Article

The Oxford Common File Layout: A Common Approach to Digital Preservation

Andrew Hankinson [1], Donald Brower [2], Neil Jefferies [1], Rosalyn Metz [3,*], Julian Morley [4], Simeon Warner [5] and Andrew Woods [6]

1 Bodleian Library, Oxford University, Oxford, OX2 0EW, UK; andrew.hankinson@bodleian.ox.ac.uk (A.H.); neil.jefferies@bodleian.ox.ac.uk (N.J.)
2 Hesburgh Libraries, University of Notre Dame, Notre Dame, IN 46556, USA; dbrower@nd.edu
3 Emory Libraries, Emory University, Atlanta, GA 30322, USA
4 Stanford University Libraries, Stanford University, Stanford, CA 94305, USA; jmorley@stanford.edu
5 Cornell University Library, Cornell University, Ithaca, NY 14850, USA; simeon.warner@cornell.edu
6 Fedora Commons, DuraSpace, Beaverton, OR 97008, USA; awoods@fedora-commons.org
* Correspondence: rmetz@emory.edu

Received: 1 March 2019; Accepted: 23 May 2019; Published: 4 June 2019

Abstract: The Oxford Common File Layout describes a shared approach to filesystem layouts for institutional and preservation repositories, providing recommendations for how digital repository systems should structure and store files on disk or in object stores. The authors represent institutions where digital preservation practices have been established and proven over time or where significant work has been done to flesh out digital preservation practices. A community of practitioners is surfacing and is assessing successful preservation approaches designed to address a spectrum of use cases. With this context as a background, the Oxford Common File Layout (OCFL) will be described as the culmination of over two decades of experience with existing standards and practices.

Keywords: digital preservation; digital repositories; digital objects; object stores; storage

---

1. Introduction

The Oxford Common File Layout (OCFL) initiative began as a discussion among digital repository practitioners about the ideal layout and characteristics for a repository's persisted objects. The need for a common layout grew out of a lack of commonality in the way objects are stored by repositories. It has since grown into an open community effort defining an application-independent way of storing versioned digital objects with a focus on long term digital preservation. The OCFL represents a specification of the community's collective recommendations, addressing five primary requirements: (1) Completeness, so that a repository can be rebuilt from the files it stores, (2) parsability, both by humans and machines, to ensure content can be understood in the absence of original software, (3) robustness against errors, corruption, and migration between storage technologies, (4) versioning, so repositories can update and make changes to objects that allow the history of the object to persist, and (5) the ability to store content on various storage infrastructures, including cloud object stores.

In March 2018, an Editorial Group was formed, made up of participants from Cornell, Stanford, DuraSpace, Oxford, and Emory. Over the next six months the Editorial Group worked with the community to identify use cases and scope the initial release of the specification. In September 2018, one year after the first discussions, the Editorial Group of the OCFL came together at Oxford University to begin writing the Oxford Common File Layout specification [1]. The editors attempted to decouple the structure of the persisted files from the software and storage infrastructure that might manage it. This document describes motivations, principles, and features of the alpha release of the specification.

2. Lessons from the Community

Over the last two decades, the repository community has struggled with the reality of supporting and maintaining repository software. Over time, software has changed significantly, either through normal upgrades and changes to the underlying technology, or because strategic decisions are made to choose a new repository solution. Either way, institutions struggle with how to make sure content outlasts the technology.

To date, there are only two common ways of structuring repository data. The first being "BagIt", which is intended for transport from one location to another. While one could consider storage transport over time, BagIt rightfully does not support versioning. The second is the Moab design for versioning objects, which lacks wide adoption.

Stanford, Notre Dame, and Emory have all struggled with a lack of a standard method for laying out objects in their repository. What follows describes the struggles each of the institutions have faced. Each of these institutions are keenly interested in the OCFL in the hopes that it can resolve some of the issues described herein.

