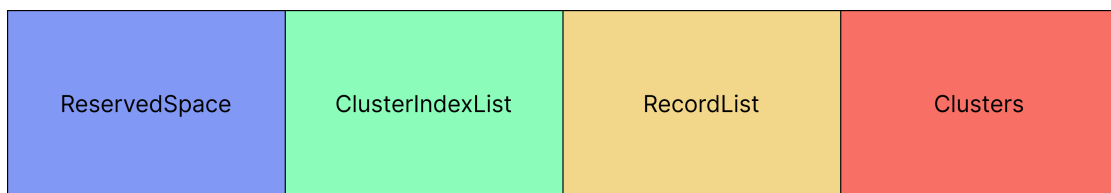


David Utyuganov's File System

December 31, 2022

DUFS Architecture Overview



General architecture of DUFS contains 4 blocks. First 3 of them are static and generated in the process of volume's mount, while the last used as storage of the content.

Both directory and file structures are linear lists and this is the key disadvantage of the filesystem at this moment. It could be remade on b-trees to significantly reduce disk operations (logarithmic complexity instead of linear).

Currently it does not support logging or any other features for safe use. If there was any defects during work of filesystem, most probably, the volume will become corrupted and some part of it's data could be lost forever.

ReservedSpace

ReservedSpace is a block that contains specific service information for fast and correct work of the filesystem. Total size of this block is 60 bytes and the fields are the following:

ReservedSpace		
Field	Bytes	Description
DUFSSignature	4	Constant value equal to 0x44554653 (DUFS). Used as key-signature to detect if the file is DUFS volume
VolumeName	16	8 UTF-16 symbols
ClusterSize	4	Cluster size value. Must be divisible by 4
VolumeSize	8	Brutto size of the volume
ReservedClusters	4	Number of netto reserved clusters
CreateDate	2	Date of volume creation
CreateTime	2	Time of volume creation
LastDefragmentationDate	2	Date of volume's last defragmentation
LastDefragmentationTime	2	Time of volume's last defragmentation
NextClusterIndex	4	Index of next free cluster index
FreeClusters	4	Number of free clusters in the volume
NextRecordIndex	4	Index of next free record index
DUFSTailSignature	4	Constant value equal to 0x4A455432 (JETB). Same as DUFSSignature

ClusterIndexList

ClusterIndexList is a linear list that keeps information about relationships between clusters. It is similar to FAT.

Each element of this list is ClusterIndexElement that takes 12 bytes.

Length of ClusterIndexList is equal to the number of reserved clusters and it cannot be greater than $2^{32} - 1$. This limit could be increased up to $2^{64} - 1$.

ClusterIndexElement structure:

ClusterIndexElement		
Field	Bytes	Description
NextClusterIndex	4	Index of next cluster in the chain
PrevClusterIndex	4	Index of previous cluster in the chain. Uses to perform fast defragmentation
RecordIndex	4	Index of record that contains this cluster. Uses to perform fast removal of record from parent directory's cluster

RecordList

RecordList is a linear list that keeps information about records in the volume.

Record is an entity to generalize files and directories in one term. It is useful because files and directories mostly contains topologically same type of meta data.

One record takes 93 bytes of data.

Length of RecordList is equal to the number of reserved clusters.

Each element of RecordList is a Record:

Record		
Field	Bytes	Description
Name	64	32 UTF-16 symbols
CreateDate	2	Date of record creation
CreateTime	2	Time of record creation
FirstClusterIndex	4	Index of first cluster in the chain
LastEditDate	2	Date of last edition of the record
LastEditTime	2	Time of last edition of the record
Size	8	Size of record in bytes
ParentDirectoryIndex	4	Index of record of parent directory
ParentDirectoryIndexOrderNumber	4	Order number of this record in parent directory's cluster. Used to find/delete/remove records from parent directory's cluster by $O(1)$
IsFile	1	Byte that says if this record is file or directory

Clusters

This block uses as storage of the content for files and as a storage of nesting records for directories.

Comparison With Other File Systems

File system	Max. record name	Max. file size	Max. number of files	Bake
NTFS	255	16TiB	$2^{32} - 1$	Yes
FAT32	8.3	4GiB	?	No
ext4	255	16TB	$2^{32} - 1$	No
DUFS	32	16TiB (?)	$2^{32} - 1$	Yes

Performance And Scalability Analysis

RAM

During filesystem usage RAM is always very low. Only 60 bytes of addition data keeps in the RAM all the time, everything else is imperceptible. Read/write/append uses bytes in RAM equal to cluster size through buffered read/write.

CPU

During filesystem usage CPU load is also imperceptible. Read/write/append overhead growth linearly to the length of the content. Find overhead growth linearly to the number of records in the directory. Low-level overhead (CPU L-cache/registers) growth probably linearly to the number of records. Defragmentation overhead depends on the quality of the initial approximation – if volume is almost defragmented, the overhead of the defragmentation on top of it will be very low.

DISK

DISK complexity is arguable. Generally it is ok from the algorithmic point of view (besides linear search complexity in cause of linear structure of record list), but it could be optimized from the software point of view to reduce the "constant" complexity (memorization/caching of the results of DISK read operations). It could be significantly reduced if linear structure of record list will be replaced with the b-tree structure. It could reduce algorithmic complexity from linear to logarithmical.

