UFC2: User-Friendly Collaborative Cloud

Minghao Zhao, *Student Member, IEEE*, Zhenhua Li, *Senior Member, IEEE*, Wei Liu, Jian Chen, and Xingyao Li

Abstract—This article studies how today's cloud storage services support collaborative file editing. As a tradeoff for transparency and user-friendliness, they do not ask collaborators to use version control systems but instead implement their own heuristics for handling conflicts, which however often lead to unexpected and undesired experiences. With specialized measurements and reverse engineering, we unravel a number of their design and implementation issues as the root causes of poor experiences. Driven by the findings, we propose to reconsider the collaboration support of cloud storage services from a novel perspective of *operations* without using any locks. To enable this idea, we design intelligent and efficient approaches to the inference and transformation of users' editing operations, as well as optimizations to the maintenance of files' historical versions and the update of individual files. We build an open-source system UFC2 (User-Friendly Collaborative Cloud) to embody our design, which can avoid most (98%) conflicts with little (2%) overhead.

Index Terms—Cloud storage, collaborative editing, conflict resolution, operation inference, operation transformation

1 Introduction

The functionality of cloud storage services, such as Dropbox, OneDrive, Google Drive, and iCloud Drive, has evolved from simple file backup to online collaboration. For example, over 300,000 teams have adopted Dropbox for business collaboration. These users submit ~4000 file edits to Dropbox per second [16]. For the ease of use, file collaboration is made transparent by almost every cloud storage service today through automatic file synchronization. When a user modifies a file in a "sync folder" (a local directory created by the service), the changed file will be automatically synchronized with the copy maintained at the cloud side. Then, the cloud server(s) will further distribute the new version of the file to the other users sharing the file.

Collaboration inevitably introduces *write-write conflicts*[1] – simultaneous edits on two different copies of the same file. However, it is non-trivial to automatically resolve conflicts, especially if the conflicting edits are on the same line of the file. Version Control Systems (VCSes) such as Git and SVN, for instance, provide instructions to find and merge concurrent edits at the line level. Changes on the same or adjacent lines will be marked as conflicts (e.g., Git flags concurrent edits on two continuous lines as conflicts [48]), which are then left to users for manual handling [55]. Nevertheless, VCSes are known to be difficult for non-technical users [62]. Even for computer professionals, solving conflicts in VCSes is somewhat miserable [24], [28], [41], [60].

Instead, today's cloud storage services all opt for transparency and user-friendliness – they devise different approaches to preventing conflicts or automatically resolving conflicts. Unfortunately, these efforts do not work well in practice, often resulting in unexpected results. Table 1 describes the major common patterns of unexpected and undesirable collaborative experiences. To debug these patterns from the inside out, we study eight widely-used cloud storage services (including Dropbox, OneDrive, Google Drive, iCloud Drive, Box [2], SugarSync [17], Seafile [12], and Nutstore [8]) based on traffic analysis with trace-driven and strategically devised experiments, as well as reverse engineering. We collect ten real-world collaboration traces, among which seven come from the users of different services and the other three are from the contributors of well-known projects hosted by Github. Our study results reveal a number of design issues with regard to collaboration support in the eight services. In particular, we find that:

- Using file-level locks to prevent conflicts is difficult due to the unpredictability of users' editing behavior (as cloud storage services can neither designate nor monitor the editor) and the latency between clients and the server.
- Existing conflict-resolution solutions are too coarse-grained and do not consider user intention – they either keep the latest version based on the server-side timestamp or distribute all the conflicting versions to the users.

Most surprisingly, we observe that the majority of "conflicts" reported by these cloud storage services are not *true conflicts* but are artificially created. In those false-positive conflicts (or false conflicts), the collaborators were editing different parts of a shared file. This is echoed by the common

1. We mainly focus on solving write-write conflicts, as they are more severe in degrading user experiences. Cf. Section 5 of supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3132496, for descriptions of read-write and write-write conflicts and our system's reaction in dealing with them.

- *The authors are with the School of Software and BNRist, Tsinghua University, Beijing 100190, China. E-mail: {zhaominghao.thu, lizhenhua1983, liuwei199803, chenjian1995.thu, lixingyao816}@gmail.com.*

Manuscript received 14 Feb. 2021; revised 13 Nov. 2021; accepted 28 Nov. 2021. Date of publication 3 Dec. 2021; date of current version 14 Feb. 2022.
This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grants 61822205, 61632020, 61632013, and 61902211.
(Corresponding author: Zhenhua Li.)
Recommended for acceptance by V. Cardellini.
Digital Object Identifier no. 10.1109/TPDS.2021.3132496

TABLE 1
Common Patterns of Unexpected and Undesired Collaborative Editing Experiences Studied in This Paper

Pattern 1: Losing updates Alice is editing a file. Suddenly, her file is overwritten by a new version from her collaborator, Bob.	All studied cloud storage services
Pattern 2: Conflicts despite coordination Alice coordinates her edits with Bob through emails to avoid conflicts by enforcing a sequential order. Every edit is saved instantly, but conflicts still occur.	All studied cloud storage services
Pattern 3: Excessively long sync duration Alice edits a shared file and confirms that the edit has been synced to the cloud. However, Bob does not receive the updates for a very long duration.	Dropbox, OneDrive, SugarSync, Seafile, Box
Pattern 4: Blocking collaborators by opening files Alice simply opens a shared Microsoft Office file without making any edits. This mysteriously disables Bob's editing the file.	Seafile (only for Microsoft Office files)
Pattern 5: Inability to save Bob finishes editing and hopes to save his edits. However, the editor refuses to save the file. Sometimes the editor forces to save it as a separate copy.	All studied cloud storage services
Pattern 6: Mysterious disappearance of edits Alice finished editing and her edits have been uploaded. However, her edits are not received by any collaborators and her local copy is overwritten.	All studied cloud storage services
Pattern 7: Multiple conflicts w.r.t. sequential edits Alice uploaded her version to the cloud and stopped editing. Afterwards, Bob intermittently edits and saves the file, but multiple "conflict copies" appear.	Box SugarSync Seafile

practice of mitigating false conflicts in cloud storage service-based collaborative editing by intentionally dividing an entire text file into multiple separate files [14], [20]. Such false conflicts can be automatically resolved at the server side without user intervention. In this paper, we show that it is feasible to provide effective collaboration support in cloud storage services by intelligently merging conflicting file versions using the *three-way merge* method [57], [67], where two conflicting versions are merged based on a common-context version. This is enabled by the inference and transformation of users' editing operations; meanwhile, no lock is used so as to achieve the transparency and user-friendliness. As depicted in Fig. 1, our basic idea is to first infer the collaborators' operation sequences [Fig. 1a], then transform these sequences based on their true conflicts (if any) [Fig. 1b] to generate the final version, and finally deliver the merged version to the clients [Fig. 1c]. Compared to a file-level or line-level conflict resolution (e.g., adopted by Dropbox or Git), our solution is more fine-grained: modifications on different parts of the same file or even the same line can be automatically merged. Building a system with the above idea, however, requires us to address two technical challenges. First, inferring operation sequences in an efficient way is non-trivial, since it is a computation-intensive task for cloud storage services.[2] As illustrated in Fig. 1a, when two versions V_1 and V_2 emerge, we need to first find the latest *common-context* version V_0 hosted at the cloud, and then infer two operation sequences S_1 and S_2 that convert V_0 to V_1 and V_2, respectively. The common approach using dynamic programming may take excessive time in our scenario, e.g., ~6.5 min for a 5-MB file. To address the issue, we first conduct *region narrowing* to significantly reduce the problem scale by cutting off the common prefix and suffix, given that edits usually only occur to a small part of a file (termed an *edit region*); then, we leverage an *edit graph* [58] to efficiently extract operations within the edit regions. With such efforts, the inference time is essentially reduced, e.g., ~1 second for a 5-MB file.

The second challenge is how to transform and merge S_1 and S_2 into S_r with *minimal conflict*, i.e., 1) simplifying manual conflict resolution of text files by sending only one merged version ($V_{1,2}$) to the collaborators; and 2) retaining the collaborators' editing intentions while minimizing the amount of conflicts to be manually resolved in $V_{1,2}$. As shown in Fig. 1b, it is easy to directly transform and merge S_1 and S_2 via *operation transformation* [39], if there is no true conflict. To address the challenging case (of true conflicts), we utilize a *conflict graph* [54] coupled with *topological sorting* to reorganize all operations, so as to prioritize the transformation of real conflicting operations and minimize their impact on the transformation of other operations.

Besides solving the two challenges, we devise a locality-aware, operation-assisted delta sync (LOADsync) mechanism to quickly deliver updates for large files (>64 KB). When a client uploads a large file to the cloud, LOADsync leverages the locality of edits to accelerate the computation-

2. In contrast, it is straightforward and lightweight to acquire a user's operation sequences in Google Docs [6], Overleaf [11], and similar services, where a dedicated editor is used and monitored in real time.

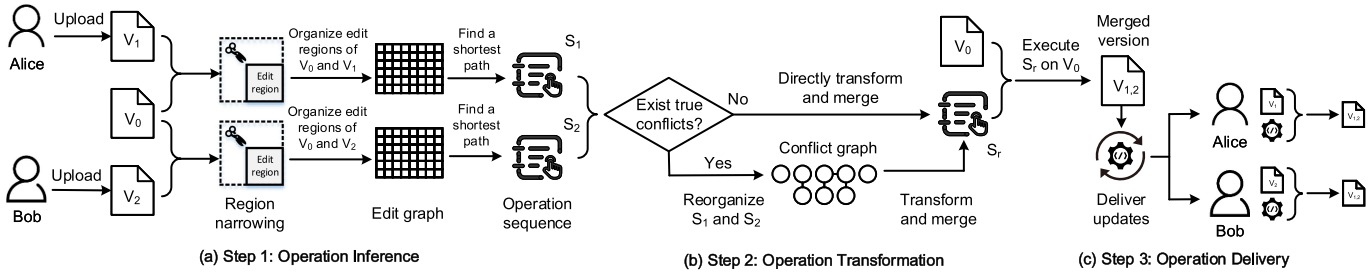


Fig. 1. Working principle of UFC2: (a) inferring the operation sequences S_1 and S_2 that respectively change V_0 to V_1 and V_2, with region narrowing and edit graphs; (b) transforming and merging S_1 and S_2 into S_r with the minimal conflict, based on conflict graphs and topological sorting; (c) generating the merged version $V_{1,2}$ by executing S_r on V_0, and quickly delivering it to clients.

intensive block comparison process in classic delta sync schemes – once two matched blocks are found, it directly moves ahead to compare their successive blocks and omits the checksum-based searching. On the other side, when the cloud distributes a merged version for a large file, LOADsync employs *operation-guided fast reconstitution* which generates instructions to direct the client to fast constitute the merged version from her local version, as shown in Fig. 1c. Additionally, we divide each file into variable-sized chunks for edit-friendly deduplication, and exploit the implicit operations inferred during conflict resolution to simplify the chunking process – only the boundaries of those chunks affected by the operations need recalculation.

We build UFC2 atop Amazon EFS and S3 to implement our design. The evaluation using real-world traces indicates that conflicts generated during collaboration are significantly reduced by 98% on average (the remainder are true conflicts). Meanwhile, the incurred time overhead by a conflict resolution is between 10 and 60 ms for regular-sized files, which is merely 0.6%–4% (2% on average) of the delivery time for a file update. Besides, LOADsync reduces the delivery time for a file update to 20 ms – 2 seconds, outpacing `rsync` [73] by $\sim 5\times$. Moreover, UFC2 is able to simultaneously support ~8500 clients using a typical VM server instance. Finally, all the source code and measurement data are released at https://UFC2.github.io.

2 Design Challenges

In this section, we employ trace-driven experiments, special benchmarks, and reverse engineering to deeply understand the design challenges of collaborative support in today's cloud storage services. In particular, we analyze the root causes of poor experiences listed in Table 1.

2.1 Study Methodology

In order to quantitatively understand how today's cloud storage services behave under typical collaborative editing workloads, we first collected ten real-world collaboration traces as listed in Table 2. Among them, seven are provided by users (with informed consent) that collaborate on code/document writing using different cloud storage services. The other three are extracted from well-known open-source GitHub projects. Each trace contains all the file versions uploaded by every involved user during the collection period.

For the first seven traces, relatively a few (i.e., 5–9) collaborators work on a project for a couple of months. Each of their workloads is unevenly distributed over time: during some periods collaborators frequently edit the shared files, whereas during the other periods there are scarcely any edits to the shared files. By contrast, in the last three traces, a large number of collaborators constantly submit their edits for quite a few months, and thus generate many more file versions. In addition, the collaborators involved in all the ten traces are located across multiple continents.

