Theory of Mergeability: Safely Merging Two Versions of a Document

Ka Cheung Sia  
UCLA  
4732 Boelter Hall  
Los Angeles, CA 90095  
kcsia@cs.ucla.edu

Susan Chebotariov  
UCLA  
4732 Boelter Hall  
Los Angeles, CA 90095  
susanc@cs.ucla.edu

Junghoo Cho  
UCLA  
4732 Boelter Hall  
Los Angeles, CA 90095  
cho@cs.ucla.edu

ABSTRACT
As technology becomes more readily available to the public, individuals increasingly have access to multiple machines (e.g., home desktop, office workstation, portable laptop, etc.) frequently moving from one machine to another while working on a particular task. As a result, maintaining and synchronizing different versions of documents on multiple machines is becoming an important problem. In this paper, we propose a theoretical framework to deal with this problem by first introducing reasonable definitions of when two simultaneous changes to the same document can be considered “safely mergeable.” We then propose new and efficient algorithms to guarantee that only “safely mergeable changes” are made to the document and prove the correctness of the proposed algorithms. Finally, we evaluate the performance of our algorithms by investigating their storage and computational costs using real data.

1. INTRODUCTION
As computer prices decrease, individuals are able to own and use multiple machines (e.g., home desktop, office workstation, portable laptop, etc.) and often move from one machine to another while working on a particular task. Remote, online collaboration is also gaining popularity as the Internet becomes widespread and people on different continents work together on the same project. In this environment, maintaining and synchronizing multiple, simultaneous changes to the same set of files (or documents) is a very important issue.

Two approaches are widely being used in this context with their own pros and cons as can be illustrated by two standard e-mail protocols: POP3 and IMAP. Under POP3, every machine (or client) stores its own local copy of e-mails downloaded from a central server. Since the downloaded e-mails are accessible offline, this setup is particularly useful when the user does not have constant network connectivity. On the other hand, finding and organizing e-mails distributed over multiple machines can be a significant challenge and requires careful coordination by the user.

The IMAP protocol tries to alleviate the distributed e-mail management problem by letting the user manage all e-mails at a central server. Under this setup, the user always sees the same set of e-mails from the server regardless of the client machine, so e-mail synchronization is not necessary. On the other hand, constant network connectivity is essential to access e-mails, so it is not suitable, say, when the user is traveling and has only intermittent network connectivity.

Our work has been mainly inspired by our desire to get the best of both worlds, where the system will automatically handle the synchronization of files (or documents) on multiple machines while the user can still access all her documents offline. In Figure 1, we show one possible architecture to achieve this goal, where all documents are stored at a central server, but the documents are also cached locally at each client machine. Under this setting, when the user makes changes to the documents from a particular client, if the client is online, the changes are propagated immediately to the server. Otherwise, the changes are made only to the cached version temporarily and are propagated to the server later when the client goes online. Therefore, the user can still work on her documents offline and does not have to constantly worry about manually synchronizing her documents.

A number of existing tools implement an architecture similar to the one above [1, 2, 3], but one important challenge that needs further attention is how the system can “safely
merge" concurrent changes made to the same document by multiple client machines. When changes are not propagated immediately to each other, multiple clients may work on the same set of documents concurrently without being aware of the changes made by others. In this case, how should these changes be merged into a single final document? When are these concurrent changes “safe to merge” and lead to a “consistent” final document? How do we ensure that only “safe” changes are allowed to be merged?

Inspired by the serializability theory in database literature that provides a firm foundation regarding when concurrent transactions can be allowed without causing any inconsistency [4], in this paper we develop a theory of mergeability that clarifies when simultaneous changes to the same document can be considered “safe” and can be merged into the final document. Based on this theory, we then develop efficient mechanisms that guarantee only merging safe changes into the document, ensuring that the user does not see an unexpected result. In particular, we make the following contributions in this paper:

- We develop a theory of mergeability that clarifies when concurrent changes to a document can be considered “safe” and can be merged together without causing “inconsistency.”

- Based on the developed theory, we then design efficient algorithms that guarantee only merging safe changes into the final document, rejecting any changes that may cause inconsistency. We provide key theorems showing the correctness of our proposed algorithms under our mergeability theory. While there are a number of tools that try to help users in this context, as far as we know, our work is the first attempt to develop a formal definition of safe changes to a document and design algorithms based on this definition.

- Using real document change history data, we evaluate the effectiveness of our algorithms in terms of the storage and computational costs and show that our algorithms are efficient and can be implemented easily in practical settings without much overhead.

The rest of this paper is organized as follows:

- In Section 2, we develop a theory of mergeability, which can be applied to the question of safely mergeable changes free from any assumptions about implementation details. We introduce the problems that need to be addressed, discuss in what cases changes are safely mergeable, and explain the concepts that support our definition of strict mergeability, which is the heart of our mergeability theory.

- In Section 3, we introduce our suggestions for efficiently checking mergeability by defining conflicting operations and introducing our representation of documents using globally unique identifiers (GUIDs). We formalize several algorithms for checking mergeability and address some questions which arise related to tracking the changes made to versions of a document.

- In Section 4, we evaluate the storage and computational costs required by our method of mergeability checking and analyze our method as compared with real CVS repositories.

- In Section 5, we distinguish our approach from the extensive research that has already been done in the area of concurrency control and serializability theory, explaining how our work builds on existing research and deviates from it making our contributions valuable in the area of addressing the theory behind mergeable documents.

2. MERGEABILITY THEORY

In this section, we first describe the basis for our mergeability theory. Then, we define two types of mergeability that can be applied to the scenario of one or more users desiring to merge a set of changes between two versions of the same document. In the subsequent section we address methods that can be used to check the mergeability of two versions and to enforce certain constraints.

2.1 Mergeable changes

Consider the examples in Figures 2 and 3. Two users, u1 and u2, start working on the same version, v1, of a document. When u1 and u2 have completed their changes, we want to determine if the changes are safely mergeable and, if so, combine them into a new, consistent version, v2, of the document.