Scalability

DUFS is opened to select any cluster size that is divisible by 4 and any volume size that is less than 16EiB (2^{64} bytes), but the number of clusters = $\left\lceil \frac{\text{volume size}}{\text{cluster size}} \right\rceil$ must be less than 2^{32} . This value theoretically could be increased up to 2^{64} .

DUPS API

DUPS is implemented as java class with the following public interface:

```

1 public class Dufs {
2     public RandomAccessFile getVolume();
3     public void mountVolume(String path, int clusterSize, long nettoVolumeSize);
4     public void attachVolume(String path);
5     public void closeVolume();
6     public void createRecord(String path, String name, byte isFile);
7     public void writeFile(String path, File file);
8     public void appendFile(String path, File file);
9     public void readFile(String path, File file);
10    public void deleteRecord(String path, byte isFile);
11    public void renameRecord(String path, String newName, byte isFile);
12    public void moveRecord(String path, String newPath, byte isFile);
13    public void printDirectoryContent(String path);
14    public void printVolumeInfo();
15    public void printVolumeRecords();
16    public void printDirectoryTree();
17    public void defragmentation();
18    public void bake();
19    public void unbake();
20 }

```

This class simulates file system on top of volume that could be mounted or attached through methods. It supports basic I/O operations like read/write/append, move/delete operations, rename, defragmentation, bake/unbake and several printing methods to represent nested data via console. DUPS volume itself is implemented on top of `java.io.RandomAccessFile`, I/O operations are implemented on top of `java.io.File`.

void	mountVolume(String path, int clusterSize, long nettoVolumeSize)
Mounts (creates) a volume by the following path with the following clusterSize and the following nettoVolumeSize. Size of volume will be bigger after the creation of file since nettoVolumeSize is the size that indendent only for clusters.	
Throws DufsException if such volume already exists by this path; Throws DufsException if given volume name length is beyond 8 symbols; Throws DufsException if given volume name length contains prohibited symbols; Throws DufsException if given nettoVolumeSize is greater than $1.1 \cdot 10^{12}$ (1 TiB or 1024 GiB); Throws DufsException if there is not enough space on disk to mount the volume; Throws DufsException if clusterSize is not divisible by 4.	

void	attachVolume(String path)
Attaches volume by the following path.	
Throws DufsException if there is no volume with such name by this path; Throws DufsException if volume signatures doesn't match with the expected values.	

void	closeVolume()
Closes the volume.	
Throws DufsException if volume is null (was not mounted or attached before this operation).	

void	createRecord(String path, String name, byte isFile)
Creates the record with the following name by the following path in the volume. Record could be either file (isFile == 1), either directory (isFile == 0).	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>); Throws DufsException if given record name length is beyond 32 symbols; Throws DufsException if given record contains prohibited symbols; Throws DufsException if record with such name and type already contains in the path ; Throws DufsException if there is not enough space on the volume to create new record.	

void	writeFile(String path, File file)
Writes the content from file into the file (<i>record</i>) in the volume by the following path . Overwrites any content that is already contains in the file (<i>record</i>)	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>); Throws DufsException if there is not enough space on the volume to write given content; Throws DufsException if file (<i>record</i>) does not exist.	

void	appendFile(String path, File file)
Appends the content from file to the file (<i>record</i>) in the volume by the following path .	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>); Throws DufsException if there is not enough space on the volume to append given content; Throws DufsException if file (<i>record</i>) does not exist.	

void	readFile(String path, File file)
Reads the content from file (<i>record</i>) in the volume by the following path and writes it into the file.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>); Throws DufsException if file (<i>record</i>) does not exist.	

void	deleteRecord(String path, byte isFile)
Deletes record by the following path from the volume.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>); Throws DufsException if isFile == 0 and the directory (<i>record</i>) is not empty; Throws DufsException if file (<i>record</i>) does not exist.	

void	renameRecord(String path, String newName, byte isFile)
Sets record's name by the following path in the volume as newName .	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>); Throws DufsException if given newName length is beyond 32 symbols; Throws DufsException if given newName contains prohibited symbols; Throws DufsException if record with newName and type already contains in the path ; Throws DufsException if record by the following path does not exist.	

void	moveRecord(String path, String newPath, byte isFile)
Moves record by the following path into the newPath in the volume.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>); Throws DufsException if record with such name and type already contains in the newPath ; Throws DufsException if record does not exist.	

void	printDirectoryContent(String path)
Prints records that contains by the following path in the volume. Used for debug purposes, so does not catch any errors. Unsafe to use.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>).	

void	printVolumeInfo()
Prints volume info, such as date/time of creation, reserved/free size etc..	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>).	

void	printVolumeRecords()
Prints all records that contains in the volume. Used for debug purposes, so the data is specific, but it prints full cluster chain for each record.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>).	

void	printDirectoryTree()
Prints volume's directory tree.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>).	

void	defragmentation()
Performs defragmentation of the volume. Order of clusters is defined by the order of indexes in the record list.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>).	

void	bake()
Bakes the volume. calls defragmentation() inside and then truncates the volume to the minimal possible size without losing any data. After baking the volume becomes read-only. Non-read operations's behaviour on the baked volume is undefined.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>).	

void	unbake()
Unbakes the volume to the initial size. After unbaking the volume is ready to be used for any operation.	
Throws DufsException if volume is null (<i>was not mounted or attached before this operation</i>).	