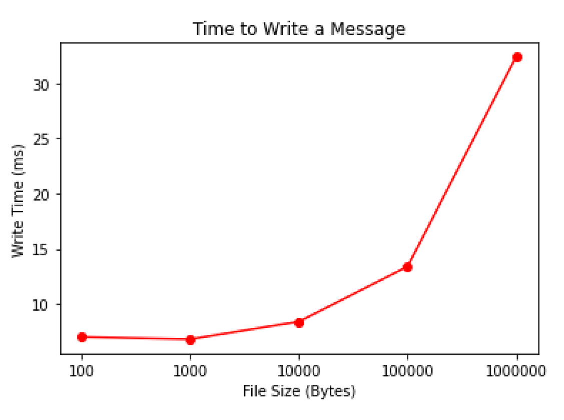
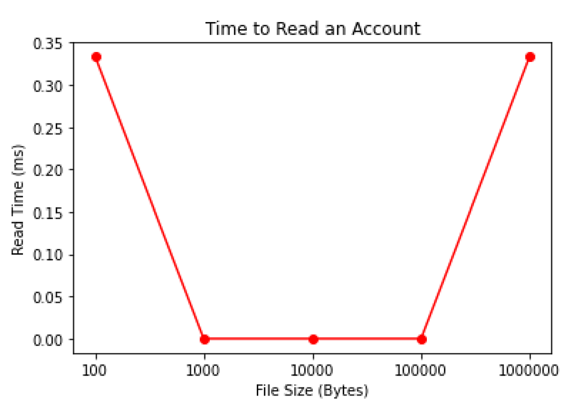
Old Storage System Performance

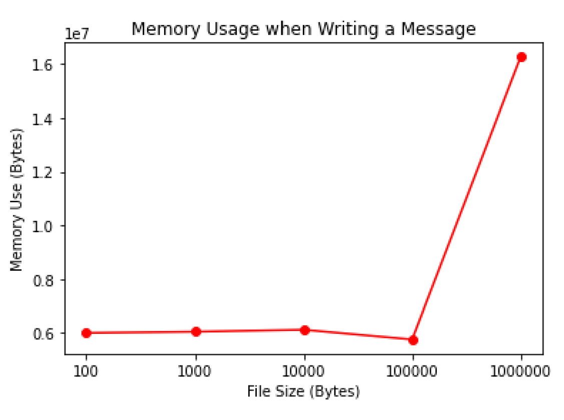
Read / Write Times

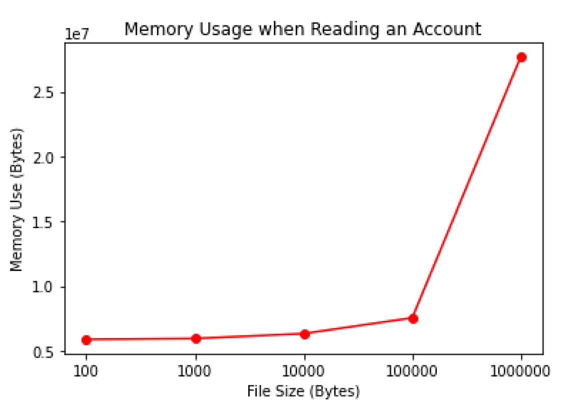




- File size is not completely exact, as the sizes of messages and accounts may not fit exactly
- The time values for messages are the average of 5 values
- The time values for accounts are the average of 3 values
  - The values for reading accounts are mostly too small for the inbuilt Javascript Date object which was used for testing so are recorded as 0

Memory Use





- The memory usage information for messages comes from a single test rather than the average from multiple so the plot may not be completely reliable
  - This is due to an issue with the testing program, which was not noticed until the tests had completed
- The memory usage information for accounts is the average from two tests, so should be more reliable

The current system becomes noticably slow when the files start to get reasonably large. This causes a noticable delay for the user when sending messages, and this problem will only get worse as the file gets larger.