2.1. Stanford University Libraries

In 2008, Stanford Libraries received funding from the Stanford University Provost for the building of a world-class digital library infrastructure. This work, known as the Digital Library Build-out (DLB), was a continuation of pioneering research on archival digital libraries performed by Stanford and its partners in the Archive Ingest and Handling Test (AIHT) [2] (2003–2005) and in the National Geospatial Digital Archive (NGDA) [3] project (2005–2008), both funded by the Library of Congress' National Digital Information Infrastructure and Preservation Program (NDIIPP). A core component of the DLB was the redesign of the Stanford Digital Repository (SDR) in 2009 [4]. The first iterations of the SDR stored preserved objects using the Library of Congress BagIt format, but after several years of experience it became clear that, while BagIt was appropriate as a transfer mechanism between repositories, it lacked several key features for managing the lifecycle of a digital object within a repository. This led to the development of Moab, an archive information package (AIP) [5] format designed to address the specific needs of a modern digital repository.

An Introduction to Moab

Moab, designed by Richard Anderson et al. in Stanford Libraries' Digital Library Software & Services team [6], is a versioned, forward-delta AIP format that allows for the efficient storage, management, and preservation of any digital content. It has been in production use by the SDR since 2012.

The selection of a forward-delta versioning system has been especially important as the SDR ingests more audio-visual content. Without forward-delta versioning, metadata updates to existing objects (a relatively common occurrence) would consume an outsized amount of storage for the actual amount of data on the disk changed. While there are filesystem-level solutions to content deduplication that are more efficient than the file-level deduplication offered by Moab (e.g., WAFL from NetApp [7] or ZFS from Oracle [8]), these almost always require vendor-specific storage solutions. Further, when the object is transferred between systems over the network, the entire fully-hydrated object must be transferred and the receiving system may not support deduplication, leading to excessive network and storage consumption.

A core component of Moab is the desire to retain human readability of preserved content and preservation metadata. This is solved by retaining the original file names of deposits and writing preservation metadata into human-parsable XML files. A lesson learned that the OCFL benefits from is the realization that many small XML files scattered across many version directories, while readable individually, are less useful for constructing a narrative of the object's history. The OCFL's decision to

consolidate all changes into a single JSON file both removes the somewhat-contradictory XML schemas in the various Moab manifests and provides a single file of record with an internally-consistent schema.

Moab was designed with the presumption that files would reside on a POSIX-compatible filesystem. This seemed like a reasonable assumption to make given the remarkable longevity of the POSIX specification. However, with the rise of object-addressable storage, especially with increased adoption of cloud-based storage, such as Amazon S3 and Openstack Swift, it is no longer reasonable to design a preservation system with the presumption that it will always have POSIX file semantics available for discovery and indexing.

Stanford tracks preserved data in Moab objects by assigning to each a unique identifier called a digital resource unique identifier (DRUID). This is a string composed of a subset of alphanumeric characters that conforms to a specific pattern, giving a namespace of 1.6 trillion possible digital objects. To enable efficient storage of these objects on a POSIX-based filesystem, Stanford also utilizes Druidtree, a modified version of Pairtree [9], to create a predictable directory hierarchy that efficiently disperses DRUIDs across a filesystem.

Given a Druid (e.g. `bb123cd4567`), it is possible to construct a Druidtree path to the expected location of the version 1 `manifestInventory.xml` file of the associated Moab object (e.g. `bb/123/cd/4567/bb123cd4567/v0001/manifests/manifestInventory.xml`) and, if found, construct a path to the signatureCatalog.xml file. By parsing that file, paths to all other files in that version and all prior versions of preserved content (but not of preservation metadata) may be constructed. However, it provides no knowledge of higher versions of that object. On a POSIX-based file system, these other versions can be discovered via a relatively inexpensive '`ls`' or '`dir`' action inside the parent directory of the Moab object, followed by some logic to enumerate and sort the results for directories that match the expected syntax of valid version directories.

