A Partial Persistent Data Structure to Support Consistency in Real-time Collaborative Editing

Qinyi Wu #1, Calton Pu #2, João Eduardo Ferreira #3

#1College of Computing, Georgia Institute of Technology, USA
#2{qxw, calton}@cc.gatech.edu
#3Department of Computer Science, University of São Paulo, Brazil

Abstract—Co-authored documents are becoming increasingly important for knowledge representation and sharing. Tools for supporting document co-authoring are expected to satisfy two requirements: 1) querying changes over editing histories; 2) maintaining data consistency among users. Current tools support either limited queries or are not suitable for loosely controlled collaborative editing scenarios. We address both problems by proposing a new persistent data structure—partial persistent sequence. The new data structure enables us to create unique character identifiers that can be used for associating meta-information and tracking their changes, and also design simple view synchronization algorithms to guarantee data consistency under the presence of concurrent updates. Experiments based on real-world collaborative editing traces show that our data structure uses disk space economically and provides efficient performance for document update and retrieval.

I. INTRODUCTION

Real-time collaborative editing facilitates geographically distributed users to work together through individual contributions. Examples of tools include Google Wave Live Editing [1] and SubEthaEdit [2]. The most recent trend suggests that they may also shape traditional applications such as emails and wikis. Two important challenges exist.

First, collaborative editing applications not only need to efficiently maintain editing histories for large amount of data, but are expected to persistently tracking changes of documents as well. Compared to standalone document editing, co-authored documents are less controlled, especially in an open community. For example, users may query the authorship of recently inserted text or attach meta-data to pieces of data. Traditional content management systems [3] have difficulties in tracking footprints of characters over document revision histories because their position indexes are frequently changed.

Second, collaborative editing applications need to coordinate edits of users to guarantee data consistency because users can update a replica of the shared document anywhere and anytime. Several view synchronization algorithms have been proposed in the literature [4], [5]. These approaches have difficulties in loosely-controlled editing scenarios with dynamic user membership during an editing session because their correctness relies on the knowledge of causal relationships which are expensive to maintain [6], [7], [8].

The limitations of early approaches to address both challenges arise from the lack of a persistent data structure to index characters of documents. The persistent indexes are immutable over document modification and can serve as keys to associate various meta-information and help design simple view synchronization algorithms. Typical reasons for preventing building a persistent data structure on the top of a document are the concerns of extra space requirements and potential degrading performance for document update and retrieval.

In this paper, we investigate the practicality of building a persistent data structure over unstructured documents. Our contributions are summarized below:

- Proposal of a persistent data structure, partial persistent sequence (PPS). We define its data model and operations.
- Prototype of a collaborative editing tool based on PPSs. Experiments based on real world collaborative editing traces show that PPSs can be updated within seconds for hundreds of edits and achieve an average compression ratio at one fifth for document version histories.

II. PARTIAL PERSISTENT SEQUENCES

A partial persistent sequence (PPS) is an ordered set of items indexed by persistent and unique position identifiers that are rational numbers. For example, the position identifiers for the PPS “abc” could be 0, 0.5 and 1 respectively. We call the position identifiers in a PPS position stamps to differentiate them from the indexes in traditional sequences. The PPS data structure keeps all data items occurring in its entire editing history. We call it partial persistent sequence because all revisions can be accessed but only the newest revision can be updated [9].

A. Data Model

A PPS is defined by a pair \( (S, M) \), where

- \( S \): a set of unique rational numbers, which are called position stamps. \( S = \{ s_i \in \mathbb{Q} \mid 1 \leq i \leq n, n \in \mathbb{N} \} \).
- \( M \): a mapping function from \( S \) to \( \Sigma \), where \( \Sigma \) is a finite set of characters. \( \Sigma \) contains a null character \( \phi \) that is different from any other characters allowed in user applications. Let \( \Sigma^c = \Sigma - \phi \).

The position stamps in \( S \) are totally ordered by less than \( < \) defined on \( \mathbb{Q} \). For \( s_i \in S \), we use \( s_{i+1} \) to denote the next position stamp in \( S \) such that \( s_i < s_{i+1} \) and \( \neg(\exists s_x \in S, s_i < s_x < s_{i+1}) \). Similarly, we use \( s_{i-1} \) to denote the previous
position stamp of \( s_i \) in \( S \). We use \( S[s_i, s_j] = \{ s_x | s_x \in S, s_i \leq s_x \leq s_j \} \) to denote the set of position stamps that fall within the range of \( s_i \) and \( s_j \) (inclusive).