In this context, one important topic that must be addressed is what we mean by “safely mergeable” changes that lead to a “consistent” final document. Intuitively, for the example of Figure 2, we may say that u1’s change (inserting \(xy\) at the beginning of the document) and u2’s change (deleting \(c\) from the end of the document) are safe to be merged into the new document because, even though the two changes were made concurrently, it does not really matter which user’s changes were made first; whether u1’s changes are applied to the document before or after u2’s changes, the final document is always \(xyab\). That is, users will always
get the same final document regardless of “who worked on it first.”

In contrast, for the example of Figure 3, we may say that the two changes from \( u_1 \) and \( u_2 \) (inserting \( xy \) at the beginning of the document by \( u_1 \) and inserting \( st \) at the beginning of the document by \( u_2 \)) are “incompatible” because, depending on whose changes are applied first, the final document can be different – \( stxyabc \) if \( u_1 \)'s changes are made first and \( xystabc \) otherwise; since the order in which the changes are made creates ambiguity in what the final document should be, we may consider this case “unsafe” and ask the users to resolve the ambiguity manually.

The above example suggests one possible interpretation of safely mergeable changes: we consider concurrent changes safe to merge if and only if we get the same final document regardless of the order in which the changes are made. The main goal of the rest of this section is to formalize this interpretation as strict mergeability.

### 2.2 Basic operations

In order to formally define strict mergeability, we start with a discussion of the basic operations on a document. In Figure 4, we show a document-editing example where a user makes two changes to the initial version, \( v_1 \), of a document and it evolves into a newer version, \( v_2 \). Note that the operations in this example are significantly different from the common operations used in serializability theory [4]. In serializability theory, the basic operations are \( \text{read}(V) \) and \( \text{write}(V) \) on an atomic variable \( V \), which are assumed to atomically read and write the entire contents of variable \( V \). In contrast, document-editing operations are much more granular; users often make insertions and deletions on parts of the document. To capture this difference better, in this paper, we model a document as a sequence of bytes (as opposed to an atomic variable) and consider the basic editing operations to be insertions and deletions on the byte sequence. In particular, we use the following notation to denote the two operations:

- **insert**: \( \text{insert}(\text{document version}, \text{insertion point}, \text{new bytes}) \)
- **delete**: \( \text{delete}(\text{document version}, \text{deletion range}) \)

Here, the meaning of the parameters are as follows:

- **document version**: refers to the initial byte stream before making any changes (\( v_1 \) in the example in Figure 4).
- **insertion point\(^1\)**: refers to the location where the bytes are inserted (between \( b \) and \( c \) in this example). In this paper, we represent an insertion point by indicating the two adjacent byte locations between which the new byte sequence is inserted. For example, the insertion point can be represented as \([2,3]\) in Figure 4.
- **new bytes**: refers to the bytes being inserted (\( j \) in this example).
- **deletion range**: refers to a consecutive sequence of bytes that are being deleted (the fourth to sixth bytes, \( def \), in this example, which we represent as \([4,6]\))

Under this notation, the insertion of \( j \) in Figure 4 will be denoted by \( \text{insert}(v_1, [2,3], j) \) and the deletion of \( def \) in the figure will be denoted by \( \text{delete}(v_1, [4,6]) \).

### 2.3 Equivalence

As we stated before, the final goal of this section is to formally define the notion of strict mergeability, which considers every possible order of concurrent operations and verifies whether we get the same final document or not. When different orders of operations are considered, we note that the exact form of an operation may be syntactically different depending on the order. For example, consider the scenario of Figure 2. In this case, if we assume \( u_2 \)'s change is applied before \( u_1 \)'s, then \( u_2 \)'s deletion will be represented as \( \text{delete}(v_1, [2,2]) \), while if \( u_2 \)'s change is applied after \( u_1 \)'s, then it will be represented as \( \text{delete}(v_1', [5,5]) \). From this example, we see that while they are not syntactically identical, the deletion range \([2,2]\) of version \( v_1 \) is “equivalent” to the deletion range \([5,5]\) of version \( v_1' \). Similarly, we see that the two operations, \( \text{delete}(v_1, [2,2]) \) and \( \text{delete}(v_1', [5,2]) \), are “equivalent” while operations are performed on two different versions of the document. We formalize this notion of equivalence as follows:

\(^1\)We use the locations \([0] \) and \([-1]\) to reference the beginning and the end of the document respectively. For example, to insert a byte at the beginning of the document use the insertion point \([0, 1]\). Similarly, to insert a byte at the end of the document use the point \([n, -1]\), where \( n \) is the number of bytes in the document.
• **Equivalent location:** When a byte stream is edited by insertions or deletions, bytes can be created, deleted, or retained. Two locations in different document versions are considered equivalent if they retain the same byte from the initial byte stream. For example, in Figure 4, the location of the third byte, c, in v1 ([3] in v1) is retained and considered to be equivalent to the fourth byte in v2 ([4] in v2). However, the [4] in v1 does not have an equivalent location in v2, because that byte was deleted.

• **Equivalent edit locations:**
  
  - **Equivalent insertion point:** Two insertion points are equivalent if both of the locations are equivalent and adjacent. For example, in Figure 4, the insertion point [7,8] in v1 (between g and h) is equivalent to [5,6] in v2, because [7] and [8] in v1 are equivalent to [5] and [6] in v2, respectively and and [5] and [6] are adjacent in v2. However, although location [2] and [3] in v1 have equivalent locations in v2 ([2] and [4], respectively), since those locations are not adjacent in v2, the insertion point [2,3] in v1 does not have an equivalent insertion point in v2.
  
  - **Equivalent deletion range:** Two deletion ranges are considered to be equivalent if all of their enclosed locations are equivalent and consecutive. For example, although locations in the deletion range [2,3] of v1 find their equivalent locations in the range [2,4] of v2, they are not equivalent because the locations are no longer consecutive. Meanwhile, the deletion range [7,9] of v1 is equivalent to [5,7] of v2, because [7], [8], and [9] in v1 are equivalent to [5], [6], and [7], which are consecutive in v2.

