A BLOCK-LEVEL-TRACKING VERSIONING FILE SYSTEM (BLT)

Manushaqe Muco
manjola@mit.edu
Jacob Strauss, R12
04/13/2013

1. Introduction

A file system which maintains several versions of a file would provide users with a record of the history of changes made to the file. This would allow for recovery from crashes and user mistakes. The Block-Level-Tracking Versioning File System (BLT), designed as an extension to the UNIX file system, satisfies these requirements in a storage-and-time-efficient manner.

BLT allows the user to create new versions of a file. Each version has an inode structure pointing to the data blocks containing corresponding data for that version. BLT creates new copies only for the blocks that get modified, thus optimizing storage. Because of this implementation, inodes of different versions share common blocks.

Reading from the versions of a file is constant, because the data can directly be read off the blocks the inode points to. However, a storage efficiency problem is introduced when writing a small amount of data at the beginning of a large file. Because all the blocks of the file will change, new copies for each block are required. To solve this problem a design using *diffs* (content differences) could be used. But, that would result in slower performance when retrieving a version, because a single version would be built combining a large number of *diffs*.

2. Design

2.1 Versioning

Each version of a file *a.txt* has the format *a#n.txt*, where # is a special character and n denotes the version number. The oldest version has a version number of 1, and the version number increases with each new version. The user is not allowed to create any file with a name format matching that of the versions. Versions have read-only access and reside in the same directory as the file. Because directory versioning is rarely needed and to avoid an increase in design complexity, BLT does not version directories.

2.2 I/O Calls

To modify an existing file, an open() call is issued. In BLT, modifications include reads, writes and version-creation. After the desired modifications are done, a close() call is issued. Because BLT assumes no failures, an open() call is always followed by a close() call.

2.3 Design Description

To create a new version of the file *a.txt*, an open() call is issued. A copy for each block of *a.txt* is created on disk. As a result, two copies of the same set of blocks become present on disk. A new version file is then created. Its inode points to the original set of blocks, while the inode of *a.txt* now points to the copied set. For each write() call to *a.txt*, in addition to writing, the indices of the blocks being written to are stored in memory. When the close() call is issued, the blocks *a.txt* points to are kept if they were written to, and freed otherwise. The inode pointers for the freed blocks are replaced by the pointers to the corresponding original unwritten blocks. The inode of the previous version also points to these blocks.

