CMPT 225

Lecture 35 – External Storage – Part 2 – B-Tree
Last Lectures

- We saw how to ...
  - Investigate various ways of managing data too large to fit in memory
  - Perform operations on such data
    - e.g.: searching using index files and data files
Learning Outcomes

At the end of these lectures, a student will be able to:

- **External Storage**
  - Perform operations on data too large to fit in memory
    - e.g.: searching using index files and data files

- **Data Organization**
  - Define B – Trees and B+ Trees
  - Demonstrate the functioning of their operations
Today’s menu

- Our goal in our next lectures is to
  - Define B-Tree
  - Demonstrate the functioning of their operations
B-Tree

- Definition:
  - A **B-Tree** is a “external” data collection that organizes its blocks \((B)\) into an \(m\)-way search tree, and in addition
    - the root of a **B-Tree** has at least 2 children (unless it is a leaf node)
    - and its other non-leaf nodes have at least \(\lceil m / 2 \rceil\) children
  
- Can be used to organize index files
**m-way tree?**

- Remember **n-ary tree**
  - 2-ary or binary tree / binary search tree
  - 3-ary tree
  - 4-ary tree
  - rooted tree in which each node has no more than **n** children
- **n-ary tree** is an **m-way tree**
- **n-ary search tree** is a **m-way search tree**
- **m** signifies the degree of tree
Example: $m$-way search tree ($m=4$)

- # of levels = 3

- Each non-leaf has at most 4 subtrees/children
- $4-1 = 3$ key values in ascending order
m-way search tree

- Definition:
  - An \( m \)-way search tree \( T \) is an \( m \)-way tree such that
  - \( T \) is either empty
  - OR
  - each non-leaf node of \( T \) has at most \( m \) children (subtrees) \( T_0, T_1, \ldots, T_{m-1} \)
  - and \( m - 1 \) search key values \( K \)'s in ascending order:
    \[
    K_1 < K_2 < \ldots < K_{m-1}
    \]
  - for every search key value \( V \) in subtree \( T_i \):
    \[
    \begin{align*}
    V &< K_1, & i &= 0 \\
    K_i &< V < K_{i+1}, & 1 &\leq i \leq m-2 \\
    V &> K_{m-1}, & i &= m-1
    \end{align*}
    \]
  - every subtree \( T_i \) is also an \( m \)-way search tree

Rules of construction
B-Tree

- A B-Tree is built from the leaves up, rather than from the root down, and so all leaf blocks in a B-Tree are on the same level.

- Hence, B-Trees are balanced m-way search trees, just as AVL trees are balanced binary search trees.
B-Tree Structure

- Each block contains a tree node
- In a node:
  - m - 1 index records:
    - `<search key, data file block #>`
  - m index file block #
    containing root of each of its subtrees/children
B-Tree Example

B-Tree of order 5 (m = 5) in which every node (except the root and the leaves) has
• at least \(\lceil 5 / 2 \rceil = 3\) children, and
• at most 5 children

Only showing

<search key, data file block #>

Children: index file block #
Back to
Search - Take 4 – Yet an even more efficient search

- A **B-Tree** is used as the data organization in the **index file**
Search - Take 4 - Using a B-Tree

1. Access (read into memory) block from index file containing the root

2. Linearly search block for target search key
   - If found: determine the matching data file block # and access that block from data file
     - If more than one data records per block, perform linear search to find target data record
     - If not found & block is leaf -> not there - done!

3. Otherwise, determine which index file block # to access next based on rules of construction of m-way search tree

4. Access block from index file and repeat above Steps 2 to 4
Search - Take 4 - Using a B-Tree ($m = 4$)
Constructing a B-Tree

- Let's construct the B-Tree shown on the previous slide where \( m = 4 \)
- To do so, we shall insert index records \(<\text{search key}, \text{data file block }\#>\) containing the following 18 search keys \((\text{elementCount} = 18)\):
  
  \[12, 1, 7, 23, 20, 6, 18, 5, 4, 22, 10, 15, 8, 3, 9, 17, 11, 16\]

Note: for space reason, we shall only insert the search key part of the index record

- Remember:
  - Index records \(<\text{search key}, \text{data file block }\#>\) are inserted in a block in ascending sort order of search key value
  - In a B-Tree, 1 block contains 1 node
B-Tree Insertion

- Let's begin by inserting element with search key 12:
  - **Data file**: Insert element (i.e., data record) into block \( b \) in data file
  - **Note**: this step will always be done the same way for each element so it shall not be repeated, simply implied
  - **Index file**: Since the B-Tree is empty, we create the first block i.e., a leaf, by inserting index record \( <12, b> \) into the block and inserting the block into index file

Drawing of this first block:

```
12
```

16
B-Tree Insertion

- **Insert 1:**
  - Compare each search key found in index record already in the node with the search key 1 and since $1 < 12$, move 12 over, then insert the index record $<1, \text{data file block #}>$ into it.