While S3 and other object stores can emulate the enumeration of files in a directory by assuming certain characters in an object name can be treated as directory markers [10], it is important to note that they are actually filtered sorts of all objects in a particular bucket, and thus suffer potential performance issues as the total number of objects in the namespace increases. Enumerating all files of a Moab object using traditional POSIX semantics when the object resides on object-addressable storage is therefore relatively inefficient and, given the pricing models employed by commercial cloud object storage, incurs unnecessary additional costs.

The OCFL neatly addresses this issue by placing the most current version of the object's inventory at the root level of the object, creating an essentially immutable object path that not only provides an index of all files in the object, but also enumerates all versions of that object. With Stanford's repository approaching two million objects and 10 million object-versions, this design offers significant speed and cost savings when conducting audit operations (see Table 1). Further, by placing the most recent inventory file in the object root, the OCFL preserves Moab's concept of immutable version directories. This feature is designed to facilitate the storage of objects on write-once, read-many media, such as tape, without compromising the ability to version those objects in the future. In the unlikely event that the inventory file in the object root does not reflect the most current version on that object store, audit tools may fall back to an iterative approach of sequentially enumerating higher version directories until no more are found.

Table 1. Cost in filesystem calls to create index of all files in all versions of an object.

Moab	OCFL
Enumerate all version directories in object root Parse every version of manifestInventory.xml and most recent version of signartureCatalog.xml	Parse inventory.json in object root
Cost = 1V+2	**Cost = 1**

Another issue that only becomes apparent at scale is the cost and latency incurred by performing multiple small-block reads to construct the history of a Moab object at its current version. For a given version (V), four manifest files must be read to fully reconstruct the object. To fully parse an object as of the given version, 4V reads must be performed. These reads are almost always small, as most manifest files are <4 KB in size, with the total number of reads required to parse an object scaling linearly with versions.

At a small scale this cost is negligible. But as the number of objects in the repository grows, and the number of versions of those objects also grows, this incurs a significant overhead, which in turn decreases the ability of audit tools to keep an accurate current inventory of objects. In Moab, an object at version 5 requires 20 small block reads to fully reconstruct, and if the number of versions is not known, then a list/dir operation must be performed on the object root to discover all potential version directories. In the OCFL only one read is necessary, albeit of a file larger than 4 KB. Across one million objects with an average of three versions each, this reduces the cost of object discovery and construction from 13 million I/O operations to one million. Additionally, for objects stored in commercial cloud systems, this represents a significant reduction on the number of request operations that must be performed, and hence a reduction in the charges incurred by audit actions (see Table 2).

Table 2. Cost in filesystem calls to reconstruct full history of an object.

Moab	OCFL
Enumerate all version directories in object root Parse every version of versionInventory.xml, versionAdditions.xml, fileInventoryDifference.xml, and manifestInventory.xml	Parse inventory.json in object root
Cost = 4V+1	**Cost = 1**

Moab made the decision to compute checksums for every file in the object using three different digests—MD5, SHA1, and SHA256. This was an attempt to avoid short-term obsolescence and provide surety that a given file was unaltered, even if one algorithm was shown to be vulnerable to compromise. Within 10 years, two of the three digests have been broken (MD5 [11], SHA1 [12]), showing the need for the next version of Moab to have an easy way to add new digests over the life of the object. The OCFL achieves this. Further, experience with Moab has shown that there is no value in storing more than two checksums for a given file, and especially not MD5, which is only useful as an ephemeral checksum used to verify successful file transfers between systems

In conclusion, although Moab is a well-designed AIP with a proven track record, several inefficiencies in the design have been identified that pose scalability challenges, especially when using object-addressable storage, such as AWS S3. The OCFL is an evolution of Moab that retains its core features of forward-delta versioning, version immutability, and file-based deduplication, whilst providing for more efficient object discoverability and reconstruction in both POSIX and object-addressable storage environments and presenting a clearer path for adding new digest algorithms over the lifetime of an object.