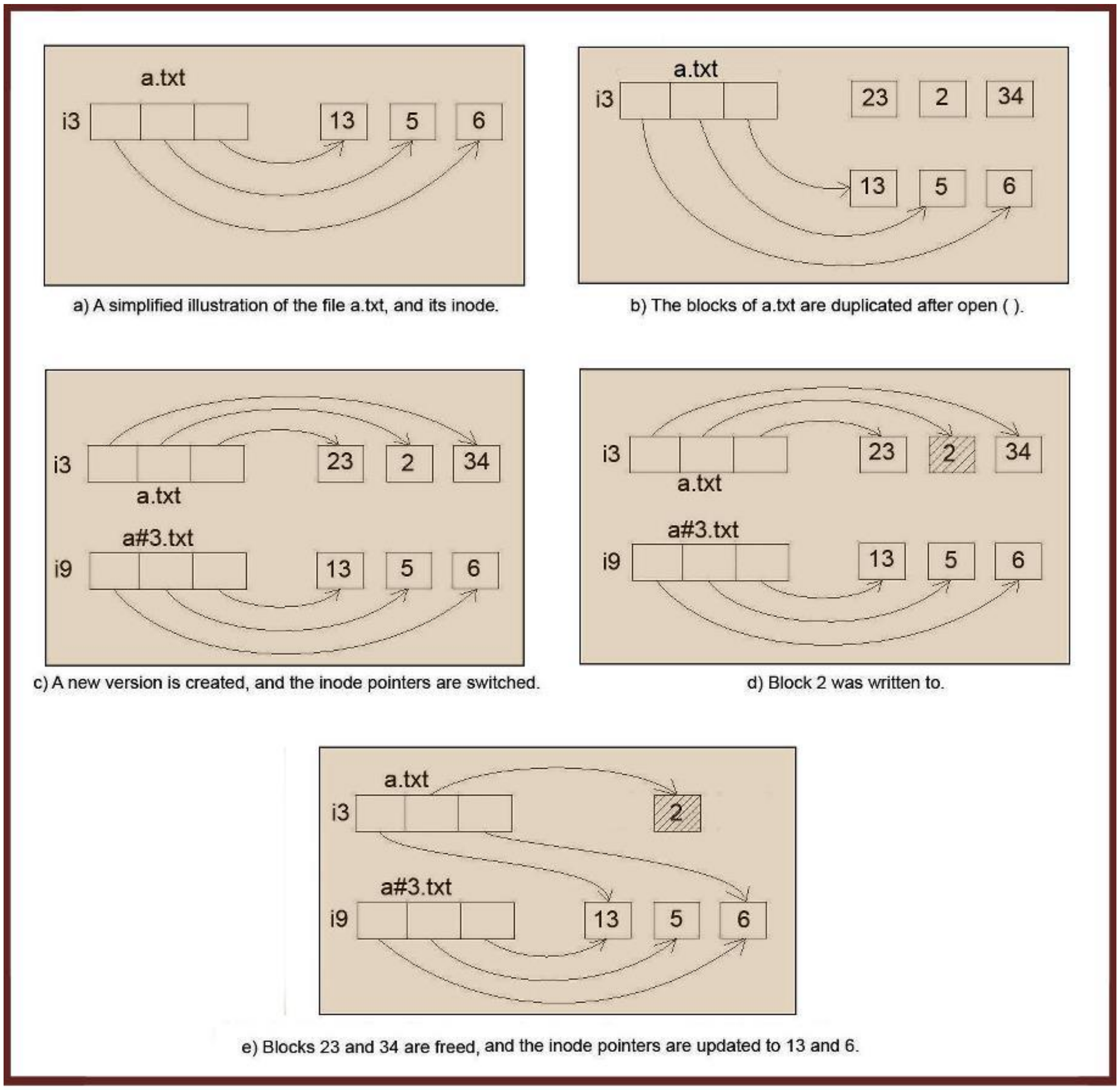
a.txt
i3
13
5
6
a) A simplified illustration of the file a.txt, and its inode.
a.txt
i3
23
2
34
13
5
6
b) The blocks of a.txt are duplicated after open ( ).
i3
23
2
34
a.txt
a#3.txt
i9
13
5
6
c) A new version is created, and the inode pointers are switched.
i3
23
2
34
a.txt
a#3.txt
i9
13
5
6
d) Block 2 was written to.
a.txt
i3
2
a#3.txt
i9
13
5
6
e) Blocks 23 and 34 are freed, and the inode pointers are updated to 13 and 6.

Figure: Creation of a version

The figure illustrates the creation of version *a#3.txt* of the file *a.txt*. The squares represent data blocks pointed to by the corresponding inode of the file or version. Numbers on the block represent the block number. Before versioning, *a.txt* is composed by the blocks 23, 2 and 34. Block 2 got written to (denoted by the shaded lines). Block 23 and 34 are freed and the pointers for the inode of *a.txt* are updated.

2.4 Supporting Data Structures

BLT introduces supplementary on-disk and in-memory data structures to support its design.

BLT introduces two new data structures **on disk**:

(1) *no_versioning_array,* an array storing the pathnames of the files whose versioning has been disabled by the user.

(2) *Information file,* created when a file is versioned for the first time. The information file for *a.txt* is denoted by *.a.txt.info*. It resides in the same directory as *a.txt* and it's hidden to the user (denoted by the "." prefix). It contains the following information:

- *recent_version_number:* an entry whose value is the version number of the most recently created version plus 1. This entry is used to decide the name of the next version to be created. For example, if recent_version_number=10, the next version to be created will be named *a#10.txt*.
- *inode_blocks_map:* a table whose entries store a mapping between an inode of a version and the block numbers it points to.
- *block_inodes_map:* a table whose entries store a mapping between a block and the number of inodes pointing to it. This table is used during garbage collection.
- *link_table:* a table which keeps track of the hard-links to the file. It has two columns: *parent* and *child*. If *a.txt* is hard-linked to *b.text*, *a.text* is stored as a child and *b.txt* is stored as a parent.

