Data Structures and Algorithms for External Storage

Lecture delivered by:

Venkatanatha Sarma Y

Assistant Professor
MSRSAS-Bangalore
Session Objectives

• To study the differences in access characteristics between main memory and external storage devices such as disks
• To introduce algorithms for sorting files of externally stored data and to understand merge sorting
• To introduce data structures and algorithms for storing and retrieving information in externally stored files
• To introduce B-trees and understand the basic techniques for retrieving, inserting and deleting information in B-trees
Model of External Computation

• In all the algorithms the amount of input data is sufficiently small to fit in main memory at the same time is assumed.

• Suppose all the employees of the government are to be sorted by length of service or store all the information in the nation’s tax returns.

• In the above problem the amount of data to be proceed exceeds the capacity of the main memory.

• Most large computer systems have on-line external storage devices, such as disks or mass storage devices, on which vast quantities of data can be stored.

• These on-line storage devices, however have access characteristics that are quite different from those of main memory.
External Sorting
Sorting

Sorting Algorithms are divided into two categories:

• **Internal Sort**
  • Any sort algorithm which uses main memory exclusively during the sort
  • This assumes high-speed random access to all memory

• **External Sort**
  • Any sort algorithm which uses external memory, such as tape or disk, during the sort
External Sorting

• Among the more popular algorithms are
  – Tag Sorts
  – Four Tape Sort
  – Polyphase Sort
  – External Radix Sort
  – External Merge

• Polyphase sort is the most efficient in terms of speed and utilisation of resources.
  – However, it is also most complicated

• In practice these sorting methods are being supplemented by internal sorts
External Sorting

• A number of records from each disk would be read into main memory and sorted using an internal sort and then output to the disk

• Sorting data organised as files; or more generally, sorting data stored in secondary memory is called external sorting

• Utilization of certain powers of operating system to control the reading and writing of blocks at appropriate times can speed up sorting by reducing the time that the computer is idle, writing for a block to be read into or written out of main memory
External Merge Sorting

• In merge sort we organize a file into progressively larger runs, that is sequences of records \( r_1 \ldots r_k \) where the key of \( r_i \) is no greater than the key of \( r_{i+1} \) for \( 1 \leq i < k \).

• We say a file \( r_1 \ldots r_m \) of records is organized into runs of length \( k \) if for all \( i \geq 0 \) such that \( k_i \leq m \), \( r_{k(i-1)+1}, r_{k(i-1)+2} \ldots r_{k_i} \) is a run of length \( k \), and if \( m \) is not divisible by \( k \) and \( m = pk + q \), where \( q < k \), then the sequence of records \( r_{m-q+1}, r_{m-q+2} \ldots r_m \) called the tail, is a run of length \( q \).

• Example:- The sequence of integers is organised into runs of length 3

• The tail is of length less than 3, but consists of records in sorted order

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>15</td>
<td>29</td>
<td>8</td>
<td>11</td>
<td>13</td>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
External Merge Sorting

• The basic step of a merge sort on files is to begin with two files, \( f_1 \) and \( f_2 \) organised into runs of length \( k \).

• Assume that
  1. The numbers of runs, including tails, on \( f_1 \) and \( f_2 \) differ by at most one
  2. At most one of \( f_1 \) and \( f_2 \) has a tail
  3. The one with a tail has at least as many runs as the other

• Then it is a simple process to read one run from each of \( f_1 \) and \( f_2 \), merge the runs and append the resulting run of length \( 2k \) onto one of two files \( g_1 \) and \( g_2 \) which are being organised into runs of length \( 2k \)

• By alternating between \( g_1 \) and \( g_2 \) we can arrange that these files are not only organised into runs of length \( 2k \) but satisfy the above assumptions
External Merge Sorting

- To see that 2 and 3 are satisfied it helps to observe that the tail among the runs of $f_1$ and $f_2$ gets merged into the last run created.
- We begin by dividing all $n$ of records into two files $f_1$ and $f_2$ as evenly as possible.
- Any file can be regarded as organised into runs of length 1.
- Then we can merge the runs of length 1 and distribute them into files $g_1$ and $g_2$ organised into runs of length 2.
- We make $f_1$ and $f_2$ empty and merge $g_1$ and $g_2$ into $f_1$ and $f_2$ which will then be organised into runs of length 4.
- Then we merge $f_1$ and $f_2$ to create $g_1$ and $g_2$ organised into runs of length 8 and so on.
External Merge Sorting