2.2. *University of Notre Dame Hesburgh Libraries*

The University of Notre Dame is committed to digital preservation and in 2016 it added tools to the institutional repository to realize that commitment. Recognizing that all systems are eventually replaced, the decision was made to have the preservation system sit below a Fedora 3 repository to ensure the data could be accessed without the need of any preservation software. In the worst-case scenario, Notre Dame could create migration tools should the data need to migrate in the future. At the time, there was little prior work available on preservation filesystem layouts. Notre Dame's system was built on ideas developed by the California Digital Library, Pairtree, and DFlat [13]. The most influential approach, though, was the Moab design for digital object versioning.

When object storage was tested on the Notre Dame tape system, it was discovered that the system disliked having many small files to keep track of. Instead, Notre Dame chose to serialize each version of an item into an uncompressed Zip file in the BagIt [14] format. This reduced the number of small files stored and it also allowed for the verification of each BagIt file using standard BagIt tools.

While the system works well for Notre Dame's needs, the biggest concern is having an organization schema unique to the institution. Ideally, Notre Dame would utilize a shared community tool similarly designed for use by a variety of repository tools, eliminating the need to do a logical file migration and allowing them to simply point new software to an existing corpus of data.

2.3. Emory Libraries

Over the years, Emory Libraries have built a number of homegrown repositories, each of which filled the very specific and unique needs of the content and departments stewarded by the libraries. These silos resulted in distinct object structures, metadata formats, and an inability to connect content across the enterprise.

In January 2014, a Digital Repository Working Group was convened to gather information and develop documentation to choose a digital repository solution. The Working Group met for approximately 32 weeks, and during the course of their work they developed a list of content currently in the Libraries' various repositories; identified applications, platforms, and tools used to store digital assets; and conducted a preservation risk analysis for all known digital assets held by the libraries. At the end of the Working Group's charge the team settled on Samvera as the framework for their future repository.

In the Fall of 2016, Emory Libraries kicked off a requirements-gathering phase to help inform their forthcoming Samvera-based repository. Working with 90+ stakeholders across the libraries and the university, Emory's Digital Library Program created a number of task forces to help identify requirements around deposit, display, and preservation of digital objects. The task forces established for this work spent a considerable amount of time normalizing metadata and object models currently in use by the libraries' existing repositories.

In the course of gathering requirements, the task force assigned to digital preservation made some key decisions. First, the task force decided that the repository would be agnostic to the types of content the repository would preserve [15]. Next, the task force identified key actions [16] it would automate as part of its regular operations. These actions would form the basis of the various workflows that the repository would use to preserve content. Finally, the task force identified the requirements for Emory's Archival Information Package [17]. These recommendations intentionally identified what information should be stored (content files and metadata) but did not identify how it should be stored (i.e., how it should be laid out on disk).

As the repository has moved into the development phase, including the migration of initial collections, Emory Libraries has struggled with the data migration process. In particular, they have been confronted with two key issues. First, the libraries hope to store previous iterations of their digital objects. The new repository will have new metadata formats, new object models, and potentially new file formats. However, the libraries recognize that information from the previous repositories may be useful for informing future generations. Second, Emory Libraries would like to future-proof against technology changes. Over the years, the libraries have seen software around repositories change dramatically. Emory Libraries hopes its digital objects won't require significant future migrations but recognizes that it may be inevitable. Instead, the libraries hope that the new repository can store digital objects in an open, transparent, and standardized manner to lessen the impact of future migrations.

3. Overview of the OCFL

The OCFL is a way of organizing files on a hierarchical filesystem or object store. Logically, it presents any number of an OCFL Objects, which are each a versioned directory, to the user. On the filesystem, these objects are stored under an OCFL storage root, and each object is given its own

subdirectory. An OCFL Object contains metadata, content files, and additional subdirectories needed to preserve the content files. To save storage space a forward-delta method is employed to deduplicate files that are unchanged between versions. The first time a file is added to an object, it is saved to disk, and, if on subsequent versions the file is unchanged, a reference to the previously-saved file is made. To make the forward-delta method work, a distinction is made between the logical path of a file and its existing file path. The logical path is the name of the file in the logical view of the object, and the existing file path is the path where the content is stored on the filesystem. The deduplication only happens with files inside the same object; in particular, the deduplication is not done across the repository as a whole and does not cross object boundaries. Making a distinction between logical and existing paths provides some benefits, such as making (logical) renames cheap and allowing logical paths to permit filenames in utf-8, even if the underlying storage filesystem does not support it.