• **Equivalent operations:** Two insert operations are equivalent if the insertion points are equivalent and the bytes being inserted are identical. Two delete operations are equivalent if the deletion ranges are equivalent. For example, insert(v1, [8,9], k) is equivalent to insert(v2, [6,7], k) and delete(v1, [7,7]) is equivalent to delete(v2, [5,5]).

2.4 Schedule

As a final step before we define strict mergeability, we introduce the notion of a schedule of operations.

**Definition 1** A schedule is an ordering of operations in which the operations are applied to the document.

Note that by the time a document needs to be merged, individual users may have performed multiple operations on the document, so we may have to merge two (or more) sets of operations. Let $S_1 = O_{(1,1)}O_{(1,2)}\ldots O_{(1,m)}$ and $S_2 = O_{(2,1)}O_{(2,2)}\ldots O_{(2,n)}$ be two sequences of operations that have been performed by the users $u_1$ and $u_2$, respectively. Also, let $S = O_1O_2\ldots O_{m+n}$ be a permutation of the concatenated sequence $S_1S_2$. Then we say that $S$ is a possible schedule from $S_1$ and $S_2$ if $S$ preserves the original ordering within $S_1$ and $S_2$. That is, if $O_i$ precedes $O_j$ in $S_1$ (or in $S_2$), then $O_i$ also precedes $O_j$ in $S$.

In some cases, even if $S$ is a possible schedule from $S_1$ and $S_2$, this schedule may not be actually “applicable” to the document. The following example illustrates this point.

Example 1 Consider the scenario in Figure 5. The document initially contains the text “alice” and is modified concurrently by two sets of edit operations: $I_{u_1} = \text{insert}(v, [3,2], g)$ and $D_{u_1} = \text{delete}(v_1, [3,5])$ by user $u_1$ and $D_{u_2} = \text{delete}(v, [2,2])$ and $I_{u_2} = \text{insert}(v_3, [0,1], f)$ by user $u_2$. Since these changes are made concurrently, a number of different schedules are possible. For example, in Figure 6(a), we show the schedule ($I_{u_1}D_{u_2}D_{u_1}I_{u_2}$). Under this schedule, we can see that all operations can be applied to the document, producing the final document “face.” In contrast, the schedule ($D_{u_2}I_{u_2}I_{u_1}D_{u_1}$) in Figure 6(b) is also a possible schedule, but it encounters a problem when it tries to apply $I_{u_1}$; since $D_{u_2}$ deletes $l$, $I_{u_1}$ cannot find an equivalent insertion point, so the operation cannot locate where the new byte $g$ should be inserted.

The above example illustrates that not every possible schedule is “valid.”

**Definition 2** If all operations in the schedule find their equivalent edit location when they execute, the schedule is valid, otherwise, it is invalid.

In Table 1, we list all possible schedules in Example 1. In the table, we also show the schedules’ validity and the final documents they produce.

2.5 Strict mergeability

We are now ready to more formally state what will constitute “safely mergeable changes.”

**Definition 3** Two concurrent sets of operations are strictly mergeable if all possible schedules derived from them are valid and produce the same final document.

One way to state “safely mergeable changes” is for the changes to be strictly mergeable. In particular, strict mergeability is very conservative in determining when multiple
sets of operations are “safe to merge.” Under this definition, multiple sets of operations are safe to merge if and only if the users get exactly the same document under every possible schedule for the operations. Depending on the application, this interpretation of mergeability may be too strict, so a more relaxed notion of mergeability may be preferable. In the next section, we briefly explore an alternative definition of mergeability.

Also, note that checking strict mergeability for sets of concurrent operations can potentially be very expensive. For example, one naïve approach to checking strict mergeability is to enumerate all possible schedules for the set of operations that need to be merged (such as was done in Table 1), check if each operation finds an equivalent edit location at its execution time under every schedule and make sure that the final document is always the same. Unfortunately, the number of possible schedules grows combinatorially with the number of operations involved, which makes this naïve algorithm prohibitively expensive even for reasonably large sets of operations. Therefore, it will be desirable to develop a more efficient method to enforce strict mergeability, which is the main focus of Section 3.

2In general, the number of possible schedules is $\prod_{i=1}^{n} m_i!$ where $n$ is the number of concurrent sets of operations and $m_i$ is the number of operations in the $i^{th}$ set.

### Table 1: All six possible schedules for the example in Figure 5.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Valid?</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_{u1} D_{u1} D_{u2} I_{u2}$</td>
<td>yes</td>
<td>face</td>
</tr>
<tr>
<td>$I_{u1} D_{u1} I_{u2} D_{u2}$</td>
<td>yes</td>
<td>face</td>
</tr>
<tr>
<td>$I_{u1} D_{u1} D_{u2} I_{u2}$</td>
<td>yes</td>
<td>face</td>
</tr>
<tr>
<td>$D_{u2} I_{u1} D_{u1} I_{u2}$</td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>$D_{u2} I_{u1} I_{u2} D_{u1}$</td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>$D_{u2} I_{u1} D_{u2} I_{u1}$</td>
<td>no</td>
<td>-</td>
</tr>
</tbody>
</table>

### Figure 6: Two possible schedules for the operations in Figure 5, (a) is valid, but (b) is invalid.

2.6 Serial mergeability

Note that under our definition of strict mergeability all changes are considered “equally important”; that is, all possible schedule from the concurrent sets of operations should be valid and produce the same final document. In some cases, it may be more desirable to give “precedence” to one set of changes over others. For example, when a particular software module is “owned” by a specific developer, we may want to give precedence to the changes made by the owner over the changes from others. Our definition of serial mergeability tries to incorporate this precedence relationship between multiple changes.

**Definition 4** Consider two sets of concurrent operations $S_1$ and $S_2$ made by users $u_1$ and $u_2$, respectively. We assume that the user $u_1$ has precedence over the user $u_2$. Then we consider that $S_1$ and $S_2$ are serially mergeable as long as the schedule $S_1 S_2$ is valid.