- **Insert 7:**
  - Compare each search key found in index records already in the node with the search key 7 and since $1 < 7 < 12$, move 12 over, then insert the index record $<7, \text{data file block #}>$ into the space made available.
B-Tree Insertion

- Insert 23:
  - Since the node is full, we split it as follows:
    - create a sibling and move the keys > middle search key (i.e., 7) into it
    - create a new block (parent) and move the middle search key (i.e., 7) into it
    - link the subtrees to the newly formed parent block using index file block #
B-Tree Insertion

- Insert 23 (cont’d):
  - Starting at the root, since $7 < 23$, 23 is inserted into its right subtree
  - Considering the root of its right subtree, since its only key $12 < 23$, insert the index record $<23, \text{data file block #}>$ after 12
B-Tree Insertion

- Insert 20:
  - Starting at the root, since 7 < 20, 20 is inserted into its right subtree
  - Moving on to the root of its right subtree, since 12 < 20 < 23, move 23 over, then insert the index record <20, data file block #> into the space made available
B-Tree Insertion

- Let's pick up the pace now...

- Insert 6, i.e., insert the index record <6, data file block #>

- Insert 18, i.e., insert the index record <18, data file block #>, but the destination block is full
B-Tree Insertion

- **Insert 18:**
  - Since the destination block is full, we split it as follows:
    - create a sibling and move the keys > middle search key (i.e., 20) into it
    - create a new block (parent) and move the middle search key (i.e., 20) into it
    - link the newly formed rightmost subtree to the parent block using index file block #
  - insert 18
B-Tree Insertion

- Insert 5, i.e., insert the index record \(<5, \text{data file block} \#>\)
B-Tree Insertion

- **Insert 4 - first split:**

  1  |  6  |  12  |  23
  ├───┼───┼───┼───
  │   │   │   │   
  ├───┼───┼───┼───
  │   │   │   │   
  ├───┼───┼───┼───
  │   │   │   │   

- **Insert the index record <4, data file block #>**
B-Tree Insertion

- Insert 22:
B-Tree Insertion

- Insert 10:
B-Tree Insertion

- Insert 15:

```
  7
 /   \
5     12 20
/       /    \
1  4  6  10  15 18
/   /   /   /   /   /
1  4  6  10  15 18 22 23
```
B-Tree Insertion

- Insert 8, 3, 9 and 17:
B-Tree Insertion

- Insert 11:
B-Tree Insertion

Insert 16:
Time Efficiency Analysis of Search - Take #4 - B-Tree

Assumption: the entire index file (B-Tree) cannot be loaded into main memory

- What is the time efficiency of searching disk-bound data using a B-Tree (index file)?

- In order to analyze the time efficiency, we need to know the height of a B-Tree accommodating 36M records

- Assuming we are using a B-Tree of order 4 to store our 36M search keys (and matching block #'s) and that each block of the B-Tree is filled (i.e., each block contains 3 index records) and that every level of our B-Tree is filled, then our B-Tree has:
In this example, we could increase the value of $m$, which would increase the number of index records in a block, which would decrease the height of our B-Tree, hence further reducing the number of disk accesses performed during a search of our data collection containing 36M Canadians.

Number of disk accesses proportional to the height of a tree (# of levels)
Advantage of B-Trees

- Good external data collection for index file and hence for disk-bound data
  - When \( n \) is large, \( m \) can be set to a large number, which keeps the height of the B-Tree short
  - Since the number of disk accesses is proportional to the height of a tree (\# of levels), then short height translates into small number of disk accesses, and hence good time efficiency for search/insert/remove operations

- In practice, commercial databases use specialized versions of these search trees where \( m \) is of the order of 100
Learning Check

- We can now ...
  - Define B-Trees
  - Demonstrate the functioning of their operations
Next Lectures

- B+ Trees