Alongside the OCFL specification [1], the implementation notes [18] provide practical guidance and discussion of implementation choices. We summarize key ideas from the specification and implementation notes here.

3.1. Digital Preservation

A key goal of the OCFL is the rebuildability of a repository from an OCFL storage root without additional information resources. Thus, implementers should be careful to ensure that OCFL Objects contain all the data and metadata required to achieve this. With reference to the OAIS model [19], this would include all the descriptive, administrative, structural, representation, and preservation metadata relevant to the object. It is also recommended that a copy of the OCFL specification and any other local documentation is stored in the top level of the OCFL storage root to make it more self-documenting.

The ability to check file fixity is usually considered essential to preservation. OCFL Objects always include digests for every file in every version. The recommended algorithm for the current version of the specification is SHA512 [20], which is also well suited to fixity checks. The OCFL validation requires that digests and files match so that validation implements fixity checking. The OCFL allows additional or alternative fixity algorithms to be used, either to retain legacy information or to meet particular requirements with new algorithms. These may be made in a fixity block of the JSON inventory file, which has the same layout as a manifest block. While the manifest block identifies all files stored in the object via a SHA512, the fixity block permits a broader range of algorithms. The fixity block does not have to include all the files in an object to permit legacy fixity to be imported without requiring continued use of obsolete digest algorithms.

The OCFL's selection of SHA512 as the primary digest, along with the option of storing additional fixity checksums in the inventory file, allows other types of objects to migrate to the OCFL. Migrated objects can either drop existing fixity-checking algorithms and/or retain them without having to re-compute them should a repository choose to make the switch to a new algorithm. When a vulnerability is discovered in SHA512, the next version of all objects can be computed using an unbroken digest and the previously-computed SHA512 checksums can be added to the list of fixity checksums in the inventory file.

3.2. OCFL Objects

The OCFL separates the existing file path of stored files from the logical file path of these files' content in OCFL object versions. This is a key feature that allows previous versions of objects to remain immutable but permitting deduplication, forward-delta differencing, and easy file renaming. Consequently, the OCFL requires only that files added to any version of an OCFL Object must be stored somewhere within the relevant version directory, with a corresponding entry in the manifest block of the JSON inventory file. The state block determines mapping from the file, referenced by its digests with the path in the manifest, to the path and name of the file within that version of the object's content. The most transparent approach is to use the same path on the disk as the path of the file within the object when accessioned. This may not be possible when, for example, complex objects with

deep file hierarchies may encounter issues if they come from a filesystem that allows longer paths than are supported by the target OCFL system. In this case, the decoupling between existing file paths and logical file paths in the OCFL allows the use of truncated paths for storage, while the full paths can be preserved in state block entries which are not length constrained.

The OCFL supports optional deduplication if a client ensures that all digests in the manifest block refer to a single file path on disk. This entry is created the first time file content is stored in an OCFL Object. Subsequent references to that file content should then occur in the state block only. This can be determined by computing the digests of incoming files and determining if they already exist in the manifest block. Similarly, content that is unchanged in subsequent versions is included by simply referring to the existing digest and, hence, the original copy. This implements forward-delta differencing, so that the storage cost of a new version is only the cost of files added or changed.

Filesystem metadata (e.g., permissions, access times, and creation times) are not portable between filesystems, preserved through file transfer operations, nor validated by fixity checks. The OCFL does not expect that these attributes remain consistent so, if retaining this metadata is important, then either the files should be encapsulated in a filesystem image format that preserves this information or the metadata should be extracted and stored explicitly in an additional file. Similarly, an empty directory consists only of filesystem metadata and therefore, as noted above, is not amenable to direct preservation in an OCFL Object. If the preservation of empty directories is considered essential then the suggested route is to insert a zero-length file, such as ".keep", into the directory, which will ensure directories are preserved as part of the object.

