to the new sibling node
B-Tree: Leaf node Splits during insertion of 11

Before Split

<table>
<thead>
<tr>
<th>18</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

Splitting Node

After Split

<table>
<thead>
<tr>
<th>13</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

Splitting Node Newly added sibling node

Once the new sibling node is created, we use separator key invariants to decide where to insert the new element.

B-Tree: Nonleaf node splits during insertion of 11

Before Split

<table>
<thead>
<tr>
<th>8</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13</td>
<td>16</td>
</tr>
</tbody>
</table>

After Split

<table>
<thead>
<tr>
<th>8</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

Newly added sibling node
Deletions are done by first locating the target leaf. When the leaf is located, the key and the value associated with it are removed.

If neighboring nodes have too few values (i.e., their occupancy falls under a threshold)?

This situation is called an underflow. Sibling nodes are merged.
Merges: If two adjacent nodes have a common parent

- And their contents fit into a single node?
  - Their contents should be merged (concatenated)

- If their contents do not fit into a single node?
  - Keys are redistributed between them to restore balance

More precisely, two nodes are merged if the following conditions hold

- For leaf nodes
  - If a node can hold up to $N$ key-value pairs, and a combined number of key-value pairs in two neighboring nodes is less than or equal to $N$
    - We move elements from one of the siblings to the other one

- For nonleaf nodes:
  - If a node can hold up to $N + 1$ pointers, and a combined number of pointers in two neighboring nodes is less than or equal to $N + 1$
B-Tree: Leaf node merge during the deletion of 16

Before Merge

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

After Merge

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

Generally, elements from the right sibling are moved to the left one, but it can be done the other way around as long as key order is preserved.

Node merges are done in **three steps**, assuming the element is already removed

- Copy all elements from the right node to the left one
- Remove the right node pointer from the parent (or demote it in the case of a nonleaf merge)
- Remove the right node
B-Tree: Nonleaf node merge during the deletion of 10

Before Merge

After Merge

Rebalancing
Rebalancing

- To improve space utilization, instead of splitting the node on overflow
  - We can transfer some of the elements to one of the sibling nodes and make space for the insertion

- Similarly, during delete, instead of merging the sibling nodes
  - We may choose to move some of the elements from the neighboring nodes to ensure the node is at least half full

Tree balancing: Distributing elements between the more occupied node to the less occupied one

Before Balancing

After Balancing
B-TREE VARIANTS

B-Tree Variants

- B⁺-tree: B-tree where leaves form a linked list
- B*-tree: B-tree where nodes always 2/3 full
The contents of this slide-set are based on the following references