BLT introduces a new data structure **in memory**:

(1) *mem_file_info,* a table created between an open() and close() call, to facilitate version creation. Each row of the table corresponds to one file and contains the following information about that file:

- *path_file:* the path of the file that's currently opened.
- *file_descriptor:* an indicator for accessing the file.
- *apps_num:* the number of applications the file is currently opened in.
- *indices_of_changed_blocks:* an array of the indices of the file blocks that get written to.

2.5 Standard UNIX Operations

BLT does not modify **read, symlink, mkdir, chdir,** and **stat.** It does, however, modify the following operations as described:

- **Write**: In addition to writing, the indices of the blocks that get written to are stored in memory.
- **Rename**: When a file is renamed, its versions are renamed as well. For example, renaming *a.txt* to *b.txt*, will rename the n-th version *a#n.txt* to *b#n.txt*.
- **Link:** If *a.txt* is hard-linked to *b.txt,* its versions are hard-linked to the versions of *b.txt* as well. That is, *a#n.txt* is hard-linked to *b#n.txt*. The information file for *a.txt* is also hard-linked to the information file for *b.txt*. The link_table records the hard-link between *a.txt* and *b.txt* by adding a row in the table with *b.txt* as parent and *a.txt* as child.
- **Unlink:** It erases all the links and updates the link_table.
- **Open()/Create():** Several new steps are introduced during versioning. The blocks of the file are duplicated. A new version file is created using the recent_version_number. The version inode is pointed to the original blocks and the file inode is pointed to the copied blocks. The

recent_version_number, block_inodes_map and inode_blocks_map of the information file are updated and the apps_num of the mem_file_info is increased. BLT implements these steps through a new system call: *initiate(pathname)*. This new call is issued at the beginning of the open() call, under the condition that the file given by the pathname is not opened in other applications and its versioning is not disabled.

- **Close():** There are two cases to consider during a close() call.

  (1) The user has not written to any blocks. In this case, the version created by initiate() is deleted.

  (2) The user has written to some blocks. In this case, the unchanged blocks will be freed and the inode pointers will be updated to the pointers corresponding to the previous unchanged blocks.

  In both cases, the information table will be updated and the row of mem_file_info corresponding to the file that was versioned will be removed. BLT implements these steps through a new system call: *commit(file)*. This new call is issued before the close() call, under the condition that the file is not opened in other applications.

2.6 Garbage Collection

Garbage collection is performed every 36 hours, based on the number of versions. If the number of versions for a file exceeds a *maximum_version_number*, versions are deleted starting with the oldest ones. So as not to delete all the versions, when the number of versions becomes equal to a *minimum_version_number*, no more versions are deleted. The maximum_version_number and minimum_version_number are parameters already built in the system, and they cannot be changed by the user.

BLT performs garbage collection through a system call: *garbage_collect(pathname, v_num)*. This system call deletes the version with version number v_num of the file specified by pathname. To delete the version, it first empties the block pointers for the version inode and deletes both the inode and the inode entry in the inode_blocks_map table. The block_inodes_map entries are also updated, by decreasing the number of inodes for each block of the version by 1. If the number of inodes pointing to the block becomes zero, the block is freed.

2.7 Use Cases

- **Writing few blocks of a large file.** Because BLT keeps only the blocks that were written to and frees the rest, this case is storage efficient.

- **Appending to a log file** is the simplest case of writing to a file, because only the last block of the file gets modified.

- **Creating a new file in a large directory** creates a hidden information file in that directory for the corresponding file.

- **Allowing users to exclude specific directories from versioning.** BLT does not version directories.

- **Allowing users to exclude files from versioning.** BLT allows the users to disable versioning for files through the API call: *disable_versioning(pathname),* where pathname is the name of the path of the file to be disabled. This API call adds the pathname to the no_versioning_array. Equivalently, BLT introduces the API call *enable_versioning(pathname),* which re-enables the versioning of the file given by the pathname. The pathname is removed from the no_versioning_array.

