Module 2

FILE ORGANIZATION FOR PERFORMANCE

3.1. Introduction

- Compression can reduce the size of a file, improving performance.
- File maintenance can produce fragmentation inside of the file. There are ways to reuse this space.
- There are better ways than sequential search to find a particular record in a file.
- Keysorting is a way to sort medium size files.
- We have already considered how important it is for the file system designer to consider how a file is to be accessed when deciding how to create fields, records, and other file structures. In this chapter, we continue to focus on file organization, but the motivation is different. We look at ways to organize or reorganize files in order to improve performance.
- In the first section, we look at how to organize files to make them smaller. Compression techniques make file smaller by encoding them to remove redundant or unnecessary information.

3.2. Data Compression

Data compression

The encoding of data in such a way as to reduce its size.

Redundancy reduction

Any form of compression which removes only redundant information.

- In this section, we look at ways to make files smaller, using data compression. As with many programming techniques, there are advantages and disadvantages to data compression. In general, the compression must be reversed before the information is used. For this tradeoff,
  - Smaller files use less storage space.
  - The transfer time of disk access is reduced.
  - The transmission time to transfer files over a network is reduced.

  But,
  - Program complexity and size are increased.
  - Computation time is increased.
  - Data portability may be reduced.
  - With some compression methods, information is unrecoverably lost.

    - Direct access may become prohibitably expensive.
    - Data compression is possible because most data contains redundant (repeated) or unnecessary information.

- Data compression is possible because most data contains redundant (repeated) or unnecessary information. Reversible compression removes only redundant information, making it possible to restore the data to its original form. Irreversible compression goes further, removing information which is not actually necessary, making it impossible to recover the original form of the data.
- Next we look at ways to reclaim unused space in files to improve performance. Compaction is a batch process that we can use to purge holes of unused space from a file that has undergone many deletions and updates. Then we investigate dynamic ways to maintain performance by reclaiming space made available by deletions and updates of records during the life of the file.

3.3 Compact Notation

Compact Notation

The replacement of field values with an ordinal number which index an enumeration of possible field values.

- Compact notation can be used for fields which have an effectively fixed range of values.
- Compact notation can be used for fields which have an effectively fixed range of values. The *State* field of the *Person* record, as used earler, is an example of such a field. There are 676 (26 x 26) possible two letter abbreviations, but there are only 50 states. By assigning an ordinal number to each state, and storing the code as a one byte binary number, the field size is reduced by 50 percent.
- No information has been lost in the process. The compression can be completely reversed, replacing the numeric code with the two letter abbreviation when the file is read. Compact notation is an example of redundancy reduction.
- On the other hand, programs which access the compressed data will need additional code to compress and expand the data. An array can used as a translation table to convert between the numeric codes and the letter abbreviations. The translation table can be coded within the program, using literal constants, or stored in a file which is read into the array by the program.

- Since a file using compact notation contains binary data, it cannot be viewed with a text editor, or typed to the screen. The use of delimited records is prohibitively expensive, since the delimiter will occur in the compacted field.

Run Length Encoding

An encoding scheme which replaces runs of a single symbol with the symbol and a repetition factor.

- Run-length encoding is useful only when the text contains long runs of a single value.
- Run-length encoding is useful for images which contain solid color areas.
- Run-length encoding may be useful for text which contains strings of blanks.
- Example:
- Uncompressed text (hexadecimal format):

  40 40 40 40 40 40 43 43 41 41 41 41 41 42
- Compressed text (hexadecimal format):

  FE 06 40 43 43 FE 05 41 42

where FE is the compression escape code, followed by a length byte, and the byte to be repeated.

Variable Length Codes

An encoding scheme in which the codes for differenct symbols may be of different length.

Huffman code: A variable length code, in which each code is determined by the occurence frequency of the corresponding symbol.

Prefix code: A variable length code in which the length of a code element can be determined from the first bits of the code element.

- The optimal Huffman code can be different for each source text.
- Huffman encoding takes two passes through the source text: one to build the code, and a second to encode the text by applying the code.
- The code must be stored with the compressed text.
- Huffman codes are based on the frequency of occurrence of characters in the text being encoded.
- The characters with the highest occurence frequency are assigned the shortest codes, minimizing the average encoded length.
- Huffman code is a prefix code.