Using these traces, we conducted a comparative measurement study of eight mainstream cloud storage services: Dropbox, OneDrive, Google Drive, iCloud Drive, Box, SugarSync, Seafile, and Nutstore. For each service, we ran its latest PC client (as of Aug. 2020) on Windows-10 VMs rented from Amazon EC2; these VMs have the same hardware configuration (a dual-core CPU@2.5 GHz, 8 GB memory, and 128 GB SSD storage) and network connection

TABLE 2
Statistics of the Ten Real-World Collaboration Traces

Trace	Timespan	# Col-s	# Files	# Versions	Avg. Size	Major File Types
Dropbox-1	11/2/2018–2/6/2019	5	305	3527	86 KB	tex (52%), pdf (16%), Matlab src (24%) & fig (4%)
Dropbox-2	4/3/2019–5/14/2019	6	216	2193	67 KB	tex (57%), pdf (21%), Matlab fig (9%)
OneDrive	3/15/2019–5/31/2019	5	253	2673	83 KB	tex (61%), pdf (15%), Matlab fig (7%)
iCloud Drive	2/1/2019–4/30/2019	6	301	3211	59 KB	tex (53%), pdf (22%), Matlab fig (12%)
Box	3/21/2019–5/2/2019	8	273	2930	60 KB	tex (66%), pdf (27%)
SugarSync	4/11/2019–5/26/2019	9	325	3472	89 KB	tex (49%), pdf (25%), Matlab src (19%) & fig (3%)
Seafile	2/17/2019–4/30/2019	7	251	2823	71 KB	tex (55%), pdf (19%), Matlab fig (10%)
Spark-Git	1/15/2018–3/27/2019	58	15181	129957	4 KB	Scala (78%), Java (6%), py (5%)
TensorFlow-Git	7/24/2018–3/27/2019	86	16754	246016	9 KB	py (30%), C header (14%) & src (29%), txt (20%)
Linux-Git	9/9/2018–3/30/2019	87	63865	901167	13 KB	C header (31%) & src (42%), txt (16%)

"Col-s" means collaborators, "src" means source code, "py" means python, and "Avg. Size" means average file size.

TABLE 3
A Brief Summary of the Collaboration Support of the Eight Mainstream Cloud Storage Services in Our Study

Cloud Storage Service	Lock Mechanism	Conflict Resolution	Message Queue	File Update Method
Dropbox	No lock	Keep all the conflicting versions	LIFO	`rsync`
OneDrive	No lock	Keep all the conflicting versions	Queue	Full-file sync
Google Drive	No lock	Keep only the latest version	-	Full-file sync
iCloud Drive	No lock	Ask users to choose among multiple versions	-	`rsync`
Box	Manual locking	Keep all the conflicting versions	Queue	Full-file sync
SugarSync	No lock	Keep all the conflicting versions	Queue	`rsync`
Seafile	Automatic/manual$*$	Keep all the conflicting versions	FIFO	CDC
Nutstore	Automatic locking	Keep all the conflicting versions	-	Full-file & `rsync`

"$*$": Seafile only supports automatic locking for Microsoft Office files. "-": we do not observe obvious queuing behavior.

(whose downlink/uplink bandwidth is restricted to 100 / 20 Mbps by WonderShaper to resemble a typical residential network connection [1], [15]).

We deployed puppet collaborators on geographically distributed VMs across five major regions (i.e., including South America, N. Virginia and N. California, Europe, the Middle-East, and the Asia-Pacific) to replay a trace, with one client software and one puppet collaborator running on one VM. For character encoded files including text, src (source code) files, and MATLAB src files, a puppet collaborator first opens the file with Notepad, and then copies the whole content of its successive version to replace the original file content, to simulate users' editing behavior. Afterward, he saves the file at the time according to the *modified time* recorded in file metadata of the collected trace. For versions generated with an interval within 23 seconds (i.e., Median in version generation interval), the editor keeps open and waits for subsequent edits. Otherwise (if the next version is generated in an interval beyond 23 seconds), the editor will be closed. This design simulates the user's editing behavior more precisely and minimizes the effects of editor launching. In terms of files of other kinds (e.g., figures, binary files, and PDF files), we directly copy them to sync folders. We record the system event in the sync folder, such as file arrival time, conflict versions, and the appearance of temporary files. Besides, to safely reduce the duration of the replay, we skipped the "idle" timespan in the trace during which no file is edited by any collaborator.

In addition, we strategically generated some "corner cases" that seldom appear in users' normal editing, so as to make a deeper and more comprehensive analysis. For example, we edited fix-sized small (KB-level) files to measure cloud storage services' sync delay, so as to avoid the impact of file size variation; we edited a random byte on a compressed file to figure out their adoption of delta sync mechanisms; and we performed specially controlled edits to investigate their usage of locks, as well as their delivery time of lock status. Besides, to investigate the effects of different applications (editors) and operating systems, we also manually conducted small-scale collaboration experiments with various applications (including Vim, VS code, MS Office, Adobe Acrobat, Adobe Photoshop, *etc.*) and on UNIX-like operating systems (i.e., Ubuntu 20.04 and macOS Big Sur). Manual experiments are used, as some applications are so heavy that they cannot be run on VMs, while some applications cannot be manipulated with scripts. Manual experiments also bring convenience to observations on details of system and software actions.

We captured all the IP-level sync traffic in the trace-driven and benchmark experiments via Wireshark [21]. From the traffic, we observe that almost all the communications during the collaboration are carried out with HTTPS sessions (using TLS v1.1 or v1.2). By analyzing the traffic size and occurrence time of respective HTTPS sessions, we can understand the basic design of these eight mainstream cloud storage services, e.g., using full-file sync or delta sync mechanisms to deliver a file update.

To reverse engineer the implementation details, we attempted to reverse HTTPS by leveraging man-in-the-middle attacks with Charles [3], and succeeded with OneDrive, Box, and Seafile. For the three services, we are able to get the detailed information of each synced file (including its ID, creation time, edit time, and to our great surprise the concrete content), as well as the delivered lock status and file update. Furthermore, since Seafile is open source, we also read the source code to understand the system design and implementation, e.g., its adoption of FIFO message queues and the CDC delta sync algorithm.

For the remaining five cloud storage services, we are unable to reverse their HTTPS sessions, as their clients do not accept the root CA certificates forged by Charles. For these services, we search the technical documentation (including design documents and engineering blogs) to learn about their designs, such as locks and message queues [4], [7], [9], [10], [18], [19], [27].

2.2 Results and Findings

Our study quantifies the occurrence of conflicts in different cloud storage services, and uncovers their key design principles as summarized in Table 3.

Architecture and Workflow. Based on our multifold efforts (including passive and active measurements, reverse engineering, as well as source code and technical reports reading, *etc.*), we figure out the common architecture and workflow of existing cloud storage services and demonstrate them in Fig. 2. As shown in this figure, each cloud storage service consists of the cloud backend and the client. The cloud utilizes dedicated data structures to organize users' filesystems, e.g., maintaining the filesystem namespace and organizing

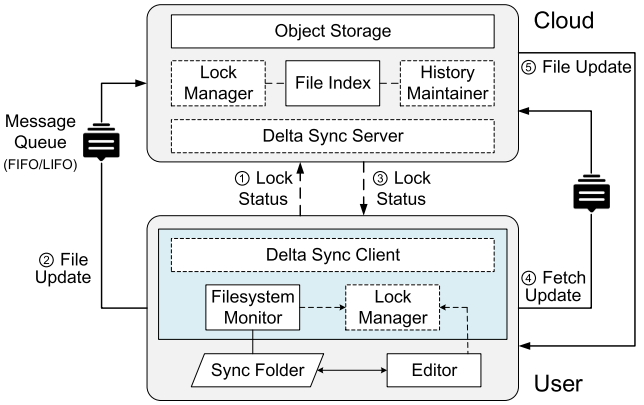


Fig. 2. Workflow and system components for supporting collaboration in mainstream cloud storage services. Solid boxes (lines) denote the common components (workflows) among all the studied cloud storage services, whereas dotted boxes (lines) denote the components (workflows) in specific ones.

the file history (for cloud storage services that support versioning). The files are normally divided into chunks and stored in the object storage. Whenever a user generates a new version, the changed data chunks will be "automatically" synced to the other collaborators.

The sync procedure is achieved by coordinated interactions between the cloud and the client. Specifically, the client monitors user's sync folder, and once a change on a file is detected, the client pushes the update to the cloud through a message queue (Step ②). For cloud storage services using locks, before uploading the file content, the client first issues a lock status (Step ①) to the cloud. This lock status will be immediately forwarded to other collaborators (Step ③), whose clients will then lock the corresponding file. In terms of downloading file chunks, the client periodically sends check/fetch requests to the cloud (Step ④), which also usually go through a message queue before arriving at the cloud, and get responses if newer versions exist (Step ⑤).

Occurrence Probability of Conflicts. When the ten traces are replayed with each cloud storage service, we find considerable difference (ranging from 0 to 4.8%) in the *ratio* of conflicting file versions (generated during a replay) over all versions, as shown in Table 4. Most notably, Google Drive

TABLE 4
Ratio of Conflicting File Versions (Over All Versions) When the Ten Traces are Replayed With Each of the Studied Cloud Storage Services

Trace	DB	OD	GD	ID	Box	SS	SF	NS
DB1	4.4%	4.4%	0	4.5%	4.3%	4.3%	4.3%	3.6%
DB2	4.7%	4.7%	0	4.8%	4.6%	4.7%	4.6%	3.8%
OD	4.1%	4.1%	0	4.2%	4.0%	4.0%	4.1%	3.5%
ID	4.1%	4.0%	0	4.1%	4.1%	4.1%	4.1%	3.4%
Box	4.3%	4.3%	0	4.4%	4.2%	4.3%	4.3%	3.7%
SS	4.2%	4.1%	0	4.2%	4.2%	4.1%	4.2%	3.7%
SF	4.5%	4.5%	0	4.6%	4.5%	4.5%	4.5%	3.8%
SG	1.3%	1.3%	0	1.3%	1.3%	1.3%	1.3%	1.2%
TG	3.5%	3.5%	0	3.5%	3.5%	3.5%	3.5%	3.2%
LG	4.0%	4.0%	0	4.0%	4.0%	4.0%	4.0%	4.0%

DB=Dropbox, OD=OneDrive, GD=Google Drive, ID=iCloud Drive, SS=SugarSync, SF=Seafile, NS=Nut-store, SG=Spark-Git, TG=TensorFlow-Git, and LG=Linux-Git.

TABLE 5
Statistics (in Unit of Second) of the Delivery Time of a File Update, Where the File is Several KBs in Size

Cloud Service	Min	Median	Mean	P99	Max
Dropbox	1.6	2.0	141.2	**312**	**17751**
OneDrive	1.6	4.0	33.4	**106**	**4415**
Google Drive	10.9	11.7	11.7	12.9	18.1
iCloud Drive	8.1	11.8	11.9	11.9	16.9
Box	4.4	5.1	41.8	**115**	**6975**
SugarSync	2.0	6.8	51.3	**124**	**7094**
Seafile	2.7	4.0	53.8	**99**	**9646**
Nutstore	4.2	5.0	5.0	5.0	5.6

appears to have never generated conflicts, because once it detects conflicting versions of a file (at the cloud) it only keeps the latest version based on their server-side timestamps. In contrast, the most conflicting versions are generated with iCloud Drive, because its *sync delay* (i.e., the delivery time of a file update) is generally longer than that of the other cloud storage services (as later indicated in Fig. 5 and Table 5). In comparison, for each trace Nutstore generates the fewest conflicting versions (with Google Drive not considered), as its automatic locking during collaboration can avoid a portion (7.6%–19.1%) of conflicts.

Locks. We observe that the majority of the studied cloud storage services (Dropbox, OneDrive, Google Drive, iCloud Drive, and SugarSync) never use any form of locks for files being edited. As a consequence, collaboration using these products can easily lead to conflicts. Box, Seafile, and Nutstore use coarse-grained file-level locks; unfortunately, we find that their use of locks is either too early or too late,[3] leading to undesired experiences. This is because cloud storage services are unable to acquire users' real-time editing behaviors and thus cannot accurately determine when to request/release locks. Specifically, locking too early leads to Pattern 4 in Table 1, locking too late (locking after editing) leads to Pattern 1, and unlocking too early leads to Pattern 2.

Box only supports manual locks on shared files. When Alice attempts to lock a shared file f and Bob has not opened it, f is successfully locked by Alice and then Bob cannot edit it (until it is manually unlocked by Alice). However, the locks may not always preform as expected. *In essence, Box implements locks by creating a process on Bob's PC,* In this case, if Bob has already opened f when Alice attempts to lock it, he can still edit it but cannot save it (resulting in Pattern 5.), because when Bob attempts to save his edit the file editor (e.g., MS Word) will re-check the permission of f. Seafile automatically locks a shared file f when f is opened by an MS Office application, and f will not be unlocked until it is closed. This locking mechanism is coarse-grained and may lead to Pattern 4. For non-MS Office files, Seafile supports manual locks in the same way as Box, and thus they have the same issue in collaboration.

Nutstore attempts to lock a shared file f automatically, when Alice saves her edit. At this time, if Bob has not opened f, f is successfully locked by Alice and Bob cannot edit it; after Alice's saved edit is propagated to Bob, f is

3. Ideally, a file should be locked right before the user starts editing, and unlocked right after the user finishes the editing.

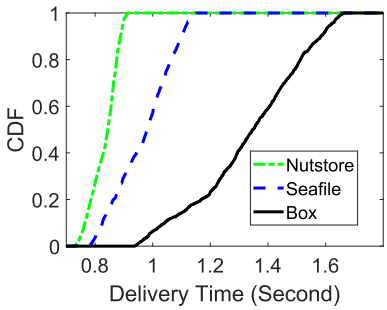


Fig. 3. CDF of the delivery time of a lock status. Note that among all the studied services, only three of them use locks.