The addition of file based messages has made this problem more apparent: a single reasonably large file (around a few megabytes in size) sent by a user can permanently slow down the entire application.

A new storage system is therefore needed.

Objectives of New System

- Access time should not increase as file size increases
  - This is the most important objective, as it is the only way to ensure that the application is scalable
- Supports concurrent access without any data corruption
- Must be able to quickly search for:
  - Accounts
    - By Username
  - Messages
    - By sender
    - By datetime
    - Should be able to return all messages between two datetimes

The most important characteristics of the new system will be speed and low CPU and memory use. Storage space can therefore be sacrificed in favour of these aims (i.e. it is acceptable for the new system to be far less efficient with space, in return for being faster and more efficient with other resources).

- However, steps should be taken where possible to be efficient with disk space as well

The main issue with the existing system is that it loads the entire file into memory. When the amount of items stored is very small, this allows for very quick read access. However, as the amount of data grows larger this very quickly starts to use an enormous amount of memory. Whenever an item is changed or added, the current system modifies the in-memory version of the data and then rewrites the entire file with the data. This is likely the reason that the speed of writing messages degrades so much as the file size increases.

To improve upon the existing system then, a new system will have to be able to read and write the data to/from the file without loading the entire thing into memory. However, this is will mean that it is unable to benefit from the $O(1)$ access time offered by the current use of hash maps. It is therefore important that the data is stored in a way that can be searched very quickly. Regardless of how well this is done, it is unrealistic to expect it to compete with hashmaps so it is likely that the new system will be slower than the existing one for small file sizes. The benefit of the new system will be that the search times remain suitably low even as file size increases.

How the new system works

Blocks

As the file is no longer loaded into memory, the new system cannot use hash maps for accessing the data. When reading it therefore has to search the file to find what it is looking for. To prevent search time increasing as the number of items in storage grows, messages and log entries are broken into separate files called **blocks**. All new items are added to the currently unfinshed block until the block reaches **1000** items, at which point the block is marked as finished and a new block is created.

When searching for items, rather than searching every item, the system only has to search the blocks whose largest and smallest timestamps match what it is looking for.

Format

Headers

To make searching a block quicker, the first 25 bytes of a block contain the following "headers":

- Block is full?
  - 1 byte
  - Any value other than 0 indicates that the block is full
    - This tells the `blockAccess.addEntry` method that this block is full and so it needs to create another one before adding the new entry
  - The `addEntry` method sets this to 1 once it has written the 1000th item to a block
- Number of entries in block
  - 8 bytes
  - Used by `addEntry` to know when 1000 items have been reached
- Next free space
  - 8 bytes
  - The position (specified in bytes) of the next empty space in the file (the position right after the end of the last entry)
  - Used by `addEntry` to skip straight to the end of the file when writing new items, rather than first needing to search for the end
- Middle entry position
  - 8 bytes
  - The position of the start of the middle entry in the file
  - Used by `getEntries` so that it only has to search half of the file
    - Jumps to the middle- if the items it is looking for have timestamps less than the middle entry then it only has to search first half, if greater it only has to search second half

Entries

Each entry contains the following fields:

Entry length (8 bytes)	Timestamp (8 bytes)	Data (variable length)

- The entry length field is needed during searching to skip to the next item, as each item is variable length The data field is simply written and read by $blockAess$, but it is created and processed by either $messa \geq sAess$ or $\log sAess$ and its contents are different for messages and logs. Logs format:

Log message (variable length)

Messages format:

Sender username (32 bytes)	Message type (1 byte)	Content length (8 bytes)	Content(variable length)	Filename length (8 bytes)	Filename (variable length)

The message type is converted to an 8 bit integer before being stored:

- Text = 1
- File = 2
- Image = 3

Searching

Blocks are searched mainly using linear search, however the middle entry header allows the system to only need to search up to half the file