OCFL Object versions are composed of a series of files/bitstreams, but the OCFL does not make any distinction between different types of files other than those reserved for the OCFL's functionality: The inventory, its digest file, and conformance declaration files. It is possible, for example, to create separate data and metadata directories within each version to help organize material, but all files are treated equally for the purpose of the OCFL's validation and management.

Objects that contain a large number of files can pose performance problems if they are stored in a filesystem as-is. Fixity checks, object validation, and version creation can require an OCFL client to process all the files in an object, which can be time consuming. Additionally, most storage systems have a minimum block size for allocation to files, so a large number of small files can end up occupying a volume of storage significantly larger than the sum of the individual file sizes. It may be sensible to package small files together in a single, larger file (e.g., ZIP or TAR). This can be parsed to extract individual files if necessary but can significantly improve the efficiency of the basic OCFL client and storage operations.

3.3. OCFL Storage Hierarchy

The basic requirement for an OCFL Storage Root is that the OCFL Objects are stored in directories, distributed beneath it in the underlying storage system. Each OCFL Object is identified by the presence of the conformance file in the object root. These definitions allow a lot of freedom as to how objects are arranged beneath an OCFL Storage Root and, while there is no strict requirement for all OCFL Objects to be arranged according to the same system, it is nevertheless good practice to do so. In addition, in the interests of rebuildability, it would be prudent to include an indication of the details of this arrangement in the storage root. In the interests of transparency, and to support random access to an object, it makes sense for an object's unique identifier and its location under the OCFL Storage Root to be aligned and be simply derivable from each other. Good examples include:

- Flat: Each object is contained in a directory with a name that is simply derived from the unique identifier of the object, possibly with the escaping/replacement of characters that are not permitted in file/directory names. This arrangement is shown in Figure 1. While this is a very simple approach, most filesystems begin to encounter performance issues when directories contain more than a few thousand files, so this arrangement is best suited to repositories with a small number of objects (or many OCFL Storage Roots).

- Pairtree: Designed to overcome the limitations on the number of files in a directory that most filesystems have. It creates a heirarchy of directories by mapping identifier strings to directory paths two characters at a time, as shown in Figure 2. For numerical identifiers specified in hexadecimal, this means that there are a maximum of 256 items in any directory, which is well within the capacity of any modern filesystem. However, for long identifiers, Pairtree creates a large number of directories, which will be sparsely populated unless the number of objects is very large. Traversing all these directories during validation or rebuilding operations can be slow.
- Truncated n-tuple tree: This approach aims to achieve some of the scalability benefits of Pairtree while limiting the depth of the resulting directory hierarchy. To achieve this, the source identifier can be split at a higher level of granularity and only a limited number of the identifier digits are used to generate directory paths. For example, using triples and two levels with the example above yields the example shown in Figure 3.

```
[storage_root]
    ├── 0=ocfl_1.0
    ├── ocfl_1.0.txt (optional OCFL spec)
    ├── d45be626e024
    │   ├── 0=ocfl_object_1.0
    │   ├── inventory.json
    │   ├── inventory.json.sha512
    │   └── v1...
    ├── d45be626e036
    │   ├── 0=ocfl_object_1.0
    │   ├── inventory.json
    │   ├── inventory.json.sha512
    │   └── v1...
    ├── 3104edf0363a
    │   ├── 0=ocfl_object_1.0
    │   ├── inventory.json
    │   ├── inventory.json.sha512
    │   └── v1...
    └── ...
```

Figure 1. An example of a flat Oxford Common File Layout (OCFL) Storage Hierarchy.

```
[storage_root]
    ├── 0=ocfl_1.0
    ├── ocfl_1.0.txt (optional OCFL spec)
    ├── d4
    │   └── 5b
    │       └── e6
    │           └── 26
    │               └── e0
    │                   ├── 24
    │                   │   └──d45be626e024
    │                   │       ├── 0=ocfl_object_1.0
    │                   │       └── ...
    │                   └── 36
    │                       └──d45be626e036
    │                           ├── 0=ocfl_object_1.0
    │                           └── ...
    │
    ├── 31
    │   └── 04
    │       └── ed
    │           └── f0
    │               └── 36
    │                   └── 3a
    │                       └── 3104edf0363a
    │                           ├── 0=ocfl_object_1.0
    │                           └── ...
    │
    └── ...
```