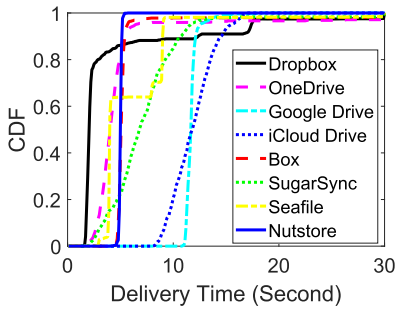


Fig. 5. CDF of the delivery time of a file update, where the file is several KBs in size.

automatically unlocked. However, if Bob opened the shared file before Alice saves the file, Nutstore has the same problems as Box and Seafile.

Finally, we are concerned with the delivery time of a *lock status* (i.e., whether a file is locked). According to our measurements, the lock status is delivered in real time with ∼100% success rates. As shown in Fig. 3, the delivery time ranges from 0.7 to 1.6 seconds, averaging at 1.0 second, much less than the file delivery time (averaging at about 50 seconds). Especially, no tail latency is observed for delivering lock status. Note that tail latency is prominent in Box and Seafile when delivering file content, and it also prevalently exists in cloud systems and web services [33], [34]. Thus, we suspect that today's cloud storage services implement dedicated infrastructure (such as queues with dedicated scheduling [47], [79]) for managing locks.

In summary, implementing desirable locks in cloud storage services is not only complex and difficult but also somewhat expensive. Therefore, we feel it wiser to give up using locks.

Conflict Resolution. We find three different strategies for resolving the conflicts. *First*, Google Drive only keeps the latest version (defined by the timestamp each version arrives at the cloud). All the older versions are discarded and can hardly be recovered by the users (Google Drive does not reserve a version history for any file). Note that this notion of "latest" may not reflect the absolute latest (which depends on the client-side time), e.g., when the real latest version arrives earlier due to network latency. *Second*, iCloud Drive asks the user to choose one version from all the conflicting versions. The user has to compare them by hand, and then make a decision (which is often not ideal). *Third*, a more common solution is to keep all the conflicting versions in the shared folder, and disseminate them to all the collaborators. This solution is more conservative (which does not cause data loss), but leaves all burdens to users.

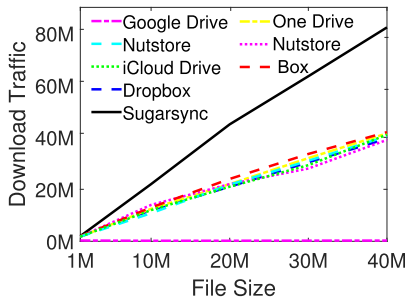


Fig. 4. Sync traffic in whole-file edit (on the side who latterly submit his edit).

Moreover, given the distributed nature, merging efforts from the collaborators could cause further conflicts if not coordinated well.

Given the difficulties in resolving conflicts, we advocate that cloud storage services should make more effort to proactively avoid, or at least significantly reduce, the conflicts.

Conflicts Identification and Replica Generation. We also investigate how the conflicts are identified, as well as how these "conflict versions" are generated and propagated. We notice that all cloud storage services adopt a *server-centric eventual consistency maintenance* strategy; here "server-centric" means only the server maintains version information (e.g., version number) which can be used for detecting conflicts, and "eventual consistency" means finally each collaborator's sync folder achieves the same state, i.e., the file marked as "conflict version" will be identical for every collaborator (including the one who generates this version).

To investigate the workflow in the scenario of conflict, we instrumented Bob to edit shared Z-byte video files, where $Z \in \{1M, 10M, 20M, 30M, 40M\}$. We use video files as they are highly compressed, so as to get rid of the influence of data compression used by cloud storage services, and their relatively large sizes bring convenience to our observation. We capture the sync traffic and monitor the behavior of Bob's client. As shown in Fig. 4, we observe that most of the studied cloud storage services (Dropbox, OneDrive, iCloud, Box, Seafile, and Nutstore) consume 1× downstream traffic when conflicts happen, which are incurred by downloading Alice's copies. In contrast, SugarSync consumes 2× traffic in downloading, which indicates that, it not only downloads Alice's version, but also redownloads the version generated by himself.

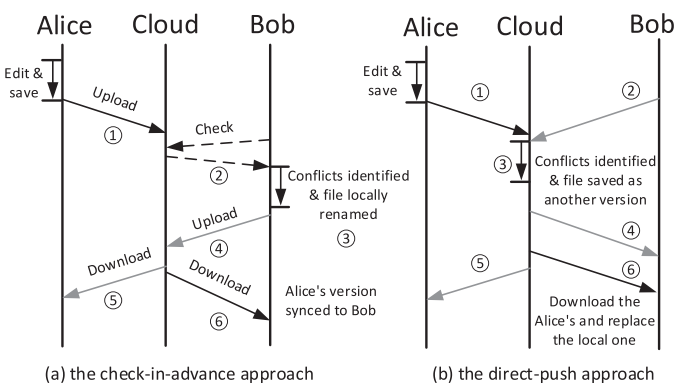


Fig. 6. Two different approaches in conflict identification.

TABLE 6
Cloud Storage Services' Replica Generation Strategies for Continuous Edits With Exclusive Editors; Note That `rename` Operation Will Normally Fail for an Opened File in Windows Operating System

Cloud Storage Services	Conflict detecting Method	Methods in generating conflicting copies in continuous editing with exclusive editors.	multiple conflicts*
Dropbox	Check-in-advance	①MS Office: using applications' interface to enabling subsequent edits made on renamed files. ② Others: abort renaming until Bob close the file.	–
OneDrive	Check-in-advance	Block all syncing until the files are closed.	–
iCloud Drive	-	Caching all conflicting versions as temporary files.	–
Box	Check-in-advance	Generating a separate file for each saving.	
SugarSync	Check-in-advance	Using temp file to catch Alices's version; directly upload Bob's edits.	
Seafile	Direct-push	Generating a separate file at the first-time saving and stop syncing for subsequent edits until the file closed.	
Nutstore	Check-in-advance	Forcing use to solve it as a new file in Unix-like system.	

** the icons of , , and denotes that the multiple conflicts issue exists in Windows, macOS, and Linux operating systems; whereas "-" denotes the services do not encounter this issue in any system.*

In essence, this disparity derives from different methods used for conflict detection and conflict-version generation. Specifically, Dropbox, OneDrive, Box, Seafile, and Nutstore adopt a *check-in-advance* approach. As shown in Fig. 6a, Alice and Bob edited a shared file based on a common, consistent version, and Alice, before Bob, saves and uploads her version to the cloud (step ① and ②). Before syncing the file contents, their clients first check with the cloud, and in this circumstance Bob will be informed he has generated a conflicting version. Thus, Bob's client locally renames Bob's version as "conflicted copy" (step ③) and uploads it (i.e., the renamed version) to the cloud (step ③). This renamed version is regarded as a new file and synced to Alice (step ⑤). Finally, Alice's version, with the original name, is delivered to Bob (step ⑥).

In comparison, SugarSync adopts a *direct-push* approach. As shown in Fig. 6b, whenever a Bob's version is generated, it will be directly delivered to the cloud without in-advance conflict checking (step ②). If Alice has uploaded her version before (step ①), the cloud will identify Bob's version as "conflicted copy", since his version is not generated based on the newest version at the cloud (i.e., Alice's version). Accordingly, Bob's version will then be renamed at the cloud (step ③) and delivered to both sides (step ④ and ⑤). When this file delivery procedure finishes, there will be two files with identical content but different names appearing at Bob's side, i.e., a renamed "conflicted copy" from the cloud and the one kept on his local folder. Finally, Alice's version is synced to Bob (with `rsync`, detailed latter) to overwrite the file with the original name (step ⑥).[4]

However, such designs cannot ensure the aforementioned conflict prevention and resolution strategies work as expected. When Bob is editing a file and Alice's version begins to be synced to him, these protocols will fail in finding conflicts. In such cases, different annoying situations will appear, for different categories of editors being used. Specifically, if Bob is using an *exclusive* editor (not allowing other applications to write the file it opened, such as MS Office and Adobe Acrobat), Alice's edits cannot be synced to Bob, leading to Pattern 3. If Bob is using a *dynamic non-exclusive* editor (allowing other applications to write the file it opened, and meanwhile periodically checking and reloading modifications), Bob's unsaved edits will be overwritten, leading to Pattern 1. Besides, if Bob opens the file with a *non-dynamic non-exclusive* editor with a checking-before-saving mechanism (i.e., the editor allows other applications to modify the opened file; but when the user saves the file, the editor will check whether it has been changed since it was opened, such as VS code), the application will refuse to save it, leading to Pattern 5; in contrast, if an editor without such a checking mechanism is used (such as Adobe Photoshop), Bob successfully saves his edits and overwrites Alice's version (which has been synced to him after he opened the file), and Bob's version will then be transparently synced to Alice and overwrite her local copy, resulting in Pattern 6. In addition, an impeccable and universally applicable implementation of existing conflict prevention strategies is hard, due to the diversity of operating systems and applications their clients work with. Specifically, for the *check-in-advance* approach which renames conflicting files at the client-side, the rename operation will fail when the user opens a file with an exclusive editor. To address this issue, cloud storage services are forced to design their own implementation-level strategies, some of which, however, result in multiple conflicts whenever Bob continuously edits and saves a file (i.e., Pattern 7), as summarized in Table 6. For Dropbox, if the user edits a file with MS Office applications, it invokes the applications' `rename` interface, instead of directly invoking the system interface, so that subsequent operations are directly moved to the new file; but for other none-Office applications, each time Dropbox finds that a conflicting version is generated locally but rename operation fails, it aborts this attempt until the application is closed. For OneDrive, once such a situation is detected, OneDrive stops syncing the corresponding file until Bob closes it. During such a period, both Alice and Bob's updates do not sync to the cloud, and all their (simultaneous) edits are made on their local copies. For Box, however, it saves each edit as a separate file with a file name like "foo.pdf (Bob's iMac-2)."

In terms of the Seafile, it adopts a similar but wiser approach. When such a situation (i.e., Bob's opening a file prevents Alice's edits sync to him) happens, Seafile finds conflict when Bob first saves his edits. Then Bob's client saves his edits as a renamed file like what Box does, with a filename like "foo.doc (Bob's conflicts + timestamp 1)."

4. Note that step ⑤ and ⑥ may operate in a different order, but both achieve the same result.

Afterward, the Seafile client stops syncing the file with the original name (say, "foo.doc"), and all the subsequent edits are made on "foo.doc." The original sync procedure restarts until Bob closes this file; afterward, Bob finds he generated a new conflicting version and then renames it as "foo.doc (Bob's conflicts + timestamp 2)." It is also worth mentioning that Nutstore also suffers this issue in UNIX-like systems, even if it adopts locks to prevent conflicts – it happens when a user begins to edit a file before a lock signal arrives. Note that Nutstore attempts to lock a file when a user first time saves his edits, and it is possible to change permissions for an opened file in UNIX-like systems.

The multiple conflict issue also exists in services using the *direct-push* approach – it happens when a user continuously edits and saves a file with an exclusive editor. In this case, another collaborator's version fails to sync to him (as the editor blocks other processes to modify this file), whereas his subsequent edits will sequentially be uploaded to the cloud and regarded as conflicts. The multiple conflict issue severely impairs user experience, as periodical saving frequently happens in continuous editing, and manually solving conflicts among multiple files is more miserable. In summary, existing conflict resolution and prevention strategies are implemented with general or OS/application-specific deficiencies, either being unable to find conflicts or leading to miserable consequences in solving and preventing conflicts in particular circumstances. Thus, we advocate a conflict resolution method that leaves minimum work to the client-side may help alleviate the implementation difficulty of their clients' running environments.

Delivery Latency and Message Queue. Delivery latency of a file (update) prevalently exists in cloud storage at both infrastructure (e.g., AWS S3) and service (e.g., Dropbox) levels [32], [44], [69], [70], [74]. It stems from multiple factors such as network jitter, system I/O, and load balancing in the datacenter [44], [51]. We measure the delivery time of a file update regarding the eight cloud storage services. As in Fig. 5 and Table 5, some services always have reasonable delivery time. On the other hand, in a few services, the maximum delivery time reaches several hours for a KB-level file, and the 99-percentile (P99) delivery time can reach hundreds of seconds. The unpredictability and long tail latency can sometimes break the time order among file updates, which is the main root cause of Patterns 2 and 3.

Additionally, we find that the implementation of message queues in some cloud storage services aggravates the delivery latency. Specifically, different services have very different message queue implementations, leading to different queueing behaviors. For a FIFO queue (used by Seafile), when the server is overloaded, many requests for file/fetch updates are processed by the server but not accepted by the client due to client-side timeout, thus wasting the server's processing resources. This problem can be mitigated by using LIFO queues (used by Dropbox). However, for a LIFO queue, the requests from "unlucky" users (who encounter the server's being overloaded after issuing fetch update requests) wait for a long duration. We suspect that the services with excessively long delivery time are using big shared queues with no QoS consideration, and may benefit from using a dedicated queue like QJUMP [42].

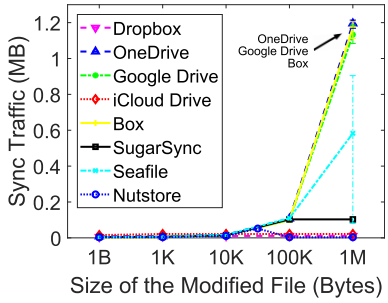


Fig. 7. Sync traffic of a random-byte edit to a compressed file.