• After i passes of this nature, we have two files consisting of runs of length $2^i$ if $2^i \geq n$, then one of the two files will be empty and the other will contain single run of length n, it will be sorted

• As $2^i \geq n$ when $i \geq \log n$, we see that $\lceil \log n \rceil$ passes suffice

• Each pass requires the reading of two files, all of length about $n/2$

• The total number of blocks read or written on a pass is thus about $2n/b$, where $b$ is the number of records that fit on one block

• The number of block reads and writes for the entire sorting process is thus $O(\left(\frac{n \log n}{b}\right))$
Merge Sorting Algorithm

merge(int out)
{
    int i, isml;
    typekey lastout;
    /* LastRec[] stores records from every input file */
    extern struct rec LastRec[];
    extern char FilStat[];
    lastout = Min_key;
    LastRec[0].key = Max_key;
    while (TRUE) {
        isml = 0;
        /*
         * Select the smallest record that is
         * no less than the last out
         */
        for (i=1; i<=maxfiles; i++)
            if (FilStat[i]=='i' && !Eof(i) &&
                LastRec[i].key >= lastout &&
                LastRec[i].key <  LastRec [isml]. key)
                isml = i;
Merge Sorting Algorithm

if (isml==0) { /* not found */
    for (i=1; i<=maxfiles; i++)
        if (FilStat[i]=='i' && !Eof(i)) return(0);
    return(DONE); /* all tapes exhausted */
}
WriteFile(out, LastRec[isml]);
lastout = LastRec[isml].key;
LastRec[isml] = ReadFile(isml);
}
} /* end of merge( ) */
File Based Storage and Retrieval
Storing Information in Files

- File is a sequence of records, where each record consists of the same sequence of fields.
- Fields can be either fixed length having a predetermined number of bytes or variable length having an arbitrary size.
- Files with fixed length records are commonly used in database management systems to store highly structured data.
- Files with variable length records are typically used to store textual information.
- The fixed-length techniques can be modified to work for variable-length records.
Storing Information in Files

• The operations on files are
  – INSERT a particular record into a particular file
  – DELETE from a particular file all records having a designated value in each of a designated set of fields
  – MODIFY all records in a particular file by setting to designated values certain fields in those records that have a designated value in each of another set of fields
  – RETRIEVE all records having designated values in each of a designated set of fields
A Simple Organisation

• The simplest, and also least efficient, way to implement the file operations is to use the file reading and writing primitives.

• In this organization records can be stored in any order.

• Retrieving a record with specified values in certain fields is achieved by scanning the file and looking at each record to see if it has the specified values.

• An insertion into a file can be performed by appending the record to the end of the file.

• For modifications of records, scan the file and check each record to see if it matches the designated fields and if so, make the required changes to the record.

• A deletion operation works the same way, but when we find a record whose fields match the values required for the deletion to take place, we must find a way to delete the record.
Speeding up File Operations

• The disadvantage of a sequential file is that file operations are slow. Each operation requires to read the entire file and some blocks may have to be rewritten as well.

• There are file organizations that allow to access a record by reading into main memory only a small fraction of the entire file.

• To make such organizations possible, each record of a file has a key, a set of fields that uniquely identifies each record is assumed.

• Another element for fast file operations is the ability to access blocks directly rather than running sequentially through the blocks holding a file.

• Many of the data structures for fast file operations uses pointers to the blocks themselves which are physical address of the blocks.
Hashed Files

- Hashing is a common technique used to provide fast access to information stored on secondary files.
- The records of a file are divided among buckets, each consisting of a linked list of one or more blocks of external storage.
Hashed Files

• There is a bucket table containing B pointers, one for each bucket
• Each pointer in the bucket table is the physical address of the first block of the linked-list of blocks for that bucket
• The buckets are numbered 0, 1, …, (B-1)
• A hash function h maps each key value into one of the integers 0 through (B-1)
• If x is a key, h (x) is the number of the bucket that contains the record with key x, if such a record is present at all
• The blocks making up each bucket are chained together in a linked list
• The header of the i^{th} block of a bucket contains a pointer to the physical address of the i+1^{st} block
• The last block of a bucket contains a NIL pointer in its header
Hashed Files

• If the size of the bucket table is small, it can be kept in main memory, otherwise it can be stored sequentially on as many blocks as necessary.