Figure 2. An example of a Pairtree OCFL Storage Hierarchy.

```
[storage_root]
    ├── 0=ocfl_1.0
    ├── ocfl_1.0.txt (optional OCFL spec)
    ├── d45
    |   └── be6
    |       ├──d45be626e024
    |       |  ├── 0=ocfl_object_1.0
    |       |  └── ...
    |       └──d45be626e036
    |          ├── 0=ocfl_object_1.0
    |          └── ...
    ├── 310
    |   └── 4ed
    |       └── 3104edf0363a
    |           ├── 0=ocfl_object_1.0
    |           └── ...
    └── ...
```

Figure 3. An example of a Truncated n-tuple tree OCFL Storage Hierarchy.

3.4. Client Behaviors

The OCFL and its inventory structure are designed to support and capture the following file operations that create OCFL versions, regardless of whether optional features, such as deduplication, are used. The OCFL is not concerned with the process of creating versions, but only the final outcome in terms of the differences with the previous version that need to be recorded and preserved.

- Inheritance: By default, a new version of an OCFL Object inherits all the filenames and file content from the previous version. This serves as the basis against which changes are applied to create a new version. A newly created OCFL Object inherits no content and is populated by file additions.
- Addition: Adds a new logical file path and corresponding new content with a physical file path to an OCFL Object. The logical path cannot exist in the previous version of the object, and the content cannot have existed in any earlier versions of the object.
- Updating: Changes the content pointed to by a logical file path. The path must exist in the previous version of the OCFL Object, and the content cannot have existed in any earlier versions of the object.
- Renaming: Changes the logical file path of existing content. The path cannot exist in the previous version of the OCFL Object, and the content cannot have existed in any earlier versions of the object.
- Deletion: Removes a logical file path (and hence the corresponding content) from the current version of an OCFL Object. The path and content remain available in earlier versions of the object.
- Reinstatement: Makes content from a version earlier than the previous version available in the current version of an OCFL Object. The content must exist in an earlier version and not the previous version. The logical file path may exist in the previous version, effectively updating the file path with older content, or it may not, effectively adding the older content as a new file.
- Purging: Purging, as distinct from deletion, covers the complete removal of content from all versions of an OCFL Object. This is a special case that is not supported as part of regular OCFL versioning operations. An approach to implementing this is covered below.

All versions in an object must use the same version numbering layout which can be easily determined by looking at one existing version—if the digit following V is a zero, then the number format is zero-padded to fixed length, otherwise it is simply an integer. Previous versions of an object should be considered immutable, since the composition of later versions of an object may be dependent on them. One key consequence of this immutability is that manifest entries should never be deleted. New entries may be created, and, if not deduplicating file content, additional references to copies of stored content may be added.

Sometimes a file needs to be deleted from all versions of an object (i.e., purged), perhaps for legal reasons. Doing this to an OCFL Object breaks the version immutability assumption and is not supported directly. The correct way to do this is to create a new object that excludes the offending file, with a revised version history taking this into account. The original object can then be deleted in its entirety. The new object need not have the same identifier as the original object. In this case, the deleted object can be replaced by a "stub" object with the original identifier and location in the OCFL Storage Root.

4. Conclusions

A total of 20 years of experience with preservation repositories has demonstrated the need for robust and efficient storage strategies that are application-independent. This is the goal of the OCFL. The OCFL is designed so that a repository can be rebuilt from just the data in the OCFL Storage Root, which implies that all information about each digital object must be serialized in the corresponding OCFL Objects. The structure of an OCFL system is human parseable, and digital objects are recoverable with basic filesystem tools. Portability is provided by relying on only a minimal set of capabilities that are implemented by most filesystems and object stores. Compatibility with tape is provided through immutability of individual version directories.