File Update Methods. Collaboration results in frequent, short edits to files. *Delta sync* is known to be efficient in updating short edits, compared with *full-file sync* where the whole file has to be transferred [50]. To understand the file update method, we instrumented Alice to modify a random byte in a Z-byte highly compressed file already shared with Bob, where $Z \in \{1, 1K, 10K, 100K, 1M\}$. Fig. 7 shows the traffic usage for delivering the file update on both sides of Alice and Bob. Highly compressed files are used to infer the traffic usage as they cannot be further compressed. We find that the sender and receiver always spend similar amount of traffic for a file update, indicating that they use the same file update method. As shown in Fig. 7, OneDrive, Google Drive, and Box adopt full-file sync: their traffic usage is always larger than Z bytes for any Z; the other services adopt delta sync (`rsync` [73] or CDC [64]). The traffic of Seafile varies from 0.08 MB to 0.9 MB for an 1-MB file, because it uses CDC [13] whose performance depends on the file content; this conforms with its CDC-based data organization of historical file versions (seen from its source code [13]).

In Nutstore, when $Z <= 64$ KB, it is always slightly larger than the file size; when $Z > 64$ KB, it sharply decreases to ~20 KB. This implies that Nutstore adopts a hybrid file update method: full-file sync for small ($\leq$64 KB) files and delta sync for the other files, so as to achieve the highest update speed, because small and large files are more suitable for full-file and delta sync, respectively (full-file sync requires fewer computation overhead and fewer rounds of client-server message exchanges).

2.3 Implications

Our study results show that today's cloud storage services either do not use any locks or use coarse-grained file-level locks to prevent conflicts. The former would inevitably lead to conflicts. The latter, however, is hard to prevent conflicts in practice for two reasons: 1) it is hard to accurately predict user's editing behaviors in real time and therefore to determine the timing of applying the lock, and 2) the latency between the client and the server can vary significantly, so file-level conflicts are generally inevitable. Furthermore, the study shows that full-file and delta sync methods can be combined to accelerate the delivery of a file update. To address the revealed issues, we explore the possibility of developing lock-free conflict resolution by inferring fine-grained user intentions. We also explore a hybrid design of full-file and delta sync methods for efficient file update and synchronization.

3 Our Solution

This section aims to address the challenges uncovered in Section 2. Our key idea is to model file editing events as `insert` or `delete` *operations* (Section 3.2). Based on the operation model, we *infer* the collaborators' operation sequences (Section 3.3), and then *transform* these sequences (Section 3.4) based on their conflicts to generate the final version. We explain the above procedure with a simple case of two file versions, and demonstrate its applicability in handling multiple versions (Section 3.5). Besides, we also devise an efficient historical version maintenance and deduplication scheme to facilitate conflict resolution (Section 3.6), and design a novel file update scheme, the LOADsync (Locality-aware, Operation-Assisted Delta rsync), to efficiently deliver edits (Section 3.7).

3.1 True and False Conflicts

We examine the conflicting file versions as listed in Table 4 in great detail. We find that $\sim 1/3$ of them come from non-text (e.g., PDF or EXE) files, which, as mentioned in Section 1, are typically generated based on text files and thus can be simply deleted or regenerated from text files for pretty easy conflict resolution. The rest ones relate to text files, the vast majority of which, to our surprise, only contain "false positive" conflicts as the collaborators in fact operated on different parts of a shared file.

Take the Dropbox-1 collaboration trace as an example. When it is replayed with Dropbox or OneDrive, among the 3,527 file versions hosted at the cloud side, 154 text files are considered (by Dropbox and OneDrive) to be conflicting versions and then distributed to all the collaborators. Actually, 152 out of the 154 *apparently* conflicting versions can be correctly merged at the cloud side. The remaining two cannot be correctly merged as two collaborators happen to edit the same part of the shared file in parallel, thus generating 9 *true* conflicts. In other words, the vast majority of the (coarse-grained) file-level conflicts are *false* (positive) conflicts when seen at the (fine-grained) operation level.

3.2 Explicit and Implicit Operations

We model *operation* as the basic unit in collaborative file editing. A shared file can be regarded as a sequence of characters, and an *explicit* operation is a user action that has truly occurred to the shared file, modifying some of its characters. In detail, an explicit operation O consists of seven properties:

- There are two possible *operation types*: `insert` and `delete`; $O.type$ represents the operation type of O.
- The *targeted string* is the string that will be inserted or deleted by O, which is denoted by $O.str$.
- The *length* of O is the (character) length of $O.str$, which is denoted by $O.len$.
- The *position* of O is where $O.str$ will be inserted to or deleted from in the shared file, which is denoted by $O.pos$.
- O must be performed on a context (file version), which is called the *base context* of O, or denoted as $O.bc$.
- O is performed on $O.bc$ to generate a new context, which is called the *result context* of O, or denoted as $O.rc$.
- The range of characters impacted by O in $O.bc$ is the *impact region* of O, denoted as $O.ir$. It is calculated as

$$O.ir = \begin{cases} [O.pos, O.pos + 1) & \text{if } O.type = \texttt{insert}; \\ [O.pos, O.pos + O.len) & \text{if } O.type = \texttt{delete}. \end{cases}$$

This formula tells that when a string is inserted to $O.bc$, the insert operation only affects the position (in $O.bc$) where the string is inserted; but when a string is deleted from $O.bc$, the positions where all the characters of the string formerly appear at $O.bc$ are affected.

Automatically acquiring a user's explicit operations is trivial and lightweight when the editor can be monitored, e.g., in Google Docs [6] and Overleaf [11]. In these systems, users are required to use a designated online file editor, by monitoring which all the collaborators' explicit operations can be directly captured in real time.

In contrast, our studied cloud storage services are supposed to work independently with any editors and support any types of text files, thus bringing great convenience to their users (especially non-technical users). Therefore, we do not attempt to monitor any editors or impose any restrictions on the file types, and thus cloud storage services cannot capture users' explicit operations. Instead, we choose to analyze users' *implicit operations* based on the numerous file versions hosted at the cloud side. For a shared file f, implicit operations represent the cloud-perceived content changes to f (i.e., the eventual result of a user's editing actions), rather than the user's editing actions that have actually happened to f. Obviously, implicit operations, as well as their various properties, have to be indirectly *inferred* from the different versions of f. Since we focus on implicit operations in this work, we simply use "operations" to denote "implicit operations" hereafter.

3.3 Operation Inference (OI)

When no conflict happens, inferring the operations from two consecutive versions of a file is intuitive, so in this part we only consider the OI when two conflicting versions emerge at the cloud. Note that when there are more than two conflicting versions, our described algorithms below still apply. When two conflicting versions of a file, V_1 and V_2 are uploaded to the cloud by two collaborators, the cloud first pinpoints their latest *common-context* version V_0 hosted in the cloud. Generally, the cloud knows which version is consistent with a collaborator's local copy during her last connection to the cloud. When the collaborator uploads a new version, this "consistent" version is regarded as the base context (version) of the new version, so that all versions of a shared file constitute a *version tree*, in which the parent of a version is its base context. Therefore, to pinpoint V_0 is to find the latest common ancestor of V_1 and V_2 in the version tree.

After pinpointing V_0, the cloud starts to infer the operation sequences (S_1 and S_2) that change V_0 to V_1 and V_2, respectively. To infer S_1, the common approach is to first find the longest common subsequence (LCS) between V_0 and V_1 using *dynamic programming* [29], [45], [61]. Then, by

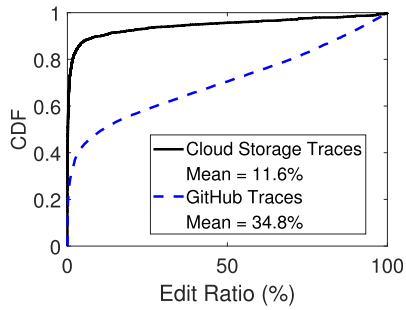


Fig. 8. CDF of edit ratio in cloud storage service and Github dataset.

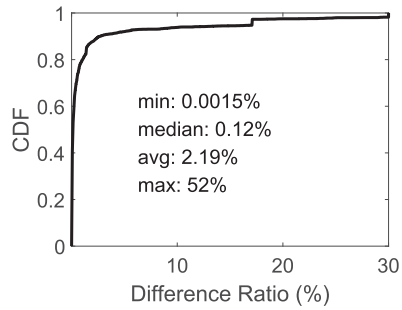


Fig. 9. Difference ratio $\frac{d}{n_0+n_1}$ between every two consecutive versions in the Dropbox-1 trace.

comparing the characters in V_0 and the LCS one by one, a sequence of `delete` operations can be acquired, which changes V_0 to the LCS; in a similar manner, a sequence of `insert` operations that changes the LCS to V_1 can be acquired. After that, the acquired `delete` and `insert` operations are combined to constitute S_1 (S_2 is constituted in a similar manner). Unfortunately, this common approach requires $O(n_0 * n_1)$ computation complexity, which may require considerable time for a large file, e.g., ~6.5 minutes for a 5 MB file.

Multifold efforts have been made to solve this issue. First, it has been observed that real-world file edits typically follow a locality pattern (i.e., edits usually concentrate on a few parts of a file). Such locality feature of file edits has been widely used for data compression and deduplication among cloud storage systems [52], [68], [68], [75], [76], [80], [81]. In terms of collaborations on cloud storage services, the locality feature is especially significant, in which a user's edits are only confined to a very small part of the targeted file. Ae refer to this continuous part that contains edits (i.e., the interval between the first and last edited characters in a file) as *edit region* (Fig. 10). Fig. 8 and Table 7 demonstrate the edit ratio (i.e., the proportion of edit region in the whole file) for files in cloud storage service traces and GitHub traces. It is manifested that the edit ratio is very small for both traces, e.g., over 90% files have an edit ratio less than 20% in cloud storage service traces, and the GitHub traces (with a much larger edit ratio than the cloud storage service traces) have an average edit ratio of less than 40%. This strong locality feature makes it feasible to improve efficiency by conducting region reduction (i.e., clipping), i.e., cutting off the common prefix and suffix of V_0 and V_1, and only applying the LCS algorithm on their edit regions.

TABLE 7
Statistics of Edit Regions Among the Collected Datasets

Trace	Min	Non-0 Min	Median	Max	Mean
Dropbox-1	0	0.004	0.15	100	5.89
Dropbox-2	0	0.002	0.01	99.7	4.23
OneDrive	0.011	0.011	0.84	99.9	18.1
iCloud Drive	0.021	0.021	11.9	75.6	9.2
Box	0	25.561	72.6	99.5	56.8
SugarSync	0	0.013	0.14	97.0	2.70
Seafile	0	0.011	0.21	99.2	2.46
Spark-Git	0.003	0.003	31.2	99.6	39.9
TensorFlow-Git	< 0.001	< 0.001	9.56	99.9	28.2
Linux-Git	< 0.001	< 0.001	20.02	99.9	35.2

Region reduction, which can be finished within linear time, significantly reduces the problem scale of the naturally complex LCS problem – it converts the issue of finding LCS between V_0 and V_1 into finding LCS of their edit regions concatenating their common prefix and suffix.

Afterward, we leverage an *edit graph* [58] to organize the edit regions of V_0 and V_1 (denoted as v_0 and v_1, with n_0 and n_1 bytes in length). As demonstrated in Fig. 11, an edit graph is a directed acyclic graph with vertices at each point in the grid (x, y), where $x \in [0, n_0]$ and $y \in [0, n_1]$. Each vertex is connected to its right and lower neighbors by a horizontal edge (representing an `delete` operation) and a vertical edge (representing a `insert` operation). If $v_0[x] = v_1[y]$, there is a diagonal edge going from vertex $(x-1, y-1)$ to vertex (x, y), which represents a common character between v_0 and v_1. A diagonal edge has weight 0 and a horizontal or vertical edge has weight 1. Then, finding the LCS between the edit regions of v_0 and v_1 is reduced to finding a minimum-cost path in the edit graph that goes from the start point (0,0) to the end point (n_0, n_1). The latter problem can be solved with $O((n_0 + n_1) * d)$ complexity (cf. Section 1 of the supplementary material, available online, for the detailed procedure of LCS calculation with this method), where $d = n_0 + n_1 - 2l$ is the number of horizontal and vertical edges (i.e., the length of difference between v_0 and v_1) and l is the number of diagonal edges (i.e., the length of the LCS) [58]. Note that d is usually much smaller than n_0 and n_1 in practice (as quantified in Fig 9). With all aforementioned efforts, it is feasible for the cloud to infer operations for version V_1 and V_2 in a relatively short time, e.g., for a 5-MB file the inference time is typically optimized from ~6.5 minutes to ~1 second, resulting in a 400× reduction.

3.4 Operational Transformation (OT)

After the operation sequences S_1 and S_2 are inferred, which contain s_1 and s_2 operations respectively (all operations in a sequence are sorted by their position and have the same base context V_0), the cloud first detects whether there exist true conflicts, and then constructs a *conflict graph* [54] (as shown in Fig. 12) if there are any. A conflict graph is a directed acyclic graph that has $s_1 + s_2$ vertices representing

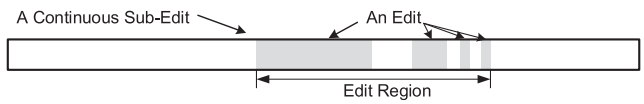


Fig. 10. An example of locality in editing and the edit region.