Note that under serial mergeability, when we check the validity of $S_1 S_2$, a conflict may occur only during the application of $S_2$’s operations (i.e., after all operations in $S_1$ is applied) because $S_1$’s operations are applied to the same version to which they were originally applied. Therefore, when $S_1$ and $S_2$ are not serially mergeable, we may simply discard the changes from $S_2$ preserving the operations from $S_1$.

Given that the above definition of serial mergeability is a “relaxed” version of strict mergeability — it checks the validity of a particular ordering of the operations as opposed to all possible orderings — it is relatively straightforward to see that strict mergeability implies serial mergeability. Specifically, if changes are strictly mergeable then all possible schedules must be valid. In particular, the schedule $S_1 S_2$ is one of these valid schedules, so the changes are serially mergeable.

### 3. ENFORCING MERGABILITY

In the previous section, we presented two possible definitions of safely mergeable changes. In the case of strict mergeability, we have also seen that the number of all possible schedules grows combinatorially with the number of operations, so enforcing strict mergeability directly by enumerating all schedules and checking their validity may not be a feasible solution. In this section, we develop algorithms that can enforce the two mergeabilities efficiently. The performance of these algorithms will then be examined later in Section 4.

#### 3.1 Conflicting operations

In our previous examples, we observed that the two sets of operations that were not mergeable often contained pairs of “conflicting” operations that operate on the same location of a document. For example, in Figure 3 both users attempt to insert new bytes at the beginning of the document, which leads to the situation where the final document may be different depending on the order of execution.

To formalize this notion of “conflicting operations,” consider the two operations $\text{insert}(v, [1, j], x)$ and $\text{delete}(v, [1, m])$, where $v$ is some version of a document, $i$ and $j$ are the left and right boundary of the insertion point, $x$ is some byte, and the deletion range $[1, m]$ denotes the consecutive set of locations enclosed by the two boundaries, $l$
and \(m\). For simplicity, we shall use the short-hand \(I(i, j, x)\) for the \textit{insert} and \(D(l, m)\) for the \textit{delete} assuming that both operations are against the same version.\(^3\)

Note that whether the two operations are “conflicting” or not may sometimes depend on our interpretation of conflict. For instance, if two users seek to delete overlapping regions of a document, it could be interpreted that the deletions do not conflict, as long as we make sure that the final document deletes the union of the deleted regions. Alternatively, we may consider that these deletions conflict because, by the time the second deletion is applied, some of its deletion ranges do not exist in the document. In this paper, we decide to use the most “conservative” interpretation, by using the following compatibility matrix:

<table>
<thead>
<tr>
<th>(I(l, m, y))</th>
<th>(I(i, j, x))</th>
<th>(D(l, m))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i = l) and (j = m)</td>
<td>(l = m) or (m \in [i, j])</td>
<td>([i, j] \cap [l, m] \neq \emptyset)</td>
</tr>
</tbody>
</table>

The above table specifies the conditions under which two operations are deemed “incompatible.”

\(\textbf{Definition 5}\) When two operations are not compatible (as identified by the above compatibility matrix), we consider them to be \textit{conflicting operations}.\(^4\)

Basically, the definition of conflicting operations looks for overlapping edit locations between two operations. If they do, it considers that one operation will alter the edit location of the other operation, resulting in the second operation being unable to find its equivalent edit location.

We illustrate some instances of conflicting and non-conflicting operations using the following four examples. In all four examples, we assume that the initial document is \textit{abcde}f\(g\).

\(\textbf{Example 2} \) \(I(3, 4, h)\) and \(I(3, 4, i)\)

Both \(I(3, 4, h)\) and \(I(3, 4, i)\) seek to insert new bytes to the same location (between \(c\) and \(d\)). According to the compatibility matrix above, the two operations are conflicting because the condition \(i = l = 3\) and \(j = m = 4\) is true. Intuitively, this choice of compatibility is reasonable, because when these operations are applied together, it is unclear if the final document should be \textit{abcde}h\(i\)\(d\)f\(g\) (if we execute \(I(3, 4, h)\) before \(I(3, 4, i)\)) or \textit{abcde}\(i\)\(h\)\(d\)\(e\)f\(g\) (if we execute \(I(3, 4, i)\) before \(I(3, 4, h)\)).

\(\textbf{Example 3} \) \(I(3, 4, h)\) and \(D(2, 3)\)

The compatibility matrix states that the two operations are conflicting because they satisfy the condition \(3 \in \{2, 3\}\). Again this is a reasonable choice, because if \(D(2, 3)\) is applied before \(I(3, 4, h)\), the insertion point of \(I(3, 4, h)\) does not exist in the document.

\(\textbf{Example 4} \) \(D(1, 3, h)\) and \(D(3, 3)\)

The compatibility matrix states that the two operations are conflicting because \(\{1, 3\} \cap \{3, 3\} = \{3\} \neq \emptyset\). Here, we are using a conservative definition of conflicting operation; for two deletion operations, we make sure that there is no overlapping region between the two.

\(3\)In some cases, the operations performed by two users may not be against the same version. In this case, we will first have to “transform” the operations to equivalent operations against a common version, and check whether transformed operations conflict or not.

\(4\)We illustrate some instances of conflicting and non-conflicting operations using the following four examples. In all four examples, we assume that the initial document is \textit{abcde}f\(g\).

\(\textbf{Example 5} \) \(I(3, 4, h)\) and \(I(4, 5, h)\)

Using the compatibility matrix, we know that the two operations are non-conflicting because neither \(3 \neq 4\) nor \(4 \neq 5\). This choice of compatibility is reasonable because regardless of which insertion is executed first, the resulting document is always \textit{abcde}h\(i\)\(d\)\(e\)\(f\)\(g\).

Given our definition of conflicting operations, the following theorem provides our key result on how we can enforce strict mergeability efficiently.

\(\textbf{Theorem 1}\) Consider two concurrent sets of operations \(S_1\) and \(S_2\). For every \(O_i \in S_1\) and \(O_j \in S_2\), if \(O_i\) does not conflict with \(O_j\), \(S_1\) and \(S_2\) are strictly mergeable.

\(\textbf{Proof}\) To prove the two sets of operations are strictly mergeable, we first show that there exists a valid schedule, and then we show that every possible schedule can be derived from this schedule while maintaining its validity and will generate the same document.