- **Finding some string across all versions of a file.** BLT allows the user to do so, through the API command *search_string(string,pathname)*. This API command calls on the system call *search(string,pathname)*. The algorithm used for search() is the following:

  ➔ Let versions_blocks be the array of block numbers currently used through all versions.
  ➔ Let version_map be a hashmap, with:
  (key,value)=(block of versions_blocks, array of inodes pointing to the block).
  ➔ Search string through each block of versions_blocks.
  ➔ Store blocks containting string in an array has_string_blocks.
  ➔ From version_map, determine the inodes pointing to the blocks that have the string.
  ➔ Return the versions corresponding to these inodes.

- **Finding some string across all versions in the file system.** The API command search_string() can be recursively run from the root of the file system.

3. Analysis

The system is analyzed for both storage and time efficiency. Storage is analyzed in terms of bytes and blocks, depending on the size of the file. Block size is assumed to be 512 bytes. Time is analyzed in terms of disk read/writes and memory read/writes. Because CPU time is negligible, it is not considered in analysis.

3.1 Repeatedly writing to a small file

The size of a small file is small enough to be contained in one block, say 40 bytes. The writes done to the small file are of the size of bytes. Even if 100 writes are performed, the size of the file will not increase beyond 512 bytes. That is, the writes are all done on the same block.

- **Storage**: BLT creates a version between an open() and close() call. There can be as many writes as needed in between. For these 100 writes, in total, one extra block will be created to get written to, and one extra inode will be needed for the new version. BLT is storage-efficient in this case.

- **Time**: UNIX handles repeated writes to a block by caching the block in memory. The needed writes will be done from the cache. For example, the first four writes will be on disk, while the rest will be in memory. Memory writes are much faster. Because BLT does not change this UNIX implementation, it will be time-efficient for this case.

3.2 Repeatedly writing a block of a large file

A large file spans over many blocks, say 4096 blocks for a file of size 2 MB. Assume 100 writes are repeatedly done on one block of this file.

- **Storage:** On open(), the file blocks are duplicated. This needs extra storage for 4096 blocks. On close(), only the block that got written to (100 times) is kept; the rest are freed. In the end, storage for only one extra block is needed. However, there might be an issue when many large files are opened at once, or when the file is large enough to create a storage issue during block duplication.

- **Time:** Because UNIX uses caching, this case is time-efficient as well.

3.3 Searching through all versions of a small file

The search algorithm searches through the blocks of all versions and returns the versions containing the string.

- **Storage:** Because the algorithm doesn't create on disk data structures, there is no need for extra storage.

- **Time:** The algorithm will be analyzed mainly for time efficiency, by going through its main steps:
  - Two data structures are created in memory: versions_blocks and version_map. The information for these data structures is obtained by the information file, through one disk read.
  - The string is searched by on-disk reading through blocks, and no two blocks are read twice. The efficiency of this step depends on the file size. If the file size is less than a block, there is no improvement compared to the UNIX-grep, because as many blocks as there are versions will be searched (each version is a separate block). If the file spans over many blocks and if its versions share many blocks, the improvement will be great, because a much smaller amount of blocks will be read from the disk.
  - Returning the versions that contain the string can be done through memory reads using the hashmap version_map. There will be as many memory reads as versions to be returned.

  Overall, this algorithm is time-efficient.

3.4 Scalability

There are two scalability issues to consider. The first issue is versioning a considerate number of large files. Each version will require a big number of blocks. The problem becomes worse for small writes at the beginning of large files, because all the blocks will change. The second issue is doing only small modifications to a file. Although the versions are not much different from the file, a new inode will be required for each version.

4. Conclusion

BLT is a storage-and-time-efficient design which allows file versioning. Storage is optimized by keeping only the modified blocks and freeing the rest. Time is optimized by providing fast access to versions through their inodes. However, further modifications can be done for storage efficiency, especially in the case of writing a small amount of data at the beginning of a large file.

5. Acknowledgment

1. Jorge Simosa

2. Various office hours.

Word count= 2430