  Example: Uncompressed Text:

  abdeacfaag (80 bits)

  Frequencies: a 4 e 1 b 1 f 1 c 1 g 1 d 1

Code: a 1   e 0001   b 010   f 0010   c 011   g 0011   d 0000

Compressed Text (binary):

10100000000110110010110011   (26 bits)

Compressed Text (hexadecimal):

A0 1B 96 60

3.4 Irreversible Compression Techniques

Any form of compression which reduces information.

Reversible compression

Compression with no alteration of original information upon reconstruction.

- Irreversible compression goes beyond redundancy reduction, removing information which is not actually necessary, making it impossible to recover the original form of the data.
- Irreversible compression is useful for reducing the size of graphic images.
- Irreversible compression is used to reduce the bandwidth of audio for digital recording and telecommunications.
- JPEG image files use an irreversible compression based on cosine transforms.
- The amount of information removed by JPEG compression is controllable.
- The more information removed, the smaller the file.
- For photographic images, a significant amount of information can be removed without noticably affecting the image.
- For line graphic images, the JPEG compression may introduce aliasing noise.
- GIF images files irreversibly compress images which contain more than 256 colors.
- The GIF format only allows 256 colors.
- The compression of GIF formatting is reversible for images which have fewer than 256 colors, and lossy for images which have more than 256 colors.
- Recommendation:
    - Use JPEG for photographic images.
    - Use GIF for line drawings.

3.5 Compression in UNIX

The UNIX *pack* and *unpack* utilities use Huffman encoding.

- The UNIX *compress* and *uncompress* utilities use Lempil-Ziv encoding.
- Lempil-Ziv is a variable length encoding which replaces strings of characters with numbers.

- The length of the strings which are replaced increases as the compression advances through the text.
- Lempel-Ziv compression does not store the compression table with the compressed text. The compression table can be reproduced during the decompression process.
- Lempel-Ziv compression is used by "zip" compression in DOS and Windows.
- Lempel-Ziv compression is a redundancy reduction compression - it is completely reversible, and no information is lost.
- The ZIP utilities actually support several types of compression, including Lempil-Ziv and Huffman.

3.6 Reclaiming Space in Files

External fragmentation: Fragmentation in which the unused space is outside of the allocated areas.

Compaction: The removal of fragmentation from a file by moving records so that they are all physically adjacent.

- As files are maintained, records are added, updated, and deleted.
- The problem: as records are deleted from a file, they are replaced by unused spaces within the file.
- The updating of variable length records can also produce fragmentation.
- Compaction is a relatively slow process, especially for large files, and is not routinely done when individual records are deleted.

Deleting Fixed- length Records for Reclaiming Space Dynamically

Linked list: A container consisting of a series of nodes, each containing data and a reference to the location of the logically next node.

Avail list: A list of the unused spaces in a file.

Stack: A last-in first-out container, which is accessed only at one end.

Record 1	**Record 2**	**Record 3**	**Record 4**	**Record 5**

- Deleted records must be marked so that the spaces will not be read as data.
- One way of doing this is to put a special character, such as an asterisk, in the first byte of the deleted record space.

Record 1	**Record 2**	*****	**Record 4**	**Record 5**

- If the space left by deleted records could be reused when records are added, fragmentation would be reduced.

- To reuse the empty space, there must be a mechanism for finding it quickly.
- One way of managing the empty space within a file is to organize as a linked list, known as the *avail list*.
- The location of the first space on the avail list, the head pointer of the linked list, is placed in the header record of the file.
- Each empty space contains the location of the next space on the avail list, except for the last space on the list.
- The last space contains a number which is not valid as a file location, such as -1.

Header	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6
3	* -1	Record 2	* 1	Record 4	Record 5	Record 6

- If the file uses fixed length records, the spaces are interchangable; any unused space can be used for any new record.
- The simplest way of managing the avail list is as a stack.
- As each record is deleted, the old list head pointer is moved from the header record to the deleted record space, and the location of the deleted record space is placed in the header record as the new avail list head pointer, pushing the new space onto the stack.