Consider a schedule in which all operations in \(B\) come after those in \(A\). For every \textit{insert} operation in \(B\), since it does not conflict with its preceding insertions and deletions in \(A\), its insertion point will neither be \textit{separated} nor be \textit{removed} when it executes. For every \textit{delete} operation in \(B\), since it does not conflict with its preceding insertions and deletions in \(A\), its deletion range will neither be \textit{discontinued} nor be \textit{removed} when it executes. Since every operation in \(B\) can find its equivalent edit location when it executes, this schedule is valid.

Under the notion of possible schedules, in which operations from the same set always maintain their relative ordering but interleave freely with operations from another set, any two schedules among all possible schedules can be transformed from one to another by swapping neighboring operations (each coming from a different set) a finite number of times.

Consider two schedules, \((\text{PRE}, a, b, \text{POST})\) and \((\text{PRE}, b, a, \text{POST})\), that differ by the ordering of two neighboring operations where \(a\) comes from \(A\), \(b\) comes from \(B\), and \(\text{PRE} / \text{POST}\) denote an arbitrary sequence of operations. Since \(a\) and \(b\) are non-conflicting operations, they will not affect each other’s edit locations, so, the two schedules generate the same document after the two operations, \(a\) and \(b\), execute and also after \text{POST} has been applied. As a result, if every operation in set \(A\) does not conflict with any operations in set \(B\), all possible schedules are valid and results in the same document, which means they are strictly mergeable.

The above theorem suggests a strict mergeability enforcement mechanism that can be significantly more efficient than the naïve \textit{enumerate} and \textit{check} method; given two concurrent sets of operations \(S_1\) and \(S_2\), we ensure that \(O_i\) and \(O_j\) do not conflict for any \(O_i \in S_1\) and \(O_j \in S_2\), which can be done in \(O(n^2)\) as opposed to in \(O(n!\) under the naïve method. Later in Section 3.3 we provide more detailed pseudo code for this mergeability enforcement.

### 3.2 Using GUID to check equivalence

Identification of conflicting pairs of operations is done by checking the compatibility matrix. However, this matrix relies on the operations that are applied to the same version of the document. One way to convert operations to
the same version of the document is by finding the equivalent locations discussed in Section 2.3. In order to help us in identifying the equivalence of locations between multiple versions of a document, we may assign a globally unique identifier (GUID) to every byte in the document and preserve the GUIDs as the document evolves over time, as long as the corresponding byte locations remain in the document. This way, we can check the equivalence of locations in two different versions simply by checking the existence of the identical GUID in the versions.

One way of making sure that GUIDs are globally unique in a distributed environment is to use the combination of the user ID and the number of bytes inserted by that user so far.

Example 6 In Figure 7 user $u_1$ (user ID: $A$) executes \texttt{insert}($e$, $[0,0]$, \texttt{abc}) to insert the characters \texttt{abc} into a blank document, so the GUID representation is $A1A2A3$. Now, two other users, $u_2$ (user ID: $B$) and $u_3$ (user ID: $C$), obtain local copies of this document and issue the commands \texttt{insert}($v_1$, $[2,3]$, \texttt{dc}) and \texttt{insert}($e_1$, $[3,-1]$, \texttt{fhg}) respectively. After $B$’s \texttt{insert} the new document becomes \texttt{abdec} (GUID: $A1A2B1B2A3$) and then $C$’s \texttt{insert} makes it \texttt{abdecfhg} (GUID: $A1A2B1B2A3C1C2C3$).

While GUIDs simplify the task of finding equivalent locations, there are some tradeoffs. First, under the GUID scheme the deletion ranges may become ambiguous. For example, if we have two versions of a document, $v_1$ ($A1A2A3$) and $v_3$ ($A1A2B1B2A3$), the deletion range $\{A2, A3\}$ could mean $\{A2, A3\}$ or $\{A2, B1, B2, A3\}$. Since it is unclear, for deletions we will specify the set of all consecutive bytes to be deleted. For example, delete($\{A2, B1, B2, A3\}$), which makes it obvious that the intention is to delete four bytes.

Another tradeoff is the increased storage requirements required by the GUID. To reduce this overhead, we may use a GUID run-length encoding where $A1A2B1B2A3C1C2C3$ can be encoded as $(A1,2)(B1,2)(A3,1)(C1,3)$ with each pair indicating the starting GUID and the number of consecutive GUIDs from the same user (refer to Figure 7 for more examples). From our analysis of real document edit traces, we find that users often generate a consecutive sequence of bytes in chunks, so this run-length encoding scheme significantly reduces the storage requirements. Later in the evaluation section, we examine the storage overhead of using the GUID scheme with the run-length encoding optimization.

### 3.3 Mergability checking algorithms

With the GUID scheme in place, we now describe our methods to efficiently check strict and serial mergability in Algorithms 1 and 2.

Consider again the example in Figure 5. The document \textit{alice} (GUID: $(A1,5)$) is being concurrently modified by $u_1$ (user ID: $B$) and $u_2$ (user ID: $C$). We want to check if these changes are strictly mergeable or serially mergeable. $B$’s changes are $\{\texttt{insert}(A2, A3, g), \texttt{delete}(\{A3\})\}$ and $C$’s changes are $\{\texttt{delete}(A2), \texttt{insert}(0, A1, f)\}$.

Algorithm 1 compares every pair of operations involved and checks for conflicting pairs (non-compatible operations). For this example, we find that $\texttt{insert}(A2, A3, g)$ conflicts with $\texttt{delete}(\{A2\})$, because $A2 \in \{A2\}$. Therefore, these operations are not strictly mergeable.

Algorithm 2 checks the validity of a schedule of a given document, ensuring that insert locations are adjacent and deletion ranges are consecutive at the time of execution. Here, the subroutines \texttt{adjacent} and \texttt{consecutive} can be defined in any number of ways, but we suggest using the GUID representation of the document and look for the GUID of each location in the latest version of the document as an efficient way to determine adjacency and consecutiveness. For this example, if we use the schedule $(I_{u_1}, D_{u_2} D_{u_1} I_{u_2})$ we find that they are serially mergeable, but the schedule $(D_{u_2} I_{u_2} I_{u_1})$ is not, because $A2$ and $A3$ are not adjacent in $C1A1A3A4A5$.

