CMPT 225

Lecture 33 – External Storage – Part 1 – Disk Bound Data
Last Lectures

- We saw how to ...
  - describe another collision resolution strategy: chaining
  - discuss efficiency related to hashing
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
  - 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
So far ... 

... we have been assuming that the data collections our software solutions manipulate always completely fit in memory.
In practice ...

- This is not always a reasonable assumption

**Problem Statement**
- We are asked to develop a software application to maintain the records of all Canadians, allowing a user to perform a variety of operations on these records such as **update**

**Update:**
- Search for the record of a particular Canadian (using SIN as the search key)
- When found, update the record such as **making a change of address**

**Design** -> Data collection size?
class Canadian {

    private:
    
    string lastName;
    string firstName;
    string middleName;
    Address address;
    string SIN;
    ...

}
Disk-bound data

- Large data collections (databases), in which records are kept in files stored on external storage (such as hard disk), cannot be read entirely into main memory

- We refer to such data as disk-bound data
Searching disk-bound data

- Time efficiency of such search?
- Analysis: we know that
  - Accessing data stored in a file kept on the hard disk is extremely slow compared to accessing data in memory
    -> order of milliseconds ($10^{-3}$)
  - In contrast, accessing data in memory is fast
    -> order of nanoseconds ($10^{-9}$)
  - Million-to-1 ratio of disk access time versus memory access time
Searching disk-bound data -> expensive

- Accessing a disk (external storage) is expensive
- To search our 36M records efficiently, we will need to devise a way that minimizes the number of disk accesses performed
  - Because disk access is the most time consuming operation when manipulating (e.g., searching) records stored in external storage (e.g., hard disk)
Introducing “block”

- In order to devise a more efficient way to manipulate disk-bound data, let’s note that ...
  - Disk-bound data accessed by block
  
  - A block is a basic unit written to/read from external storage (disk)
  - Block size: 1kB to 64kB
Example of blocks

- This example illustrates a situation in which the elements of a binary tree have been stored in a file, in one possible “block” configuration.
File

- A file may contain many blocks
- All input/output done at block level rather than record (i.e., element) level

Different types of files

- Random access file
  - Like an array-based linear data collection
- Sequential access file
  - Like a link-based linear data collection
Search - Take 1

- We could store our 36M Canadian records in a disk file
- Let’s assume each block on disk contains only 1 record
- Time efficiency to search for target Canadian record
  - If our records are not sorted: linear search -> $O(n)$
  - Therefore, ________ disk accesses
Search - Take 1

If records !sorted by SIN, then linear search $\rightarrow O(n)$

36M disk accesses
Search - Take 2 – Improved search

- We could store our 36M Canadian records in a disk file accessed randomly
- Once again, let’s assume each block on disk contains only 1 record
- Time efficiency to search for that Canadian
  - If our records are sorted: binary search \( \rightarrow O(\log_2 n) \)
  - Therefore, \___________\ disk accesses
Search - Take 2

If records sorted by SIN, then binary search $\Rightarrow O(\log_2 n)$

$\therefore \log_2 36M \approx 25$ disk accesses
Search - Take 3 – An even more efficient search

- In Take 1 and in Take 2, we have stored our 36M Canadian records in a file
- In take 3, we shall call this file the **Data file** because it contains the data
- And we introduce a second file:
  - **Index file**: which contains a search data collection
    - In this solution (Take 3), the search data collection used is an AVL
- **Data** and **Index files** are both accessed randomly
Search - Take 3 – An even more efficient search
Search - Take 3 – An even more efficient search

- Each node of our AVL contains:
  - The index record <search key, data file block #>, where the search key is the search key of 1 Canadian record, and where data file block # is the block # of the block in the data file containing that Canadian record
  - The location of this AVL node’s left subtree
  - The location of this AVL node’s right subtree
Search - Take 3 – An even more efficient search
Search - Take 3 – An even more efficient search

Conceptual representation of content of index file
Search - Take 3 – An even more efficient search

- In analyzing the search time efficiency, we need to know how many levels an AVL tree (accommodating 36M records) has …

- Analysis:
  - We know that a complete binary tree, with its last level filled, has $2^H - 1$ nodes, where $H$ is the height of the tree (or $2^L - 1$ nodes, where $L$ is the number of levels in the tree)
  - **If we assume 1 AVL node deals with 1 data record**
    - $2^H - 1 = 2^L - 1 = 36M$
  OR (another way of arriving at the same answer)
  - We know that worst case scenario of the time efficiency for searching an AVL is $O(\log_2 n)$, which represents the # of levels
    - $\log_2 36M = L$
If the entire AVL tree stored in the Index file can be loaded into main memory when your program starts, then the “location of this AVL node’s left and right subtrees” will be the memory address of their respective root.

Block:
Search – Take 3 – Variation 1

- Time efficiency to search for a record will be:
  
  \[O(\log_2 n)\] comparisons (worst case)
  
  + 1 disk access to fetch the block in the data file that contains the desired record (using block # found above)
  
  + \(O(m)\) comparisons (worst case)

- Therefore, time efficiency to search for a particular Canadian out of 36M will be:
  
  -> about 25 comparisons + 1 disk access (data file) + \(O(1)\)

If \(m = 1\), i.e., 1 Canadian record per block -> no need to do any comparisons in order to look for the desired record in the block.
If the entire AVL tree stored in the Index file cannot be loaded into main memory, then each of its nodes, stored in a block, will contain as the “location of this AVL node’s left and right subtrees” the block # of the block in the Index file containing the root of the left/right subtree.

Block:
Search – Take 3 – Variation 2

To perform a search:

- The block containing the root of the AVL is first accessed: read from the Index file to memory
- AVL search algorithm is performed on node contained in that block
- The block # of the next AVL node (block in Index file) is determined and the block containing that node is accessed
- Above two steps are repeated until the desired search key is found or leaf of AVL is reached
  - If search key not found -> done!
  - If search key found -> the data file block containing the matching record is accessed using the block # of index records
Search – Take 3 – Variation 2

- Time efficiency to search for a record will be:
  
  \[ O(\log_2 n) \] disk accesses (worst case) and comparisons
  
  + 1 disk access to fetch the block in the data file that contains the desired record (using block # found above)
  
  + \( O(m) \) comparisons (worst case)

- Therefore, time efficiency to search for a particular Canadian out of 36M will be:
  
  -> about 25 disk accesses (index file)
  
  + 1 disk access (data file) + \( O(1) \)

If \( m = 1 \), i.e., 1 Canadian record per block -> no need to do any comparisons in order to look for the desired record in the block

for searching the AVL and finding the desired search key and block #
Search - Take 4 – Yet an even more efficient search

- In order to minimize the number of disk accesses, we need to minimize the height of our search tree or the number of levels in our search tree
  - This means we need to flatten our tree
- This can be achieved by increasing the number of records each node of our search tree can deal with
- A B-Tree can help ...
Learning Check

- We can now ...
  - 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
Next Lectures

- B-Trees