The OCFL supports provenance and versioning through the capture of a version history for objects and provision for the implementation of an audit trail. Versioning is efficient because content that is unchanged between object versions is deduplicated. Fixity verification is supported, both as a by-product of the use of content-based addressing within OCFL objects and support for additional fixity information.

The development of the OCFL is an open process that encourages community engagement and input. The OCFL specification is nearing a beta release and software implementation is underway at multiple institutions.

Author Contributions: All authors contributed to this article. J.M. provided background from Stanford (Section 2.1), D.B. provided background from Notre Dame (Section 2.2), and R.M. provided background from Emory (Section 2.3). The remaining sections were contributed by all authors.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Oxford Common File Layout Specification. Available online: https://ocfl.io/0.1/spec (accessed on 27 February 2019).
2. The Archive Ingest and Handling Test. Available online: http://www.digitalpreservation.gov/partners/aiht.html (accessed on 27 February 2019).
3. National Geospatial Digital Archive Project. Available online: http://www.digitalpreservation.gov/partners/ngda.html (accessed on 27 February 2019).
4. Cramer, T.; Kott, K. Designing and Implementing Second Generation Digital Preservation Services: A Scalable Model for the Stanford Digital Repository. *D-Lib Magazine 2010*, 2010; Volume 16, Number 9/10. [CrossRef]
5. 6.3.1 Archival Information Package (AIP). Available online: https://www.iasa-web.org/tc04/archival-information-package-aip (accessed on 27 February 2019).
6. Anderson, R. The Moab Design for Digital Object Versioning. *Code4Lib J.* **2013**. Available online: https://journal.code4lib.org/articles/8482 (accessed on 27 February 2019).
7. NetApp Data Compression, Deduplication, and Data Compaction. Available online: https://www.netapp.com/us/media/tr-4476.pdf (accessed on 27 February 2019).
8. ZFS Deduplication. Available online: https://blogs.oracle.com/bonwick/zfs-deduplication-v2 (accessed on 27 February 2019).
9. Pairtrees for Collection Storage (V0.1). Available online: https://confluence.ucop.edu/display/Curation/PairTree?preview=/14254128/16973838/PairtreeSpec.pdf (accessed on 27 February 2019).

10. How Do I Use Folders in an S3 Bucket? Available online: https://docs.aws.amazon.com/AmazonS3/latest/user-guide/using-folders.html (accessed on 27 February 2019).
11. MD5. Available online: https://en.wikipedia.org/wiki/MD5#Collision_vulnerabilities (accessed on 27 February 2019).
12. We have Broken SHA-1 in Practice. Available online: https://shattered.io/ (accessed on 27 February 2019).
13. D-flat. Available online: https://confluence.ucop.edu/display/Curation/D-flat (accessed on 27 February 2019).
14. The BagIt File Packaging Format (V1.0). Available online: https://tools.ietf.org/html/rfc8493 (accessed on 27 February 2019).
15. File Format Recommendations. Available online: https://wiki.service.emory.edu/display/DLPP/File+Format+Recommendations (accessed on 27 February 2019).
16. Preservation Event and Workflow Recommendations. Available online: https://wiki.service.emory.edu/display/DLPP/Preservation+Event+and+Workflow+Recommendations (accessed on 27 February 2019).
17. Archival Information Package Recommendations. Available online: https://wiki.service.emory.edu/display/DLPP/Archival+Information+Package+Recommendations (accessed on 27 February 2019).
18. Implementation Notes, Oxford Common File Layout Specification 0.1. Available online: https://ocfl.io/0.1/implementation-notes/ (accessed on 27 February 2019).
19. OAIS Reference Model (ISO 14721). Available online: http://www.oais.info/ (accessed on 27 February 2019).
20. FIPS PUB 180-4 Secure Hash Standard. U.S. Department of Commerce/National Institute of Standards and Technology. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (accessed on 27 February 2019).

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).