Header	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6
5	* -1	Record 2	* 1	Record 4	* 3	Record 6

- When a record is added, it is placed in the space which is at the head of the avail list.
- The push process is reversed; the empty space is popped from the stack by moving the pointer in the first space to the header record as the new avail list head pointer.

Header	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6
3	* -1	Record 2	* 1	Record 4	Record 5	Record 6

- With fixed length records, the relative record numbers (RRNs) can be used as location pointers in the avail list.

Deleting Variable-Length Records

- If the file uses variable length records, the spaces not are interchangable; a new record will not fit just any unused space.

- With variable length records, the byte offset of each record can be used as location pointers in the avail list.
- The size of each deleted record space should also be placed in the space.

Storage Fragmentation

Coalescence: The combination of two (or more) physically adjacent unused spaces into a single unused space.

Internal fragmentation: Fragmentation in which the unused space is within the allocated areas.

Placement Strategies

A policy for determining the location of a new record in a file.

First fit: A placement strategy which selects the first space on the free list which is large enough.

Best fit: A placement strategy which selects the smallest space from the free list which is large enough.

Worst fit: A placement strategy which selects the largest space from the free list (if it is large enough.)

- **First Fit**

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
370	* -1 70	Record	* 50 100	Record	* 200 60	Record

- The simplest placement strategy is *first fit*.
- With first fit, the spaces on the avail list are scanned in their logical order on the avail list.
- The first space on the list which is big enough for a new record to be added is the one used.
- The used space is delinked from the avail list, or, if the new record leaves unused space, the new (smaller) space replaces the olf space.
- Adding a 70 byte record, only the first two entries on the list are checked:

Header	Slot @50	Slot @120	Slot @200	Slot @230	Slot @300	Slot @370	Slot @430
370	* -1 70	Record	* 50 30	Record	Record	* 200 60	Record

- As records are deleted, the space can be added to the head of the list, as when the list is managed as a stack.

Best Fit

- The *best fit* strategy leaves the smallest space left over when the new record is added.
- There are two possible algorithms:

1. Manage deletions by adding the new record space to the head of the list, and scan the entire list for record additions.
2. Manage the avail list as a sorted list; the first fit on the list will then be the best fit.

- Best Fit, Sorted List:

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
370	* 200 70	Record	* -1 100	Record	* 50 60	Record

- Adding a 65 byte record, only the first two entries on the list are checked:

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
370	Record	Record	* -1 100	Record	* 200 60	Record

- Best Fit, Unsorted List:

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
200	* 370 70	Record	* 50 100	Record	* -1 60	Record

- Adding a 65 byte record, all three entries on the list are checked:

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
200	Record	Record	* 370 100	Record	* -1 60	Record

- The 65 byte record has been stored in a 70 byte space; rather than create a 5 byte external fragment, which would be useless, the 5 byte excess has become internal fragmentation within tbe record.
- **Worst Fit**
- The *worst fit* strategy leaves the largest space left over when the new record is added.
- The rational is that the leftover space is most likely to be usable for another new record addition.
- There are two possible algorithms:
    1. Manage deletions by adding the new record space to the head of the list, and scan the entire list for record additions.
    2. Manage the avail list as a reverse sorted list; the first fit on the list, which will be the first entry, will then be the worst fit.

- Worst Fit, Sorted List:

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
200	* 370 70	Record	* 50 100	Record	* -1 60	Record

- Adding a 65 byte record, only the first entry on the list are checked:

Header	Slot @50	Slot @120	Slot @200	Slot @235	Slot @300	Slot @370	Slot @430
50	* 370 70	Record	* -1 35	Record	Record	* 200 60	Record

- Worst Fit, Unsorted List:

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
200	* -1 70	Record	* 370 100	Record	* 50 60	Record

- Adding a 65 byte record, all three entries on the list are checked:

Header	Slot @50	Slot @120	Slot @200	Slot @235	Slot @300	Slot @370	Slot @430
200	* -1 70	Record	* 370 35	Record	Record	* 50 60	Record

Commonalities

- Regardless of placement strategy, when the record does not match the slot size (as is usually the case), there are two possible actions:
  1. Create a new empty slot with the extra space, creating external fragmentation.
  2. Embed the extra space in the record, creating internal fragmentation.