### 3.4 History tracking

In order to use the previous algorithms to check mergeability, we also need a mechanism to determine (1) when documents have diverged, (2) what operations were executed since the divergence, and (3) how to synchronize documents that have diverged. We briefly go over possible mechanisms for history checking and synchronization in the rest of this section.
Algorithm 2 Algorithm for checking serial mergeability

\begin{verbatim}
serial_mergeability_check(S, doc)
    if valid(S, doc) then return true
    else return false
end if
valid(S, doc)
for every operation e in schedule S
    if e is insert then
        let e be insert(a, b, x)
        if adjacent(a, b, doc) then doc = doc.insert(a, b, x)
        else return false
    end if
    else if e is delete then
        let e be delete(C)
        if consecutive(C, doc) then doc = doc.delete(C)
        else return false
    end if
end for
\end{verbatim}

Figure 8: Time at which each document was last synchronized with each neighbor.

Divergence

One naïve history tracking mechanism would require each user to store the contents of every version of every document together with the information when documents were synchronized with other users as is illustrated in Figure 8. Then whenever two users want to synchronize their documents, they can easily identify what changes need to be merged together.

A slight modification to this approach would be for each document to have a designated owner. The owner would be responsible to store the entire history of changes (and at what times the changes occurred) for that document, while the other users would only need to store the history of changes made locally since the last synchronization. As before, document divergence could be detected by determining from the owner of the document when versions had last been synchronized with a particular user and then checking to see if either the owner or the other user had made any changes since that time. Again, the history would be examined to isolate the sets of operations made by each user since the divergence point.

Table 2: Metadata of documents investigated

<table>
<thead>
<tr>
<th>Properties</th>
<th>TeX</th>
<th>PHP</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of files</td>
<td>88</td>
<td>45</td>
<td>970</td>
</tr>
<tr>
<td>Avg. initial file size (bytes)</td>
<td>6,813</td>
<td>5,208</td>
<td>3,708</td>
</tr>
<tr>
<td>Avg. final file size (lines)</td>
<td>158</td>
<td>165</td>
<td>119</td>
</tr>
<tr>
<td>Avg. final file size (bytes)</td>
<td>9714</td>
<td>13,042</td>
<td>4,215</td>
</tr>
<tr>
<td>Avg. number of versions</td>
<td>12.2</td>
<td>27.7</td>
<td>6.8</td>
</tr>
<tr>
<td>Avg. line insertions</td>
<td>142</td>
<td>34.9</td>
<td>970</td>
</tr>
<tr>
<td>Avg. line deletions</td>
<td>29.6</td>
<td>10.8</td>
<td>9.4</td>
</tr>
<tr>
<td>Distribution of co-authorship</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># files - (# authors)</td>
<td>42 - (1)</td>
<td>4 - (1)</td>
<td>497 - (1)</td>
</tr>
<tr>
<td>29 - (2)</td>
<td>6 - (2)</td>
<td>273 - (2)</td>
<td></td>
</tr>
<tr>
<td>17 - (3)</td>
<td>3 - (3)</td>
<td>135 - (3)</td>
<td></td>
</tr>
<tr>
<td>13 - (4)</td>
<td>46 - (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 - (5+)</td>
<td>19 - (5+)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Synchronization

After we identify the divergence point and the sets of operations applied to each version since then, all operations need to be transferred to one of the users in order to check their mergeability using the algorithms described before. Once the changes are deemed mergeable and the final document is obtained, both users need to update their local cache to this final document either by getting the final document (if a complete document change history is not maintained by all users) or by transferring the union of changes by all users.

4. EVALUATION

In this section, we evaluate the effectiveness of our two definitions of mergeability and the efficiency of our mergeability enforcement algorithms. In particular, we are interested in addressing the following questions: (1) how likely will users encounter non-mergeable changes under our definitions of mergeability? (2) what is the computational cost of mergeability enforcement? How efficient are our mergeability enforcement algorithms compared to the naïve enumerate and check method? (3) how much storage overhead does the GUID scheme incur?

To study these issues, we use the document edit traces collected from three real CVS repositories, one which has been used by our research group for collaborative authoring of papers over a span of four years and two which are publicly hosted through SourceForge [5, 6] and used for collaborative software development by people residing in multiple continents. We start with the description of this dataset.

4.1 Description of the dataset

Our dataset consists of TeX files from our CVS repository created in 2003, PHP source files from the phpPgAdmin [5] repository, and Java source files\(^4\) from the Hibernate [6] repository. This selection allows us to explore the characteristics of both text and source code document editing. A summary of the metadata of the documents in this dataset is shown in Table 2. From the table, we observe that source code documents tend to allow more collaborators to work on the same file and, in general, the average number of changes between versions are less than that of text documents.

\(^4\)Java files under the src sub-directory.
due to missing information, we have to make a few assumptions to reconstruct the editing history to the best of our abilities. Later in Section 4.5, we provide a more detailed explanation of the information in the CVS repository and the assumptions that we made during this history reconstruction process.

### 4.2 Frequencies and types of conflicts

From Table 2, we can see that more than half of the documents in our dataset involve two or more authors. However, we want to know how frequently multiple authors encounter concurrent edit scenarios and, of those, how many require the users to manually resolve conflicts. Table 3 we outline some of the important statistics related to this issue. From the table, we see that roughly 5% of all commits (i.e., groups of changes made by the users) involved concurrent edit operations. Among these concurrent edit scenarios, around 20%-30% resulted in "conflict"5 according to the CVS history requiring manual intervention, while the rest were successfully merged without manual intervention. Given the definition of the conflict of CVS, all of the mergeable operations in the CVS history are serially mergeable, but it is interesting to note that, depending on the repository, significantly fewer are strictly mergeable. In particular, we note that for source code CVS repositories (PHP and Java datasets), most of the concurrent edit operations are serially mergeable, but not strictly mergeable.