• To look for the record with key $x$, we compute $h(x)$ and find the block of the bucket table containing the pointer to the first block of bucket $h(x)$.

• Blocks of bucket $h(x)$ are read successively, until a block that contains the record with key $x$ is found.

• If all the blocks in the linked list is exhausted for bucket $h(x)$, we conclude that $x$ is not the key of any record.

• This structure is quite efficient if the operation is one that specifies values for the fields in the key, such as retrieving the record with a specified key value or inserting a record.
Indexed Files

- Indexed files maintains the file sorted by key values which is the another way to organize a file of records
- File can be searched as dictionary; or a directory can be used for scanning only the first key of block
- To facilitate the search, a second file, called a sparse index is created, which consists of pairs \((x, b)\) where \(x\) is a key value and \(b\) is the physical address of the block in which the first record has key value \(x\)
- Sparse index is maintained sorted by key values
Unsorted Files with a Dense Index

• Another way to organise a file of records is to maintain the file in random order and have another file called a dense index to help locate records.

• The dense index consists of pairs \((x, p)\) where \(p\) is a pointer to the record with key \(x\) in the main file.

• These pairs are sorted by key value so a structure like sparse index or the B-tree could be used to help find keys in the dense index.

• With this organisation, dense index can be used to find the location in the main file of a record with a given key.

• To insert a new record, track of the last block of the main file is kept and the new record is inserted there, getting a new block from the file system if the last block is full.
Secondary Indexes

• The hashed and indexed structures speed up operations based on keys substantially; but none of them help when the operation involves a search for records given values for fields other than the key fields.

• If records with designated values are to be found in some set of fields $F_1 \ldots F_k$, a secondary index on those fields is needed.

• A secondary index is a file consisting of pairs $(v, p)$ where $v$ is a list of values, one for each of the fields $F_1 \ldots F_k$ and $p$ is a pointer to a record.

• There may be more than one pair with a given $v$ and each associated pointer is intended to indicate a record of the main file that has $v$ as the list of values for the fields $F_1 \ldots F_k$. 
B-Trees
Where are trees used?

• Manipulate hierarchical data

• Make information easy to search

• Manipulate sorted lists of data
Access Time

• Information stored in high speed memory can be retrieved fast

• Information stored in disk takes longer to retrieve

• Information stored over the internet takes even longer to retrieve

• What can we do to reduce the overall access time to retrieve search results?
Reducing Access Times

• Make the height of the tree as small as possible
  – No empty sub trees appear above the leaves
  – All leaves be on the same level

• Every node (except leaves) have at least some minimal number of children
B-trees

- A B-tree of order m is an m-way tree in which
  - All leaves are on the same level
  - All internal nodes except the root have at most m nonempty children, and at least m/2 nonempty children
  - The number of keys in each internal node is one less than the number of its nonempty children, and these keys partition the keys in the children in the fashion of a search tree
  - The root has at most m children, but may have as few as 2 if it is not a leaf or none if the tree has only the root node
B-tree of Order 5
Insertion into a B-tree

- All insertions start at a leaf node
- To insert a new element
- Search the tree to find the leaf node where the new element should be added
- Insert the new element into that node with the following steps
- If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered
- Cont'd …
Insertion into a B-tree

• Otherwise, the node is full, so evenly split it into two nodes
  • A single median is chosen from among the leaf's elements and the new element.
  • Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
  • Insert the separation value in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).
Insertion into a B-tree

agfbkdhmjesirxclntup
Insertion into a B-tree

abfg → f → f → fj → fjr → j
ab gk abd ghkm abd gh km
abde ghi kmrs abde ghi km sx
ab de ghi klnm stux abc de ghi kl np stux
Deletion from a B-tree

- There are two popular strategies for deletion from a B-Tree.
  - locate and delete the item, then restructure the tree to regain its invariants
  - do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring
Deletion from a B-tree

- The algorithm below uses the former strategy.
  - There are two special cases to consider when deleting an element:
    - the element in an internal node may be a separator for its child nodes
    - deleting an element may put its node under the minimum number of elements and children.
Deletion from a B-tree

- **Deletion from a leaf node**
  - Search for the value to delete.
  - If the value is in a leaf node, it can simply be deleted from the node,
  - If underflow happens, check siblings to either transfer a key or fuse the siblings together.
  - if deletion happened from right child retrieve the max value of left child if there is no underflow in left child
  - in vice-versa situation retrieve the min element from right
Deletion from a B-tree