Finding Things Quickly: An Introduction to Internal Sorting and Binary Searching

Search by Guessing Binary search: A search which can be applied to an ordered linear list to progressively divide the possible scope of a search in half until the search object is found.

- Example: Search for 'M' in the list:

  A B C D E F G H I J K L M N O P Q R S
- Compare 'M' to the middle in the list:

  A B C D E F G H I **J** K L M N O P Q R S
- 'M' > 'J': Narrow the search to the last half. (Eliminate the first half.)

  A B C D E F G H I J **K L M N O P Q R S**
- Compare 'M' to the middle li in the remainder of the list:

A B C D E F G H I J **K L M N O P Q R S**

- 'M' < 'O': Narrow the search to the first half of the remainder. (Eliminate the last half.)

  A B C D E F G H I J **K L M N** O P Q R S

- Compare 'M' to the middle li in the remainder of the list:

  A B C D E F G H I J **K L M N** O P Q R S

- 'M' > 'L': Narrow the search to the last half. (Eliminate the first half.)

  A B C D E F G H I J K L **M N** O P Q R S

- Compare 'M' to the middle li in the remainder of the list:

  A B C D E F G H I J K L **M N** O P Q R S

  'M' == 'M': The search is over.

Binary Search versus Sequential Search

- A binary search of *n* items requires $\lfloor \log_2 n \rfloor + 1$ comparisons at most.
- A binary search of *n* items requires $\lfloor \log_2 n \rfloor + 1/2$ comparisons on average.
- Binary searching is $O(log_2\ n)$.
- Sequential search of *n* items requires n comparisons at most.
- A successful sequential search of *n* items requires n / 2 comparisons on average.
- A unsuccessful sequential search of *n* items requires n comparisons.
- Sequential searching is $O(n)$.
- Binary searching is only possible on ordered lists.

Sorting a Disk File in Memory

Internal sort: A sort performed entirely in main memory.

- The algorithms used for internal sort assume fast random access, and are not suitable for sorting files directly.
- A small file which can be entirely read into memory can be sorted with an internal sort, and then written back to a file.

The limitations of Binary Searching and Internal Sorting

- Binary searching requires more than one or two accesses.
- More than one or two accesses is too many.
- Keeping a file sorted is very expensive.
- An internal sort works only on small files.

Keysorting

Keysort: A sort performed by first sorting keys, and then moving records.

Description of the Method

- Read each record sequentially into memory, one by one
- George Washington 1789 1797 None
- John Adams 1797 1801 Fed
- Thomas Jefferson 1801 1809 DR
- James Madison 1809 1817 DR
- James Monroe 1817 1825 DR
- John Q Adams 1825 1829 DR
- Andrew Jackson 1829 1837 Dem
- Martin Van Buren 1837 1841 Dem
- William Harrison 1841 1841 Whig

Save the key of the record, and the location of the record, in an array (*KEYNODES*).

- Washington George 1
- Adams John 2
- Jefferson Thomas 3
- Madison James 4
- Monroe James 5
- Adams John Q 6
- Jackson Andrew 7
- Van Buren Martin 8
- Harrison William 9

After all records have been read, internally sort the *KEYNODES* array of record keys and locations.

- Adams John 2
- Harrison William 9
- Jackson Andrew 7
- Jefferson Thomas 3
- Madison James 4
- Monroe James 5
- Van Buren Martin 8
- Washington George 1

  Using the *KEYNODES* array, read each record back into memory a second time using direct access. Write each record sequentially into a sorted file.

Limitations of the Keysort Method

- Keysort is only possible when the *KEYNODES* array is small enough to be held in memory.
- Each record must be read twice: once sequentially and once directly.
- The direct reads each require a seek.
- If the original file and the output file are on the same physical drive, there will also be a seek for each write.
- Keysorting is a way to sort medium size files.

Another Solution: Why Bother to Write the File Back?

- Rather than actually sorting the data file, the *KEYNODES* array can be written to a disk file (sequentially), and will function as an index.
- The next chapter examines indexes.

Pinned Records

A record which cannot be moved without invalidating existing references to its location