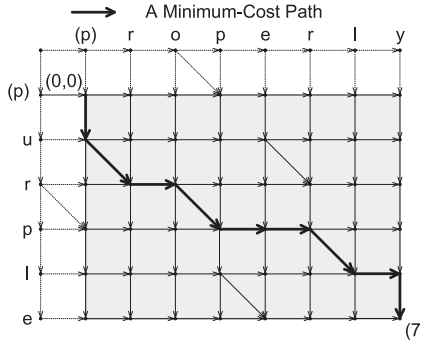


Fig. 11. A simple edit graph for reconciling V_0 (the horizontal word "properly") and V_1 (the vertical word "purple"). A diagonal edge has weight 0 and a horizontal or vertical edge has weight 1; then finding the LCS between V_0 and V_1 is converted to finding a minimum-cost path that goes from the start point (0,0) to the end point (7,5) Cf. Section 1 in the supplementary material, available online, for details about the manipulation and implementation of the edit graph-based searching algorithm.

the aforementioned $s_1 + s_2$ operations. After that, *operation transformation* (OT) [39] is adopted to transform and merge S_1 and S_2 into a result sequence S_r, which can be executed on V_0 to generate the merged file version $V_{1,2}$.

Detecting True Conflicts. In order to detect true conflicts between S_1 and S_2, the cloud first merges S_1 and S_2 into a temporary sequence S_{temp} sorted by the operations' position, and initializes the conflict graph G with $s_1 + s_2$ vertices and 0 edges. Then, for each operation in S_{temp}, the cloud checks whether the operations behind it conflict with it – this is achieved by checking whether the impact regions of two operations overlap each other. If two operations $S_{temp}[i]$ and $S_{temp}[j]$ are real conflicting operations, an edge $e_{i,j}$ connecting v_i to v_j (denoted by solid arrows in Fig. 12) is added to G to represent a true conflict. If there are no true conflicts between any two operations, G is useless and simply discarded. The detection, in the worst case (where each operation in S_1 conflicts with each operation in S_2), bears $O((s_1 + s_2)^2)$ complexity. However, in common cases there exist only a few conflicts, and thus the detection can be quickly carried out with $O(s_1 + s_2)$ complexity.

Basics of OT. As the *de facto* technique for conflict resolution in distributed collaboration, OT [39] has been well

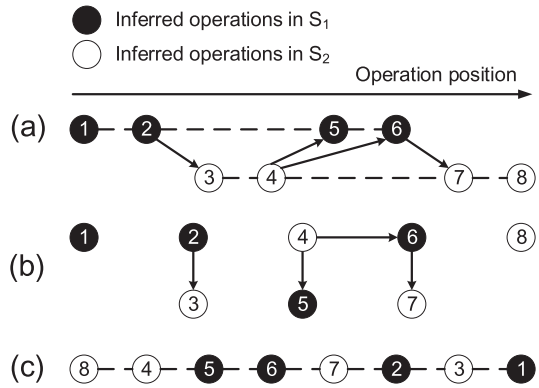


Fig. 12. Reordering conflicting operations with a conflict graph. (a) In the two operation sequences S_1 and S_2, a dashed line denotes a sequence, while a solid arrow represents a true conflict. (b) S_1 and S_2 are reorganized into a conflict graph, where conflicting operations are linked with directed edges. (c) In the result sequence S_r, operations are sorted by their topological order in the conflict graph.

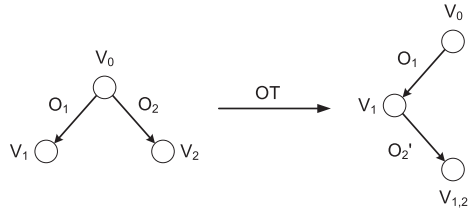


Fig. 13. An example of OT that merges V_1 and V_2, in which O_2 is transformed to O_2' to resolve the conflict between O_1 and O_2.

studied [40], [66] and used (e.g., Google Docs [6], Overleaf [11], and Etherpad [5]). It resolves conflicts by transforming parallel operations on a shared file to equivalent sequential operations (if possible). A very simple example of OT is shown in Fig. 13. More details and examples of OT can be found at Section 2 of the supplementary material of this paper, available online.

OT When There are no True Conflicts. According to our detection results on the ten collaboration traces (cf. Table 2), when a file-level conflict occurs there are no true conflicts with a very high ($>95\%$) probability, which is consistent with the results of our manual examination in Section 3.1. When there are no true conflicts detected, the cloud directly applies OT on S_1 and S_2 to generate S_r and $V_{1,2}$. Traditionally, the computation complexity of OT is deemed as $O((s_1 + s_2)^2)$. In our case, since there are no true conflicts and S_{temp} are already sorted by the operations' position, we choose to transform the operations in S_{temp} in their descending order of position, thus achieving a much lower complexity of $O(s_1 + s_2)$. After the transformation, we get S_r and execute S_r on V_0 to generate the merged version.

OT in the Presence of True Conflicts. If there are true conflicts detected, it is impossible to directly and correctly resolve the conflicts as in the above case. Consequently, we choose to prioritize the mitigation of user intervention while preserving potentially useful information, so as to facilitate users' manual conflict resolution. Specifically, two principles should be followed: 1) the cloud should send only one merged version $V_{1,2}$ to the collaborators for easy manual conflict resolution; and 2) users' editing intentions should be retained as much as possible, while the number of conflicts that have to be manually resolved in $V_{1,2}$ had better be minimized.

To realize the two principles, our first step is to utilize *topological sorting* [46] to reorganize and help transform S_1 and S_2 (via their conflict graph G) following two rules. First, real conflicting operations should be transformed and put into S_r in the ascending order of their position, so that their conflicts can be resolved at one time and thus do not negatively impact the transformation of other operations. Second, non-conflicting operations should be put into S_r in the descending order of their position, so that they can be quickly transformed like in the case of no true conflicts.

After S_1 and S_2 are topologically sorted and put into S_r (see Fig. 12c), we apply our customized OT scheme to embody the aforementioned two principles for resolving true conflicts. First of all, we classify true conflicts into different categories that are suited to different processing strategies. Given two conflicting operations O_1 and O_2 working on the same base context (V_0), there seem to be four different categories of conflicts in the form of "$O_1.type/O_2.type$": 1)

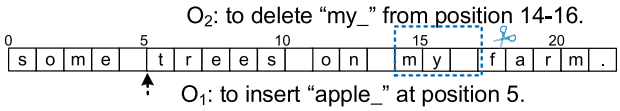


Fig. 14. An Example of the `insert/delete` situation (non-conflict). In this case, the operation O_1 only affects the position 5 (i.e., where the characters are inserted) of this original string, and the characters at positions 6-21 are not changed. For delete O_2, its effect positions 14-15 are also in terms of the original string, independent of operation O_1.

`delete/delete`, 2) `delete/insert`, 3) `insert/delete`, and 4) `insert/insert`. Here "/" means $O_1.pos \leq O_2.pos$. However, by carefully examining the impact regions of O_1 and O_2 ($O_1.ir$ and $O_2.ir$) in each category, we find that `insert/delete` conflicts are never true conflicts, because an insert operation only affects the targeted string at the position it appears and never affects a to-be-deleted string that starts behind this position (as shown in Fig. 14) . Thus, we only need to deal with the other three categories. See supplementary material Section 2.3, available online, and the conference version [30] on how we customize OT to solve conflicts of such categories meanwhile retain users' edit intentions.

3.5 Merging Conflicts of Multiple Versions

See the conference version and supplementary material Section 3, available online.

3.6 Historical Versions Maintenance and Deduplication

The merged version $V_{1,2}$ of a shared file, as well as the previous versions, should be kept in the cloud so that 1) users can retrieve any previous versions as they wish, and 2) the cloud can pinpoint V_0 from historical versions in future conflict resolutions. To save the storage space for hosting historical versions, we break each version into data chunks for deduplication. After that, (the indexes of) all versions and their data chunks are stored in a two-layer HashTable (cf. Fig. 15). The first layer of the HashTable is the file metadata that indicates chunks consisting of each version of a file, whereas the second layer indicates (objects of) data chucks.

To efficiently manage data chunks when they are modified, a simplified content-defined chunking mechanism is devised. It utilizes the precise positions of edits obtained from file update and conflict resolution to locate modified chunks; only chunks subject to changes need to conduct boundary shift. Specifically, once a new large file is uploaded to the cloud, it is first divided into equal-sized chunks (64 KB for each, with the last one may be smaller). The size of 64 KB is chosen, with the consideration of catering for the hybrid file update method (i.e., full-file sync for files with size $\leq$ 64 KB, cf.Section 3.7 for detail) and drawing on Microsoft's experience on the effectiveness of 64 KB-chunks-based deduplication [38]. Each chunk has a minimal and maximal size threshold (32 KB and 128 KB, respectively).

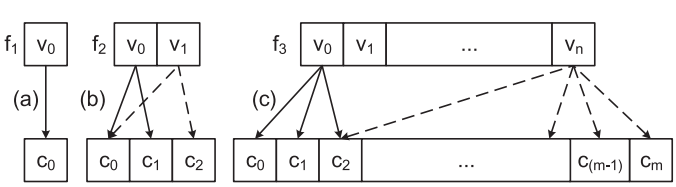


Fig. 15. Two-layer HashTable for hosting historical versions of a shared file. (a) A file has one version which consists of one chunk; (b) A file has two versions and they share a common chunk c_1. (c) A file has multiple versions and each of them consists of multiple chunks.

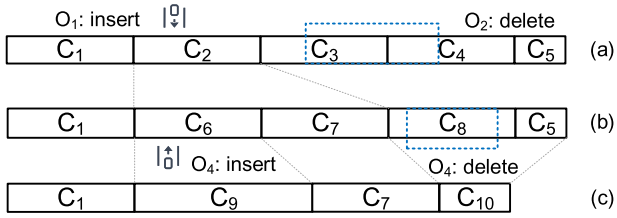


Fig. 16. A demonstration of boundary shift in the chunk-based deduplication scheme. (a) A file is consists of 5 (C_1 - C_5) chunks; afterward, some data are inserted in C_1 (by O_1) and some data crossing (the boundary of) C_3 and C_4 are deleted (by O_2). (b) C_2 is split to C_6 & C_7 and C_3 and C_4 is merged as C_8 after insert(or delete) as their size exceed (or fall below, respectivelly) the chunck size limitation. (c) Chunk C_6 becomes C_9 after insert, and C_8 is merged with C_5 after delete, forming chunk C_{10}.

Once a chunk exceeds the max size limitation due to data insertion, it will be split into multiple equal-sized chunks (i.e., $\frac{n}{\lfloor \frac{n}{64} \rfloor}$ KB for each[5]); once the size of a chunk becomes under the minimum-size limitation, it will be merged with the chunk adjacently behind it, and if the merged chunk exceeds the maximal size limitation, it will be split into two equal-sized chunks. For a delete operation covering several chunks, their remaining parts will be merged into one chunk as long as either part's size becomes under the minimal-size limitation; the merged chunk will be reorganized again if necessary (i.e., merging or re-balancing with an adjacent chunk, see Fig. 16 for a demonstration).

3.7 File Updates Delivery

Guided by the findings in Section 2.2, we utilize full-file sync for small ($\leq$64 KB) files and delta sync for larger files ($>$64KB), to achieve the (expected) shortest upload time. Especially, to quickly deliver updates for large files, we devise the LOADsync (Locality-aware, Operation-Assisted Delta sync), which utilizes the locality feature of uses' edits and implicit operations inferred during conflict resolution to accelerate the performance of `rsync`-like delta sync schemes.

Locality-Aware Uploading. As we have noticed that uses' edit features prominent locality, i.e., the edits are distributed in certain parts of a file (cf. Fig. 10), and unchanged blocks of the file maintain their original order, these features can be utilized to accelerate the computation-intensive 3-level chunk search in classic delta sync schemes. Specifically, when uploading a user's edits, the client first sends a pre-request to the server. The server latterly splits f (i.e., the server version) into fixed-size blocks, calculates fingerprints of each block (including both weak hash Adler32 and strong hash SHA-1) as a Checksum List, and sends it to the client (the same as what traditional `rsync` does). Second, on receiving the Checksum List, the client uses a byte-by-byte sliding window over f' to identify duplicated blocks; and a hash table, generated from the Checksum List, is used to

5. The notation $\lfloor x \rfloor$ represents the greatest integer less than or equal to x. This design makes each chunk have a size larger than 64 KB and meanwhile within the maximal size limitation after splitting.

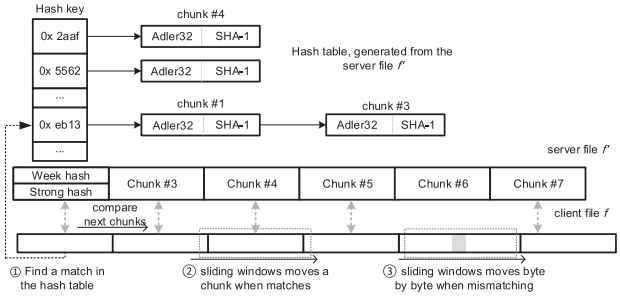


Fig. 17. The workflow of LOADsync in uploading edits.