- Although this is only true if range of timestamps being searched for does not contain the middle entry
- As disk reads are very slow 64kb of the file is loaded at a time to allow much faster in-memory searching
  - When testing, loading 64kb took the same amount of time as loading a single entry

Deleting

- The `blockAccess.wipeEntries` method "deletes" all entries between two given timestamps (inclusive) in a given block
- **Entries can't really be deleted**, as otherwise all following entries would need to be moved backwards to fill the space
- Instead, the `blockAccess.wipeEntries` method sets the timestamps of the entries to be "deleted" to -1 (so they won't be returned by any searches), leaves their length fields intact (as they are still needed to skip over them), and overwrites the rest of the entries with 0s.
- This means that deleting entries cannot be used to reduce space, but can be used to remove sensitive data

Indexes

The index files are used during searching to find out which block contains the item we are looking for.

Format

Headers

To make searching quicker, index files contain three 8 byte "headers" containing information about the index:

- Index length
  - The number of entries in the index
- Smallest timestamp
  - The smallest timestamp of any entry in the index
  - Will be the smallest timestamp of the first entry
- Largest timestamp
  - The largest timestamp of any entry in the index
  - Will be the largest timestamp of the last entry
  - Used so we can do basic checking before searching without needing to jump to the end of the file to get the largest timestamp of the last item
    - Before searching we can check that the smallest timestamp being searched for is not greater than the largest timestamp in the index
      - If it is, we know the file doesn't contain any matching entries so there is no point searching

Entries

Each entry contains three fields, each 8 bytes:

- Smallest timestamp
  - The smallest timestamp in the referenced block
    - Will be the timestamp of the first entry in the block
- Largest timestamp
  - The largest timestamp in the referenced block
    - Will be the timestamp of the last entry in the block
- Block number
  - The number of the block referenced by the entry
- Entry format is as follows:

Smallest timestamp (8 bytes)	Largest timestamp (8 bytes)	Block number (8 bytes)

Overall Format

- To save space, **the index files are written as raw binary**
- For the same reason there are also no separators or control characters
  - Entries are separated purely by their positions
    - e.g. if we want to read entry 10, we know that each entry is 24 bytes and that there are 24 bytes worth of headers at the start of the file. So we can calculate that the entry 10 will begin at byte $(24 + (24 * 10))$
- The index is formatted as follows:

Index length header (8 bytes)	Smallest timestamp header (8 bytes)	Largest timestamp header (8 bytes)	Entry 1 (24 bytes)	Entry 2 (24 bytes)	...	Entry n

Searching

The `indexAccess.getBlocks` method takes a start timestamp (`startTime`) and an end timestamp (`endTime`) and searches the index for all blocks which could contain entries with timestamps between `startTime` and `endTime`. For example, the method can be used to help get all messages from between two dates.

It works using binary search:

- The headers are used to find the middle entry in the file
  - We then check the middle entry
  - If `startTime` is less than the entry's smallest timestamp then we know the correct entries (if they exist) must be on the left hand side of this entry, so we repeat but using the middle entry between this one and the first one
  - If `startTime` is greater than the entry's largest timestamp then we know the correct entries (if they exist) must be on the right hand side, so we repeat but using the middle entry between this one and the last one
  - If `startTime` is greater than the entry's smallest timestamp but less than its largest one, then we have found one of the correct entries
    - We then do a simple linear search of all following entries until we reach one whose smallest timestamp is too big, meaning we have found all entries in the range we are looking for

As disk reads are very slow compared to memory access, `getBlocks` tries to avoid reading the disk when checking each entry by reading in many entries at a time.