In Figure 9 we show how many conflicting pairs of operations existed when concurrent changes resulted in non-strictly mergeable scenarios. The overall height of each bar corresponds to the average number of conflicting pairs of operations existing in non-strictly mergeable changes. This number is further broken down by types of conflicts, where \(i\) stands for insertion and \(d\) stands for deletion. Thus, \(i-i\) represents an insertion which conflicts with a deletion that was previously committed to the server.

On average, there are 4.91, 5.22, and 3.69 pairs of conflict operations for files in the \(\TeX\), PHP, and Java repositories respectively. Among them, conflicting \(i-i\) and \(d-i\) pairs occur more often than the other combinations, contributing to 63%, 87%, and 95% of the total number of conflicting operations respectively.

### 4.3 Cost of mergeability checking

We now evaluate the cost of carrying out the mergeability checking suggested by this paper.

For strict mergeability, we need to identify the set of changes made at the server and the set of changes made locally by the client and then determine if there are any conflicting operations between the two sets. In Table 4 we show the statistics related to the mergeability checking cost. From the table, we see that on average the set of changes made at the server involves between 15 and 26 operations with between 10 and 16 operations made at the client. Therefore, using Algorithm 1 to enforce strict mergeability requires checking for conflicts among close to 300 pairs of operations. This is a vast improvement over the naïve enumerate and check method, which requires checking an average of over 7.7 million schedules \((11+15)!/11!15!\) in the case of the Java repository.

To enforce serial mergeability using Algorithm 2, it is only necessary to determine if the schedule is valid by ensuring that the insertion points are still adjacent and the deletion ranges are still consecutive at the time of execution. The average size of GUIDs at the point of mergeability checking is roughly between 100 and 200 run-length pairs, so, serial mergeability checking will be much more efficient than strict mergeability checking.

### 4.4 Evaluation of the GUID scheme

As we suggested in Section 3.2, one way to efficiently check the mergeability of operations is by using our proposed GUID scheme to track equivalent locations across document versions. When locating these equivalent locations is necessary, the GUID scheme is quite handy. However, according to our study, roughly 95% of the updates do not require merging changes, so there is no savings provided by the GUID representation in the majority of situations. Therefore, we need to investigate the storage and computational overhead of maintaining the GUIDs to ensure the extra costs are worth the savings achieved by only 5% of the updates.

#### Storage costs

We were able to extract the RCS formatted forward edit scripts from the CVS repositories allowing us to rebuild the

5RCS formatted forward edit scripts specify the locations and contents of line insertions and deletions required to
A working file was patched to match the repository.

A working file was copied from the repository.

A merge was necessary and it succeeded.

A merge was necessary but collisions were detected.

Checkout

A file was modified.

108283

Explanation

PHP

6890 records

40

50

A file was added for the first time.

Java

architecture, where a client checks out the server’s version of the document to the latest version. CVS employs the server-client line additions and deletions, from the creation of the document over time and stores the entire edit history, represented as documents’ edit histories.

In this section, we describe our analysis of the CVS repositories and explain the assumptions we made to reconstruct the documents over time. All of the information we need to evaluate the documents, including GUID representations, is within the repositories themselves.

4.5 Description of CVS repositories

As was previously mentioned, CVS does not keep track of all of the information we need to evaluate the documents, so we made several assumptions to reconstruct the data. In this section, we describe our analysis of the CVS repositories and explain the assumptions we made to reconstruct the documents’ edit histories.

The CVS repository tracks changes made to documents over time and stores the entire edit history, represented as line additions and deletions, from the creation of the document to the latest version. CVS employs the server-client architecture, where a client checks out the server’s version of the document from one version to the next.

GUID representation of the documents (in terms of lines of text rather than bytes) over their evolution.

For this evaluation, we assume that the GUID run-length encoding uses eight bytes of data: one byte for the user ID, four bytes for the starting sequence ID, and three bytes to store the number of sequential bytes (or lines) encoded by this GUID. We compare the ratio of the size of the GUID to its corresponding document size as the document evolves. As Figure 10 demonstrates, the GUID encoding is significantly smaller than the corresponding document (usually less than 10%). As the number of revisions increases, this ratio does not increase significantly (files which were revised hundreds of times still maintain a GUID size less than 12% of the overall document size), indicating that the storage overhead of the GUID scheme is reasonably low.

Computational costs

Two types of operations contribute to the computational costs of maintaining the GUID scheme: the cost of finding an equivalent location and the cost of recalculating the document’s GUID representation after an insertion or deletion. Whether it is an insertion or deletion, finding the equivalent edit location can be done in at most a single pass over the GUID representation of the document and, if the user keeps a counter of the number of insertions so far in the document, updating the GUID representation after an edit is also of negligible cost.

Guidance representation of the documents (in terms of lines of text rather than bytes) over their evolution.

For this evaluation, we assume that the GUID run-length encoding uses eight bytes of data: one byte for the user ID, four bytes for the starting sequence ID, and three bytes to store the number of sequential bytes (or lines) encoded by this GUID. We compare the ratio of the size of the GUID to its corresponding document size as the document evolves. As Figure 10 demonstrates, the GUID encoding is significantly smaller than the corresponding document (usually less than 10%). As the number of revisions increases, this ratio does not increase significantly (files which were revised hundreds of times still maintain a GUID size less than 12% of the overall document size), indicating that the storage overhead of the GUID scheme is reasonably low.

Computational costs

Two types of operations contribute to the computational costs of maintaining the GUID scheme: the cost of finding an equivalent location and the cost of recalculating the document’s GUID representation after an insertion or deletion. Whether it is an insertion or deletion, finding the equivalent edit location can be done in at most a single pass over the GUID representation of the document and, if the user keeps a counter of the number of insertions so far in the document, updating the GUID representation after an edit is also of negligible cost.

Guidance representation of the documents (in terms of lines of text rather than bytes) over their evolution.