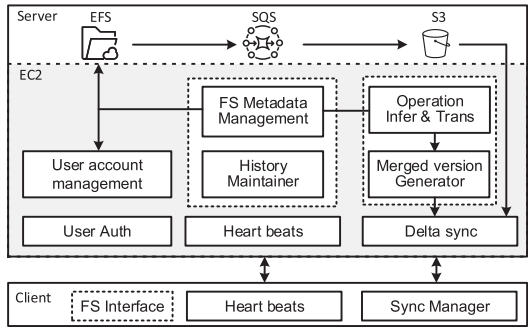


Fig. 18. System architecture and components of UCF2.

accelerate the matching procedures. In this stage, the locality pattern is utilized to bypass a considerable portion of unnecessary chunk search operations. Whenever a block (say ith) in f' matches a certain block (say jth) of f (step ① in Fig. 17), we directly move forward and only compare the strong hashes of their next blocks (i.e., $(i+1)$th block of f' and $(j+1)$th block of f), and if still, we move forward again (step ②); otherwise, we move the sliding window for one byte and return to the regular 3-level chunk searching scheme (step ③). Third, after the block comparison, the client obtains mismatched blocks (referred to as Delta Bytes) and sends them to the server, which afterward will use the Delta Bytes and server file f to re-construct the client file f'.

Distributing Merged Versions With Operation-Guided Fast Reconstitution. To deliver the server-merged version $V_{1,2}$ to clients, LOADsync employs *operation-guided fast reconstitution*, which leverages the implicit operations inferred during the conflict resolution to generate instructions that direct the client to fast constitute $V_{1,2}$ from its local version. In detail, to instruct Alice to generate $V_{1,2}$ from V_1, the server sends the operation sequence $S_{1\rightarrow1,2}$, which is generated by merging "reverse" operations of S_1 into the resulting sequence S_r, i.e., $S_{1\rightarrow1,2} = S_r + S_1^{-1}$. Note that operation merging in our modified OT is not *commutable*, and thus the implementations of the reversed merge for conflicting and non-conflicting operations are different. In essence, for non-conflicting operations $o_i \in S_1$ and $o_j \in S_2$, the result of the "reversed merge" of these two operations is simply preserving o_j in $S_{1\rightarrow1,2}$. However, in terms of conflicting operations in S_1 and S_2, the "reversed merge" is first to reverse each operation in S_1 (i.e., transform the "insert" to "delete", and transform the "delete" to "insert"), and then merge it with the operation that conflicts with it using OT, either for `delete/delete`, `delete/insert`, or `insert/insert` conflicts.

Reversed Sync for Extending Scalability. Besides, on syncing files when no conflicts happen, or syncing a merged version to collaborators that do not contribute to this conflict, we reverse the sync procedure by putting the computation-intensive checksum search into the client-side. In other words, we sacrifice a little extra computation overhead of the client for improving the throughput of our system.

4 Implementation and Evaluation

To implement our design, we build a prototype system UFC2 (User-Friendly Collaborative Cloud) on top of Amazon Web Services (AWS), as well as evaluate its performance under conventional and extreme environments.

4.1 Implementation

We build UFC2 on top of AWS with 5,000 lines of Python code. Our implementation is compatible with the general workflow and system architecture of existing cloud storage services (cf. Fig. 2). As shown in Fig. 18, at the infrastructure level, we host the (hierarchical) metadata of historical versions in Amazon EFS for efficient file system access, and the (flat) data chunks in Amazon S3 for economic content storage. We make such a design for an optimized balance between performance and efficiency [36], [37], [65], as the unit storage price of EFS ($\sim$\$0.3/GB/month) is around 10$\times$ higher than that of S3. Besides, the web service of UFC2 runs on a standard VM (with a dual-core CPU @2.5 GHz, 32-GB memory, and 256-GB SSD storage) rented from Amazon EC2. The employed EFS storage, S3 storage, and EC2 VM are located at the same data center in Northern Virginia and coordinated through Amazon Simple Queue Service (SQS), so there is no bottleneck among them.

The UFC2 client is almost identical to other mainstream cloud storage services. It periodically sends liveness beacon (heartbeats) to and receives sync notifications from the server, monitors the user's filesystem to detect file modification, and delivers user's edits using delta sync.

4.2 Experiment Setups and Evaluation Methodology

Performance Under Typical Workloads. Our experiment first aims to evaluate the effectiveness (i.e., whether conflicts are effectively and user-friendly resolved) and performance of UFC2 under normal collaboration circumstances. To achieve this, we deploy puppet collaborators at the client-side on geo-distributed VMs rented from Amazon EC2 to replay our collected ten real-world collaboration traces (cf. Table 2). Details of these VMs and the replay processes are the same as those described in Section 2.1.

Performance Under Extreme Workloads. Second, we hope to investigate its performance and throughput in extreme environments, including editing with large files and plenty of users simultaneously editing their files. These workloads, however, do not exist in our collected data traces, and seldom do they appear in real-world collaboration (or even text-editing) scenarios. Consequently, we dedicatedly select experimental materials and generate workloads.

TABLE 8
Measurement Statistics When the Ten Collaboration Traces are Replayed With UFC2

Trace	# File Versions	# Conflicting Versions	# MV Conflicts	# Conflicts	# True Conflicts	Reduction of Conflicts
Dropbox-1	3527	154	8	501	9	98.2%
Dropbox-2	2193	104	12	257	5	98.1%
OneDrive	2673	109	10	284	7	97.5%
iCloud Drive	3211	133	9	402	8	98.0%
Box	2930	125	5	374	8	97.9%
SugarSync	3472	147	13	523	11	97.9%
Seafile	2823	126	11	411	9	97.8%
Spark-Github	129957	1728	133	6724	167	97.5%
TensorFlow-Github	246016	8621	845	66231	1097	98.3%
Linux-Github	901167	36048	3210	216584	2882	98.7%

"MV Conflicts" denote the conflicts of multiple versions, i.e., ≥ 3 conflicting versions are generated from the same base version.

Specifically, to investigate UFC2's performance in dealing with large files, we utilize version iterations in genome sequencing of several species as experimental materials to simulate the scenarios of collaborative editing with write-write conflicts. Taking human (Homo sapiens) genome data as an example, the human genome consists of 23 pairs of chromosomes and the Human Genome Project has finished sequencing all of them [31]. In genome sequencing, a new version will formalize and be formally published to replace the former one, by filling its gaps and/or correcting its errors (which are inevitably generated as the limitations of current sequencing technologies). In our experiment, we use different versions of genome data of several species, including human (Homo sapiens), mouse (Mus musculus), rice (Oryza sativa), and fruit fly (Drosophila melanogaster).[6] The size of a single file (representing a genomic sequence of a single chromosome) ranges from 1 to 250-MB. As the genome files are sequentially published, to simulate the cases of collaborations with file-level conflicts, for every successive 3 versions, we regard the middle one as V_0 and the former and the latter version as V_1 and V_2, respectively.

To evaluate the scalability of UFC2, i.e., how many clients it supports to simultaneously work online, we deploy 30 VMs across the globe; each of them is configured with dual-core CPU@2.5 GHz, 8-GB DDR memory, and 1.5+ Gbps outgoing bandwidth. Besides, we deploy two physical PCs (a Dell T7920 with 2*Intel Xeon 4114 CPU and 160-GB DDR memory, and a Dell T5820 with Intel W-2245 CPU and 64-GB DDR memory) located at Beijing; there is no network limitation enforced on them (with 960-Mbps outgoing speed, tested by FastBTS [78]). We use light-weight scripts to simulate the client's editing behavior, and run multiple "clients" on each PC or VM. Both virtual and physical machines are used, to fit in with the distributed nature of cloud storage services and enable us to economically lunch much more clients simultaneously. Besides, we cut off all idle timespan to keep the server computation intensive.

4.3 Experiment Results

Ratio of Conflicts Resolved. Our first metric to evaluate the collaboration support of cloud storage services is the number of conflicts. We replay the ten traces with UFC2, and observe that the file versions generated by UFC2 (at the cloud side) are slightly different from those generated by Dropbox/OneDrive/iCloud Drive/Box/SugarSync/Seafile (cf. Section 2.2) due to the variation (*esp.*, in latency) of network environments; also, the resulting conflicts are slightly different. Notably, all the false conflicts are automatically resolved by UFC2. The remaining conflicts are all true conflicts that should be manually resolved by the collaborators, assisted with the helpful information automatically added by UFC2. As in Table 8, the ratio of conflicts is reduced by 97.5%–98.7% for different traces, i.e., an average reduction by 98%.

Time Overhead of Conflict Resolution. Conflict resolution in UFC2 consists of two steps: operation inference (OI, Section 3.3) and operation transformation (OT, Section 3.4). Thus, we first examine the time overhead incurred by the two steps separately, and then analyze the total time of conflict resolution (compared to the delivery time of a file update).

First, we record the time of OI in every conflict resolution when replaying the ten traces with UFC2. We also conduct stress testing with large source-code files (50–1000 KB) collected from GitHub, to test the performance of this most computation-intensive component under relatively intensive workloads. We compare our method with OI using the conventional edit graph method without clipping (which is also referred to as the Myers algorithm, and is the de facto standard delta algorithm adopted in the newest version of Git [61] and the early version of UFC2 [30]).

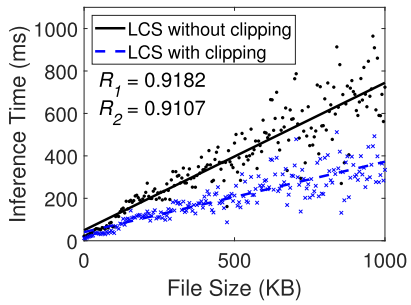


Fig. 19. Time overhead incurred by operation inference, OI with clipping *versus* using the naive Myers diff. Here R is the correlation coefficient between the measurements and linear fitting.

6. Available at https://www.ncbi.nlm.nih.gov/datasets/genomes/

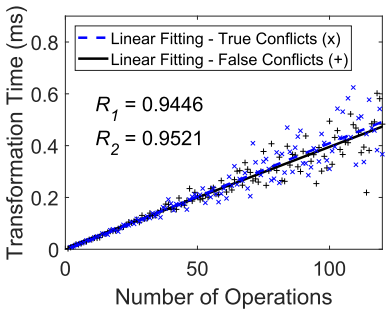


Fig. 20. Time overhead incurred by our devised operation transformation. Here R_1 and R_2 are the correlation coefficients with and without true conflicts.

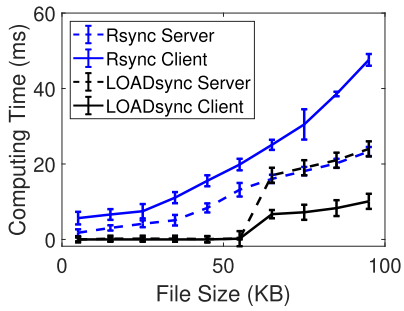


Fig. 22. Server and client-side computation time of LOADsync in uploading edits, compared with classical `rsync`.

The results are plotted as a scatter diagram shown in Fig. 19, together with a linear fitting. The correlation coefficient (R) between the measurements and the linear fitting results is up to 0.9107, indicating that the computation time of OI is generally proportional to the file size. This linearity mainly derives from the fact that, although both schemes have a superliner complexity (in terms of file length and number of characters changed), the modified parts of a file stay stable with the increment of the file size (rather than increasing proportionally with it). Second, we record the time of OT in every conflict resolution, and find it is very small (<1 ms) compared to the time of OI. As shown in Fig. 20, the time of OT is highly proportional to the number of operations; in addition, the performance is quite similar with or without true conflicts. According to Section 3.4, the complexity of our devised OT is $O(s_1 + s_2)$, which explains the experiment results. Further, we calculate the total time of a conflict resolution, and record the delivery time of the corresponding file update (using the hybrid full-file/delta sync method). As shown in Fig. 21, the total time of a conflict resolution is 10–80 ms, while the delivery time of a file update is 1.5–3 seconds. The former is merely 0.6%–3% (on average 2%) of the latter, showing that our conflict resolution brings negligible performance overhead to the collaboration in cloud storage.

Time Overhead of LOADsync versus Traditional `rsync`. We record the server and the client-side computation time in uploading and downloading updates. Meanwhile, we also compare the server and client-side computation overhead in uploading edits with that of traditional `rsync`. We do not compare their downloading performance, as the `rsync` is a symmetric scheme, and thus its computation overhead in downloading is largely similar to that of uploading.

As shown in Fig. 22, both LOADsync server and client consume a little time (0.3–2 ms) in uploading edit for files with size $\leq$64 KB. This is because LOADsync uses full-file sync to deliver edits for small files, and thus not much computation is required. For large files ($\geq$64 KB), the performances of LOADsync server and traditional `rsync` are almost the same, as they calculate the Checksum Lists with identical procedures. In terms of the `rsync` client, it consumes more time ($\sim 3\times$) than `rsync` server, as the client has more computation workload (e.g., checksum searching). In contrast, the LOADsync client consumes much less time than the LOADsync server (e.g., $\sim$10 ms for a 100KB file), as it utilizes the locality feature to boost the chunk search and comparison procedure (cf. Section 3.7 for detail).

For updates downloading, as shown in Fig. 23, when no file-level conflicts exist, UFC2 incurs little computation overhead in delivering small files ($\leq 64KB$). In terms of big files, we notice that the UFC2 server consumes much less computation overhead than the UFC2 client; this is because the LOADsync reverses the `rsync` procedure, leaving heavy workloads to the client for extending throughput. In downloading server-merged versions, i.e., when file-level conflicts exist, both UFC2 server and UFC2 client consume very short time (0.2 – 8 ms, 5 ms in average) in generating/executing the "reversed merge" operations (cf. Section 3.7).