- For the binary search part this is currently 2730 entries (the largest number of whole entries that fit into 64kb (65536 bytes))
  - When the index is large, it will take many iterations to get any benefit from this as the middle entries will be very far apart at first
  - In testing, reading 64kb seemed to take the same amount of time as reading a single entry
- For the linear search part this is currently 170 entries (the largest whole number of entries that can fit into 4096 bytes)
  - 4096 bytes is the default cluster size on most modern operating systems, meaning that we cannot read less than 4096 bytes at a time (if we try, it just discards the ones we didn't ask for)

Adding

The `indexAccess.addBlock` method adds a new entry to the end of the index file

Trees

As users are searched by username rather than timestamp, blocks and indeces are not an effective way to store them. Instead, a binary search tree is used. This allows the users to be stored in a (partially) ordered way without needing reorganise the file everytime a new user is added.

Format

Headers

- Length (8 bytes)
  - The number of nodes in the file (including ones that have been deleted)
  - Used to skip to the end when we need to add a new node
- Next free space (8 bytes)
  - Contains the position in the file of the next free space that a new node can be written to
  - Free space is created when a node is deleted
  - Keeping track of free space left by deleted nodes allows new nodes to be added in space previously occupied by deleted ones
    - Helping to keep the file size smaller
      - The first 8 bytes of each empty space will contain a pointer to the next free space if there is one (or 0 if there is not)
      - Contains 0 if there is no free space (meaning new nodes have to be added to the end of the file, increasing file size)
- Root pointer (8 bytes)
  - Contains the position in the file of the root node

Entries

Each user is a node in the tree, and has up to two child nodes:

Username (32 bytes)	Username value (8 bytes)	Left child node pointer (8 bytes)	Right child node pointer (8 bytes)	First name (32 bytes)	Last name (32 bytes)	Password (60 bytes)	Profile picture location (8 bytes)

The username value is the sum of all unicode characters in the username, and is used to order the items

- Any usernames, first names, or last names less than 32 bytes are padded with the unicode padding character (code 128)
  - However this is not included when calculating the username value The profile picture location is the position within the profile pictures file that the picture for this user can be found

Searching

- Start from the root node
- Is this node the user we are looking for?
  - If yes then stop
  - If no:
    - If the username value that we are looking for is less than or equal to this node's username value, then jump to the node pointed to by the left child pointer and repeat
    - If it is greater then do the same but with the right child node
    - If the child node pointer is empty, then the node we are looking for isn't in the tree

Adding

- First write the node to the file
  - If the "next free" header is 0, then append the node to the end of the file
  - Otherwise, there is some free space where a previous node was deleted
    - Go to the location specified by the "next free" header
    - Copy the first 8 bytes, as they will contain the location of the next free space (or 0 if there is no more), and write them to the "next free" header
    - Write the node starting at the location
- Then find its parent
  - Same procedure as searching, except when we find the node without a child pointer we place the address of the new node in the pointer
  - If it does not have a parent, then this must be the first node and is therefore the root so we record its location in the "root" header

This way of adding does not create a fully ordered tree, as the position of the node in the tree is not entirely decided by username value, but also by the time that it was added However, as we are simply using pointers we could create a fully ordered tree (as we would just have to adjust a few pointers when adding a new item, we wouldn't have to physically move any data)

- While all things being equal this would make searching much faster, in reality the partially ordered system is likely faster (though this has not put to the test)
  - This is because disk reads are **much** slower than in-memory access
    - So the system uses a buffer to read in 64kb from the file at a time and then searches that
  - If the tree was fully ordered, there would be no connection between position in the tree and physical position in the file (as tree position would purely be decided by username value)
    - We would therefore be frequently jumping to completely different parts of the file when moving from a parent to its child, so the buffer wouldn't be of much use (at least once the files get quite large)
  - The partially ordered system however means that it is likely that similar layers of the tree are in similar parts of the file, so we will likely have to refill the buffer far less often

Deleting

- First find the node to be deleted (this is done in the same way as searching normally)
- Overwrite the node with 0s
- Copy the "next free" header into the first 8 bytes of where the node was
- Overwrite the "next free" header with the position of the node that was just deleted