For this evaluation, we assume that the GUID run-length encoding uses eight bytes of data: one byte for the user ID, four bytes for the starting sequence ID, and three bytes to store the number of sequential bytes (or lines) encoded by this GUID. We compare the ratio of the size of the GUID to its corresponding document size as the document evolves. As Figure 10 demonstrates, the GUID encoding is significantly smaller than the corresponding document (usually less than 10%). As the number of revisions increases, this ratio does not increase significantly (files which were revised hundreds of times still maintain a GUID size less than 12% of the overall document size), indicating that the storage overhead of the GUID scheme is reasonably low.

Computational costs

Two types of operations contribute to the computational costs of maintaining the GUID scheme: the cost of finding an equivalent location and the cost of recalculating the document’s GUID representation after an insertion or deletion. Whether it is an insertion or deletion, finding the equivalent edit location can be done in at most a single pass over the GUID representation of the document and, if the user keeps a counter of the number of insertions so far in the document, updating the GUID representation after an edit is also of negligible cost.

4.5 Description of CVS repositories

As was previously mentioned, CVS does not keep track of all of the information we need to evaluate the documents, so we made several assumptions to reconstruct the data. In this section, we describe our analysis of the CVS repositories and explain the assumptions we made to reconstruct the documents’ edit histories.

The CVS repository tracks changes made to documents over time and stores the entire edit history, represented as line additions and deletions, from the creation of the document to the latest version. CVS employs the server-client architecture, where a client checks out the server’s version of the document from one version to the next.

The document (making a local copy), modifies the local copy, and then commits those changes to the final version which resides on the server. CVS keeps a history of changes made to the server’s version and also keeps a log of client interaction operations (such as clients checking out the server’s version, clients committing changes to the server, and any conflicts which may have been encountered). The subset of operations we investigated is described in Table 5 and the distribution of their relative occurrence is shown in Figure 11.

Figure 12 illustrates a typical scenario when two users edit the same document concurrently and combine their changes using the CVS system. At time $T_1$, both user $A$ and user $B$ checkout the latest version ($v_1$) of the document (identified by the $O$, $U$, or $P$ record types). User $A$ starts modifying her local copy, produces a modified version (denoted as $v_1'$), and commits the changes to the CVS repository (identified by the $M$ record type) at time $T_2$. Since the server copy was not modified between times $T_1$ and $T_2$, user $A$ commits her changes without any conflicts and advances the document in the repository to a newer version ($v_2$). Concurrently, user $B$ starts modifying his local copy and produces another modified version (denoted as $v_1''$). When user $B$ tries to commit his changes, the CVS repository notices that the document has been updated from version $v_1$ to $v_2$, so CVS mandates that $B$ update his local copy to match the server’s version $v_2$ merging the differences between the two files (identified by the $C$ or $G$ record types). After merging the changes (either without conflicts or after $B$ manually reconciles the differences) user $B$ commits the final changes to the CVS repository.
5. RELATED WORK

A large body of related work exists in the literature. We outline some of the most relevant work here.

The serializability theory and concurrency control mechanisms [7] have been extensively studied in the database community. Similarly to our work, the serializability theory also formalizes the notion of a “good schedule” when multiple transactions are executed concurrently, but the basic unit of operation is different: read and write on atomic variables as opposed to insert and delete on a byte stream. In a collaborative authoring environment, we find that users tend to work on a relatively large document for an extended period of time, so it is often too restrictive to model an individual document as a single variable and to provide exclusive access to a document to a particular user during an edit. Therefore, the granularity of modifying individual bytes, as our mergeability supports, is an improvement over modifying entire documents as a single unit.

Version control systems, such as CVS [1] and subversion [2], are designed for source code management among a group of developers. Although concurrent modifications to the same source file are allowed, they are generally not encouraged. The exact conflict-detection mechanism of these systems varies, but the basic principle can be summarized as “no overlapping region” among concurrent edits. This principle is closely related to the conflict-detection mechanism described in Section 3, but we investigate the relationship of this mechanism to the higher level notion of “mergeable changes.” In addition, we study the usage pattern of collaborative users to understand how often and what type of conflicts the users are likely to encounter during concurrent edits by analyzing a real dataset. Therefore, rather than restricting or discouraging concurrent updates, our mergeability theory learns from actual usage patterns and suggests a variety of methods to enforce mergeability according to a user’s or application’s specifications.

The work by Ellis and Gibbs [8] proposed a framework for groupware systems that allow synchronous real-time collaborative authoring. The core idea is a technique called operational transformation where modifications to a document are transformed before applying the changes to local copies held by each participant. Various research efforts [9, 10, 11, 12, 13] have further elaborated on this framework and have applied similar techniques under different system settings [14, 15, 16]. The underlying principle of this body of work is, again, to allow concurrent edits only on non-overlapping regions. Based on this principle, they describe mechanisms to allow only such operations [17, 18] or to prioritize the operations when conflicts occur [8, 9, 11]. In this paper we apply this technique to concurrent document editing, ensuring that only non-overlapping regions can be modified in order for two operations to be considered “mergeable.”

Distributed and/or offline file systems, such as NFS [19], AFS [20], GFS [21], or Windows Offline Files [3], employ various concurrency-control and caching mechanisms to allow accessing files from multiple machines. However, the basic unit of concurrency control is typically a file, and, due to the risk of losing changes, it is not recommended (or even disallowed) that users modify the same file from multiple machines concurrently. When the same file is modified concurrently, the task of merging the changes is left to the user. Our mergeability theory takes some of the burden off of the user by allowing automatic merges more frequently, by using a byte stream as the basic unit, rather than a whole file.

6. CONCLUSION

In this paper, we developed a theory of mergeability that attempts to model the scenario of merging two versions of a document that are being modified concurrently. We also enumerated several operations involved in this merge process and clarified conditions for when such changes can be merged safely without causing inconsistency. Based on the developed theory, we described a GUID scheme and algorithms to carry out the mergeability checking efficiently to ensure only compatible changes are merged to the final document.

In addition, we analyzed real document change history data acquired from several CVS repositories to evaluate the efficiency of our proposed scheme in terms of its storage requirements and computational costs.

7. REFERENCES


[21] The OpenGFS Project.