Network Overhead. We compare the sync traffic of UFC2 with those of Dropbox, Google Drive, iCloud Drive, and Nutstore, for a file update. We only select the four cloud storage services since Dropbox, Google Drive, and iCloud Drive each represent a typical strategy for conflict resolution adopted by existing cloud storage services (i.e., keep all

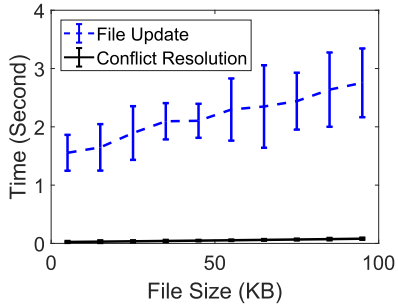


Fig. 21. Total time overhead of a conflict resolution *versus* the delivery time of a file update (using the hybrid full-file/delta sync method).

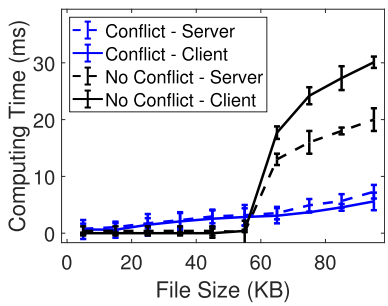


Fig. 23. The computation time of LOADsync when there are and there are not file-lever conflicts.

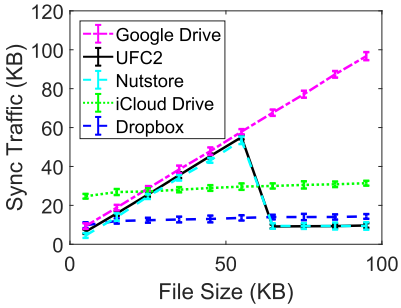


Fig. 24. Sync traffic of UFC2 and representative cloud storage services for a file update when there are no file-level conflicts.

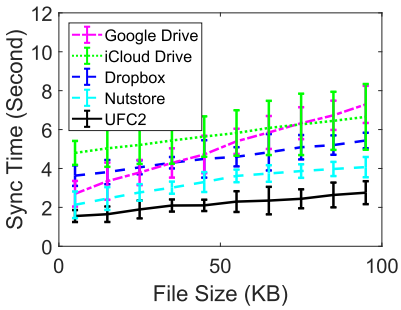


Fig. 25. Sync time of UFC2 and representative cloud storage services for a file update when there are no file-level conflicts.

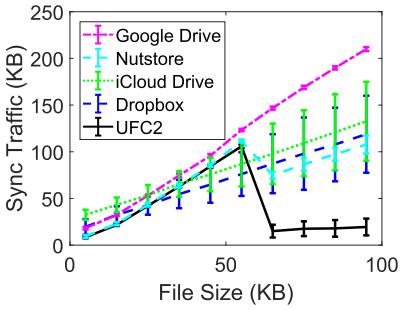


Fig. 26. Sync traffic of UFC2 and representative cloud storage services for a file update when there exist file-level conflicts.

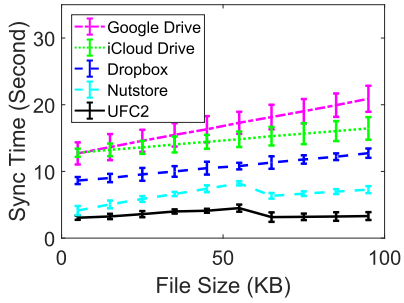


Fig. 27. Sync time of UFC2 and representative cloud storage services for a file update when there exist file-level conflicts.

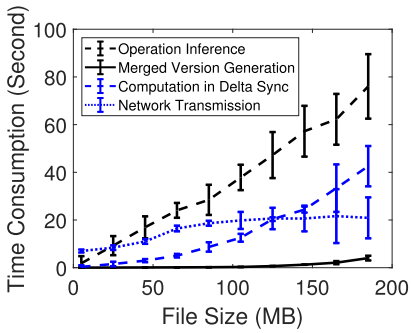


Fig. 28. Time consumption of UFC2 in dealing with large files; "Merged Version Generation" denotes the process of OT and executing S_r on V_0 to generate $V_{1,2}$.

conflicting versions, only keep the latest version, and force users to choose one version, cf. Section 2.2) while Nutstore is the only service that combines full-file sync and delta sync to enhance the file update speed.

As shown in Fig. 24, when there are no file-level conflicts, the sync traffic of Google Drive is close to the file size, as Google Drive adopts full-file sync. In contrast, Dropbox and iCloud Drive always consume nearly 10 KB and 30 KB of sync traffic respectively due to their adoption of delta sync; we infer that the sync granularity of Dropbox is finer than that of iCloud Drive. In contrast, Nutstore and UFC2 resemble Google Drive for small ($\leq$64 KB) files and Dropbox for larger files, as they both adopt full-file sync for small files and delta sync for larger files to achieve the shortest sync time (see Fig. 25). This hybrid sync method results in substantial savings of sync traffic for Nutstore and UFC2 after the turning point (64 KB) in Figs. 24 and 26.

As shown in Fig. 26, when there exist file-level conflicts, the sync traffic of Google Drive is nearly twice of the file size. This is because (the client of) Google Drive first uploads the local version, and then downloads the cloud-hosted newer version to overwrite the local version. In contrast, the sync traffic consumed by Dropbox or iCloud Drive is close to the file size; this is because the client of Dropbox (or iCloud Drive) renames one of the conflicting versions, and the renamed one is uploaded as a newly-created file using full-file sync (which usually consumes more traffic than necessary since delta sync can still be applied).

The case of Nutstore in Fig. 26 is a bit complex: for small files, its sync traffic is nearly twice of the file size (similar to Google Drive); for larger files, the traffic is slightly larger than the file size (similar to Dropbox/iCloud Drive). This is because Nutstore renames one of the conflicting versions when a file-level conflict occurs – if the file is small ($\leq$64 KB), the two files are both uploaded to the cloud using full-file sync; otherwise, the renamed file is uploaded using full-file sync (which usually consumes unnecessary traffic) whereas the original file is uploaded using delta sync.

Finally, we examine the case of UFC2 in Fig. 26. Its client first uploads a conflicting version and then downloads the merged version from the cloud. For a small file, the two versions are both delivered using full-file sync, so the sync traffic is nearly twice of the file size; for a larger file, the two versions are both delivered using delta sync (which is more traffic-saving than what Nutstore does for a larger file), so the sync traffic is always as small as $\sim$20 KB. This is why UFC2 achieves the shortest sync time, as shown in Fig. 27.

Performance Under Large Files. We record the time consumed in delivering updates (including computation in delta sync, and network transmission) and resolving conflicts (including OI, OT, and exerting the merged operation

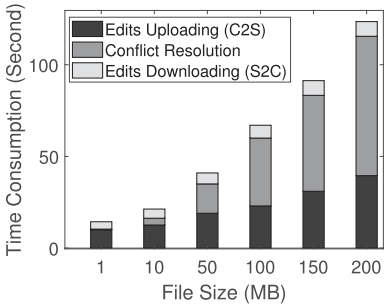


Fig. 29. Total time consumption in a full lifecycle of delivering edits and solving conflicts, with file size from 1 to 200 MB. "C2S" means "(from) client to server".

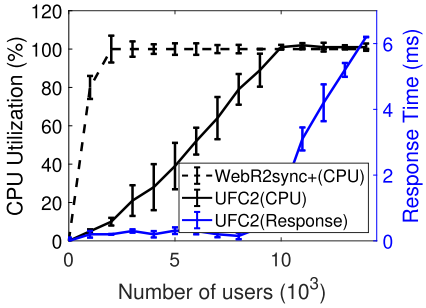


Fig. 30. Capacity in holding multiple users under intensive workloads, comparing with WebR2sync+.

sequence on the base version to generate the final version) in dealing files of size ranging from 1-MB to 200-MB. It can be seen that the time overhead of OI increases sharply along with the growth of file-size (cf. Fig. 28), and accounts for the largest parts among the full collaboration procedure when file-size is larger than 100-MB (cf. Fig. 29). The huge computing overhead derives from the intrinsic complexity of the LCS problem. Recall that the full procedure in collaboration includes uploading edits to the cloud, solving conflicts, and delivering conflict resolution result to clients (Fig. 29). For other steps in conflict resolution (combined as "final version generation" in Fig. 29), they take a very few parts in the full procedure ($\leq$ 3 second).

Different from the small-file settings, *data delivery*-related overhead is small than OI for big files. Specifically, although the computation time in delta sync also increases nearly linearly with file size, it grows much more gently than OI. Counter-intuitively, network transmission time overhead first increases gradually (for files $\leq$ 100-MB) but then stays stable. We infer that this is perhaps because network transmission time is mainly affected by the size of data transmitted, which, in the genome dataset, does not grow proportionally with the file-size. For example, the relatively large files ($\leq$ 130-MB) normally belong to the human genomes which are updated with a high frequency; thus each version has fewer modifications. On the contrary, files with smaller sizes may belong to other species – they update at a long time interval but each update includes more modifications.

Throughput. We also investigate the service throughput of UFC2, i.e., the number of concurrent clients it can support. For an intuitive perception, we also compare UFC2 with WebR2sync+ [77], a highly efficient and scalable cloud storage service for web users. We find that the server CPU of WebR2sync+ is fully occupied (i.e., CPU utilization on all cores approaching 100%) when it serves $\sim$1.5K users, indicating its the user capacity is $\sim$1.5K under intensive workloads. By construct, UFC2 has a 6$\times$ throughput of WebR2sync+, supporting $\sim$9K users (Fig. 30). Note that it is the user capacity under *a single VM server instance*; for real-world services with more concurrent clients, it is feasible to cater to such scenarios by deploying more server instances.

5 Limitation, Extension, and Discussion

The design of UFC2 is mainly focused on collaboration with pure text files. In this section, we discuss the limitations of our approach, and examine the possibility in extending this method to broader scenarios.

Dealing With Binary Files. This work provides user-friendly collaboration on text files but does not automatically solve conflicts for binary files (e.g., multimedia files and executable files). This simplification is reasonable as text-oriented collaboration covers the overwhelming majority of collaboration scenarios, and binary files are usually generated as the output of text files processed or compiled by specific programs. Nevertheless, there are also application requirements to cooperatively work on none-text files. Theoretically, with OT paradigms, it is possible to construct collaborative platforms for other applications, as long as we can precisely capture the user's editing operations and replay them on the server. Thus, it is feasible to build dedicated collaborative editors for vector-based tasks (e.g., graphics modeling and CAD [23], [53]), as customized software can directly catch users' operations which can be efficiently replayed.[7] But for cloud storage services, acquiring edit instructions for all kinds of applications is almost impossible. To the best of our knowledge, Arnold [35] is the only academic prototype that can record user's instructions (or equivalent operational effects) for any applications. However, it relies on a modified Linux kernel to record the execution of OS processes. This technique is definitely not applicable to real-world cloud storage systems.

Dealing With Formatted Text. See Section 6 of the supplementary material for details, available online.

6 Related Work

Various schemes (and techniques) have been proposed to address the collaboration conflicts in distributed file systems (DFS), version control systems (VCSes), and real-time editors. We briefly survey conflict resolution in VCS in this part; refer to the supplementary material, available online, to see a review of other related topics.

Conflict Resolution in VCSes. Finding and handling conflicts is an important task in version control systems. Popular VCSes, such as SVN, CVS, Git, Subversion [63], RCS [72] and SunPro [22] adopt a *unstructured merge* approach. They use delta algorithms like `bdiff` [71] and Myers's diff [58] to find differences at a line granularity. The changed lines will be marked as `Insert` and `Delete` actions, and will then be

7. Note that in such tasks, data are represented as vectors, and operations are in essence mathematical calculations on them.

combined to form a merged version. However, if two users' modifications are made on the same or adjacent lines, the users are forced to manually solve the conflict (e.g., choose which line to retain).

To address its limitations of insufficient semantics and coarse-grained granularity, some advanced programming tools adopt the *structured merge* [25], [26], [49], [82] approach, which takes the syntactic structure of programs into account and hopes to provide a fine-grained collaboration support for programmers. These systems regard the programs as a abstract syntax trees (AST) [59], or similar structures such as graphs [43], [56], so as to incorporate all kinds of information on the underlying programming language. With such structures, it is possible to resolve concurrent edits at a syntax level and skip conflicts at non-essential parts (e.g., blanks, tabs and comments) of the program. Nevertheless, the *structured merge* approach is language-specific, which means one scheme can only support collaboration for one specific language, e.g., `JDime` [49] and `JDiff` [26] only work for JAVA programs. In this work, we focus on conflict resolution in cloud storage-based collaborations with general purpose, in which line-level granularity is too coarse whereas no syntax structures can be utilized.

7 Conclusion

Despite a rich body of techniques for resolving conflicts in collaborative systems, today's mainstream cloud storage services still use the simplest form, i.e., coarse-grained file-level conflict detection and resolution. Given that collaboration has become a major use case of cloud storage services, existing mechanisms, as revealed in this paper, are deficient, inconvenient, and sometimes frustrating.