- Delete h, r
Deletion from a B-tree

- Delete p
Deletion from a B-tree

- Delete d
Deletion from a B-tree

- Delete d
B-tree heights

- N – Number of Nodes
- M – Order of B-tree
- Best case
  - $\log_m n$
- Worst case
  - $\log_{m/2} n$
Databases and B-trees

- Table – Student
  - student_id (primary key)
  - first_name
  - last_name
  - Email

- SQL query for retrieving all students with 'FirstName1' as their first_name

```sql
select * from student where first_name = 'FirstName1';
```
Databases and Files

• Generally databases are too large to fit in memory
  – Data is stored on the file system
  – E.g., RDBMS, collection of tables
  – Each table stored in a separate data file
  – Data files have records which map to the records in tables
  – Files are stored in blocks

• Database Indexes
  – Indexes are access structures that speeds up the retrieval of records
B-trees and Indexes

- Indexes
  - Primary
  - Secondary
  - Multilevel

- Index Characterization
  - Dense
  - Sparse
B-trees and Primary Indexes

• Primary Index
  – on the primary key, only one per table
  – ordered file with two fields (primary key, pointer to a disk block)
  – one entry in index for each block
  – first record is called anchor record or block anchor
  – sparse index
  – binary search used to retrieve records
B-trees and Primary Indexes

INDEX FILE

Block Anchor
Primary Key

Block Pointer

Abbas
Agarkar
Akthar
Anitha
Archana

Wahab
Williams

Abbas
Achyuth
Adam

Agarkar
Agarwal
Akram

Akthar
Alexander
Anil

Anitha
Anusha
Aravind

Archana
Arjun
Arnold

Wahab
Watson
Wazib

Williams
Wright
Zamal
B-trees and Primary Indexes

• Problems with ordered files
• Insertion and deletion
• What happens when a block is full?
• Insertion – overflow files or linked list of overflow records is a possible solution
B-trees and Secondary Indexes

- Secondary Index
  - in addition to the primary index
  - on any field in the table
  - more than one secondary index per table
  - can be on a column with unique or duplicate values
  - ordered file with two fields (column data, block pointer or record pointer)
  - dense index
  - e.g., secondary index on a column with distinct values
B-trees and Secondary Indexes
B-trees and Secondary Indexes

• Secondary vs. Primary Indexes
  – Secondary index needs more storage space
  – Longer search time because of more entries
  – Improvement in search time for an arbitrary record is much greater in a secondary index
  – Insertion and deletion of index records similar to primary indexes
B-trees and Multilevel Indexes

• Multilevel Indexes
  – an index that spans multiple levels
  – primary indexes do binary searches
  – primary index on \( b \) blocks \( \log_2 b \)
  – do better than base 2
B-trees and Multilevel Indexes

First Level
- 2
- 35
- 55
- 85

Index blocks = 4/4 = 1 block

Fanout = 4

Second Level
- 2
- 8
- 15
- 24
- 35
- 39
- 44
- 51
- 55
- 63
- 71
- 80

Index blocks = 13/4 = 4 blocks

Primary Key | Data File
--- | ---
2 | 5
8 | 12
15 | 21
24 | 29
35 | 36
39 | 41
44 | 46
51 | 52
55 | 58
63 | 66
71 | 78
80 | 82
85 | 89
89 |

26 records, 13 blocks
B-trees and Multilevel Indexes

• Number of keys in B-trees is order–1; Number of keys in multilevel indexes is order

• Node values are not repeated in B-trees, they are repeated in multilevel indexes

• Data pointers are at each node in B-trees, data pointers are only at the leaf level nodes in multilevel indexes
Summary

• On-line external storage devices, have access characteristics that are quite different from those of main memory. A number of data structures and algorithms are developed to utilize these devices effectively.

• The time to read data from a disk or tape is greater than the time spent doing simple computations with that data such as merging.

• Files with fixed length records are commonly used in database management systems to store highly structured data.

• Files with variable length records are typically used to store textual information.

• Hashing is a common technique used to provide fast access to information stored on secondary files.
Summary

• Indexed files maintains the file sorted by key values which is the another way to organize a file of records
• B-tree is a special kind of balanced m-ary tree that allows to retrieve, insert and delete records from an external file with a guaranteed worst-case performance
• The B-tree is particularly well suited for external storage and has become a standard organisation for indices in database systems.
• B-tree has become increasingly popular as a means of accessing information stored in database systems