To address the issue, we make a series of efforts towards understanding and improving collaboration in cloud storage services from a novel perspective of operations without using any locks. We find that the vast majority of conflicts reported by today's cloud storage services are false conflicts, and design intelligent approaches to efficient operation inference, user-friendly operation transformation, and judicious maintenance of historical versions. We implement all the approaches in an open-source prototype system that can significantly reduce collaboration conflicts and meanwhile preserve the transparency and user-friendliness of cloud storage services.

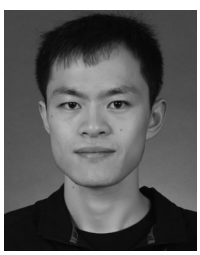
Acknowledgments

We thank the reviewers for their valuable comments. We appreciate Xianlong Wang and Wei Tang for their discussions and technical supports.

References

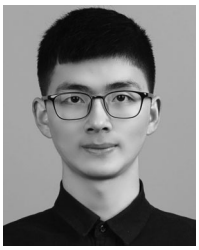
[1] Average U.S. Internet Speeds More Than Double Global Average. [Online]. Available: https://www.ncta.com/whats-new/average-us-internet-speeds-more-double-global-average
[2] Box – Secure File Sharing, Storage, and Collaboration. [Online]. Available: https://www.box.com/
[3] Charles Web Debugging Proxy. [Online]. Available: https://www.charlesproxy.com
[4] Dropbox Tech Blog. [Online]. Available: https://dropbox.tech/
[5] The Etherpad Foundation, "Etherpad: Really real-time collaborative document editing," [Online]. Available: https://github.com/ether/etherpad-lite
[6] Google Docs: Free Online Documents for Personal Use. [Online]. Available: https://www.google.com/docs/about/
[7] D. Kopytkov and P. Lee, "Meet bandaid, the dropbox service proxy", 2018. [Online]. Available: https://blogs.dropbox.com/tech/2018/03/meet-bandaid-the-dropbox-service-proxy/
[8] Nutstore – Share Your Files Anytime, Anywhere, with Any Device. [Online]. Available: https://www.jianguoyun.com/
[9] Nutstore Help Center. [Online]. Available: http://help.jianguoyun.com/
[10] A. Ivanov, "Optimizing web servers for high throughput and low latency," 2017. [Online]. Available: https://blogs.dropbox.com/tech/2017/09/optimizing-web-servers-for-high-throughput-and-low-latency/
[11] Overleaf, Online LaTeX Editor. [Online]. Available: https://www.overleaf.com/
[12] Seafile - Open Source File Sync and Share Software. [Online]. Available: https://www.seafile.com/en/home/
[13] Seafile Source Code. [Online]. Available: https://github.com/haiwen/seafile
[14] Simultaneous Collaborative Editing of a LaTeX File (Online Forum Discussion). [Online]. Available: https://tex.stackexchange.com/questions/27549/simultaneous-collaborative-editing-of-a-latex-file
[15] Speedtest Global Index – Global Speeds August 2019. [Online]. Available: https://www.speedtest.net/global-index
[16] C. Smith, "Staggering dropbox statistics and facts (2021)," 2021. [Online]. Available: https://expandedramblings.com/index.php/dropbox-statistics/
[17] SugarSync – Cloud File Sharing, File Sync & Online Backup From Any Device. [Online]. Available: https://www2.sugarsync.com/
[18] SugarSync Help Center. [Online]. Available: https://support.sugarsync.com/hc
[19] The SugarSync Blog. [Online]. Available: https://www.sugarsync.com/blog/
[20] B. Malmskog, "Tool for the (collaborative) job," [Online]. Available: https://blogs.ams.org/phdplus/2016/09/11/tool-for-the-collaborative-job/
[21] Wireshark. [Online]. Available: http://www.wireshark.org
[22] E. Adams, W. Gramlich, S. S. Muchnick, and S. Tirfing, "SunPro: Engineering a pratical program development environment," in *Proc. Int. Workshop Adv. Program. Environ.*, 1986, pp. 86–96.
[23] Agustina and C. Sun, "Dependency-conflict detection in real-time collaborative 3D design systems," in *Proc. ACM Conf. Comput. Supported Cooperative Work*, 2013, pp. 715–728.
[24] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *Proc. ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2017, pp. 58–67.
[25] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: Balancing precision and performance," in *Proc. 27th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2012, pp. 120–129.
[26] T. Apiwattanapong, A. Orso, and M. J. Harrold, "JDiff: A differencing technique and tool for object-oriented programs," *Automated Softw. Eng.*, vol. 14, pp. 3–36, 2007.
[27] R. Bhargava, "Evolution of Dropbox's edge network, 2017. [Online]. Available: https://blogs.dropbox.com/tech/2017/06/evolution-of-dropboxs-edge-network/
[28] C. Brindescu, I. Ahmed, R. Leano, and A. Sarma, "Planning for untangling: Predicting the difficulty of merge conflicts," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 801–811.
[29] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An enhanced line differencing tool," in *Proc. ACM/IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 595–598.
[30] J. Chen *et al.*, "Lock-free collaboration support for cloud storage services with operation inference and transformation," in *Proc. 18th USENIX Conf. File Storage Technologies*, 2020, pp. 13–28.
[31] H. G. S. Consortium *et al.*, "Finishing the euchromatic sequence of the human genome," *Nature*, vol. 431, pp. 931–945, 2004.
[32] Y. Cui *et al.*, "TailCutter: Wisely cutting tail latency in cloud CDNs under cost constraints," *IEEE/ACM Trans. Netw.*, vol. 27, no. 4, pp. 1612–1628, Aug. 2019.
[33] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, pp. 74–80, 2013.
[34] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Commun. ACM*, vol. 61, pp. 65–72, 2018.

[35] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proc. 11th USENIX Conf. Oper. Syst. Des. Implementation*, 2014, pp. 525–540.
[36] J. E, Y. Cui, Z. Li, M. Ruan, and E. Zhai, "HyCloud: Tweaking hybrid cloud storage services for cost-efficient filesystem hosting," *IEEE/ACM Trans. Netw.*, vol. 28, no. 6, pp. 2629–2642, Dec. 2020.
[37] J. E, Y. Cui, M. Ruan, Z. Li, and E. Zhai, "HyCloud: Tweaking hybrid cloud storage services for cost-efficient filesystem hosting," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 2341–2349.
[38] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, Art. no. 26.
[39] C. Ellis and S. Gibbs, "Concurrency control in groupware systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1989, pp. 399–407.
[40] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation*, 2010, pp. 337–350.
[41] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 285–296.
[42] M. P. Grosvenor *et al.*, "Queues don't matter when you can JUMP them!," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 1–14.
[43] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proc. IEEE Int. Conf. Softw. Eng.*, 1992, pp. 392–411.
[44] B. Hou and F. Chen, "GDS-LC: A latency-and cost-aware client caching scheme for cloud storage," *ACM Trans. Storage*, vol. 13, 2017, Art. no. 40.
[45] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta algorithms: An empirical analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, pp. 192–214, 1998.
[46] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, pp. 558–562, 1962.
[47] A. Kumar, I. Narayanan, T. Zhu, and A. Sivasubramaniam, "The fast and the frugal: Tail latency aware provisioning for coping with load variations," in *Proc. ACM Web Conf.*, 2020, pp. 314–326.
[48] H. Le Nguyen and C.-L. Ignat, "Parallelism and conflicting changes in Git version control systems," in *Proc. 15th Int. Workshop Collaborative Editing Syst.*, 2017. [Online]. Available: https://hal.inria.fr/hal-01588482/
[49] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 543–553.
[50] Z. Li *et al.*, "Towards network-level efficiency for cloud storage services," in *Proc. Conf. Internet Meas. Conf.*, 2014, pp. 115–128.
[51] G. Liang and U. C. Kozat, "FAST CLOUD: Pushing the envelope on delay performance of cloud storage with coding," *IEEE/ACM Trans. Netw.*, vol. 22, no. 6, pp. 2012–2025, Dec. 2014.
[52] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. 7th Conf. File Storage Technologies*, 2009, pp. 111–123.
[53] X. Lv, F. He, Y. Cheng, and Y. Wu, "A novel CRDT-based synchronization method for real-time collaborative CAD systems," *Adv. Eng. Informat.*, vol. 38, pp. 381–391, 2018.
[54] D. Marx, "Graph colouring problems and their applications in scheduling," *Periodica Polytechnica Elect. Eng.*, vol. 48, pp. 11–16, 2004.
[55] G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek, "On the nature of merge conflicts: A study of 2,731 open source Java projects hosted by GitHub," *IEEE Trans. Softw. Eng.*, vol. 46, no. 8, pp. 892–915, Aug. 2020.
[56] T. Mens, "Conditional graph rewriting as a domain-independent formalism for software evolution," in *Proc. Int. Workshop Appl. Graph Transformations Ind. Relevance*, 1999, pp. 127–143.
[57] T. Mens, "A state-of-the-art survey on software merging," *IEEE Trans. Softw. Eng.*, vol. 28, no. 5, pp. 449–462, May 2002.
[58] E. W. Myers, "An *O(ND)* difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.
[59] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, 2005.
[60] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, "The life-cycle of merge conflicts: Processes, barriers, and strategies," *Empir. Softw. Eng.*, vol. 24, pp. 2863–2906, 2019.
[61] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in Git?," *Empir. Softw. Eng.*, vol. 25, pp. 790–823, 2020.
[62] S. Perez De Rosso and D. Jackson, "What's wrong with Git?: A conceptual design analysis," in *Proc. ACM Int. Symp. New Ideas New Paradigms Reflections Program. Softw.*, 2013, pp. 37–52.
[63] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version Vontrol With Subversion: Next Generation Open Source Version Control*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2008.
[64] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2004, Art. no. 6.
[65] K. P. Puttaswamy, T. Nandagopal, and M. Kodialam, "Frugal storage for cloud file systems," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 71–84.
[66] B. Shao, D. Li, T. Lu, and N. Gu, "An operational transformation based synchronization protocol for web 2.0 applications," in *Proc. ACM Conf. Comput. Supported Cooperative Work*, 2011, pp. 563–572.
[67] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *Proc. ACM Program. Lang.*, vol. 2, 2018, Art. no. 165.
[68] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proc. 10th USENIX Conf. File Storage Technologies*, 2012, Art. no. 24.
[69] Y. Su, D. Feng, Y. Hua, and Z. Shi, "Understanding the latency distribution of cloud object storage systems," *J. Parallel Distrib. Comput.*, vol. 128, pp. 71–83, 2019.
[70] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 513–527.
[71] W. F. Tichy, "The string-to-string correction problem with block moves," *ACM Trans. Comput. Syst.*, vol. 2, pp. 309–321, 1984.
[72] W. F. Tichy, "RCS – A system for version control," *Softw.: Practice Experience*, vol. 15, pp. 637–654, 1985.
[73] A. Tridgell and P. Mackerras, "The Rsync algorithm," Technical report, 1996. [Online]. Available: https://openresearch-repository.anu.edu.au/bitstream/1885/40765/3/TR-CS-96–05.pdf
[74] Z. Wu, C. Yu, and H. V. Madhyastha, "Costlo: Cost-effective redundancy for lower latency variance on cloud storage services," in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 543–557.
[75] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.
[76] W. Xia, H. Jiang, D. Feng, and Y. Hua, "SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 26–28.
[77] H. Xiao *et al.*, "Towards web-based delta synchronization for cloud storage services," in *Proc. 16th USENIX Conf. File Storage Technologies*, 2018, pp. 155–168.
[78] X. Yang *et al.*, "Fast and light bandwidth testing for internet users," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 1011–1026.
[79] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 456–468.
[80] X. Zhou, J. Yuan, P. Shilane, W. Xia, and X. Wang, "The Dilemma between deduplication and locality: Can both be achieved?," in *Proc. 19th USENIX Conf. File Storage Technologies*, 2021, pp. 171–185.
[81] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technologies*, 2008, Art. no. 18.
[82] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proc. ACM Program. Lang.*, vol. 2, 2018, Art. no. 166.



Minghao Zhao (Student Member, IEEE) received the BS degree from Harbin Engineering University, Harbin, China, in 2014, and the MS degree from Shandong University, Jinan, China, in 2017. He is currently working toward the PhD degree in the School of Software, Tsinghua University, Beijing, China. His research interests include cloud computing/storage, operating system, and software engineering. He is a student member of the ACM and CCF.

Zhenhua Li (Senior Member, IEEE) received the BS and MS degrees from Nanjing University, Nanjing, China, in 2005 and 2008, respectively, and the PhD degree from Peking University, Beijing, China, in 2013, all in computer science and technology. He is currently an associate professor with the School of Software, Tsinghua University. His research interests include mobile networking/emulation and cloud computing/storage. He is a senior member of the ACM.



Wei Liu received the BS degree from Jilin University, Changchun, China, in 2020. He is currently working toward the PhD degree in the School of Software, Tsinghua University, Beijing, China. His research interests include, but are not limited to video streaming applications and operating systems.

Jian Chen received the BS and MS degrees from Tsinghua University, Beijing, China. He is currently a staff engineer with ByteDance Inc. His research interests include cloud computing, cloud storage, and data center networking.

Xingyao Li received the BS degree from the School of Software, Tsinghua University, Beijing, China, where he is currently working toward the MEng degree. His research interests include cloud computing and future networking.