The editing history of a PPS is defined by a set of revisions \( \{(S_k, M_k), 0 \leq k \leq n\} \). When a document is first created, its initial revision is an empty PPS with \( S_0 = \{0, 1\}, M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\} \). The PPS is updated by a sequence of parameterized operations that take form in one of the kinds: \( \text{ADD} \) and \( \text{HIDE} \). An \( \text{ADD} \) operation adds a new position stamp into \( S_k \) and a new mapping into \( M_k \). A \( \text{HIDE} \) operation changes the mapping in \( M_k \). Each \( \text{ADD} \) and \( \text{HIDE} \) operation \( (\delta_i) \) transforms \((S_k, M_k)\) to \((S_{k+1}, M_{k+1})\). These two operations are defined as follows:

- \( \text{ADD}(s_i, s_{i+1}, x): s_i, s_{i+1} \in S_k, x \in \Sigma^c \). It adds the character \( x \) between the character indexed by \( s_i \) and the character indexed by \( s_{i+1} \). Let \( s_{\text{new}} \in \mathbb{Q} \) be a position stamp that satisfies the constraint of \( s_i < s_{\text{new}} < s_{i+1} \). It updates \((S_k, M_k)\) to \((S_{k+1}, M_{k+1})\), where

  - \( S_{k+1} = S_k \cup \{s_{\text{new}}\} \)
  - \( M_{k+1} = M_k \cup \{s_{\text{new}} \mapsto x\} \)

- \( \text{HIDE}(s_i): s_i \in S_k \). It changes the mapping of \( s_i \) from its old value \( x \) to the null character \( \phi \). It updates \((S_k, M_k)\) to \((S_{k+1}, M_{k+1})\), where

  - \( S_{k+1} = S_k \)
  - \( M_{k+1} = M_k \cup \{s_i \mapsto \phi\} \).

The value of \( s_{\text{new}} \) must fall within the range between \( s_k \) and \( s_{k+1} \) defined in \( S_k \) in order to keep position stamps consistent to the sequential structure of the document. The algorithm that computes the value of \( s_{\text{new}} \) is called an encoding scheme. Partial persistent sequences leave the freedom of choosing a particular encoding scheme. We assume that all sites in an editing session use the same encoding scheme.

**B. Global Uniqueness of Position Stamps**

Based on the way we assign position stamps for newly added characters, each of them will obtain a unique position stamp unless several users simultaneously modify the same position. The global uniqueness of position stamps can be resumed if we allocate the distance between two position stamps for each user in advance. Suppose \( n \) sites are involved in an editing session. Given the distance \( d_{i+1} \) between two position stamps \( s_i \) and \( s_{i+1} \), it is pre-divided into \( n \) sub-ranges \( (s_i, s_i + \frac{d_{i+1}}{n}) \), \( (s_i + \frac{d_{i+1}}{n}, s_i + \frac{2d_{i+1}}{n}) \), \ldots, \( (s_i + \frac{(n-1)d_{i+1}}{n}, s_{i+1}) \). For site \( \text{site}_p \), it assigns new position stamps for its local characters within the range of \( (s_i + \frac{(p-1)d_{i+1}}{n}, s_i + \frac{pd_{i+1}}{n}) \). Taking the above example, \( \text{site}_1 \) will use the space in the range \((0, 0.5)\), \( \text{site}_2 \) will use the space in the range \((0.5, 1)\). As a result, the position stamps for \( 'a' \) and \( 'b' \) become 0.25 and 0.75 respectively. This approach uses site identifiers to break ties. We assume the global uniqueness of position stamps henceforth.

**III. COLLABORATIVE EDITOR BASED ON PPSs**

**A. Consistency Model**

Interactions of collaborative users need to be coordinated due to editing conflicts. Similar to [6], [7], [8], we consider two properties for the correctness of data consistency: *eventual consistency and intention preservation*. PPS guarantees eventual consistency of a shared document because it creates a total order for all the characters. It also satisfies intention preservation because the relative order of characters is captured by position stamps. In fact, PPS is a commutative replicated data type (CRDT) [7], which has been proved to be commutative for all concurrent operations.

**B. System Architecture Overview**

Figure 1 illustrates the system architecture for a local editor. We call the sequence data structure from the user’s perspective *logical view* and the implementation data structure from the editing system’s perspective *physical view*. The logical view of a document is represented by a buffer. When edits occur, the editor updates the document content in the buffer and refreshes the user’s view instantly. There are two queues: *local edit queue* (LEQ) and *remote edit queue* (REQ). LEQ stores local edits to be sent to other sites. REQ stores remote edits sent from other sites. Edits are broadcast through a publish/subscribe system.

A background thread, called edit processing thread (EPT), is responsible for both processing local edits and refreshing the logical view with remote edits. EPT maintains two variables in memory: *previous logical view* (prev\_lv) variable and *previous position stamps* (prev\_ps) variable, which store the buffer’s characters and their position stamps in its previous check. If EPT detects new local edits since its last check, it updates the underlying PPS following the steps of (1)-(5) in Figure 1. Otherwise, it proceeds to check REQ for the existence of any remote edits. If not empty, it processes remote edits following the steps of (6)-(8).

**C. PPS Update**

PPSs need to support efficient insertion and deletion. We choose Berkeley DB [10] to store PPSs on disk with the access method B+tree. The PPS data structure inherits the cost of B+tree. We choose the default page size for a B-tree node (4096 bytes) and represent a position stamp in 4 bytes. If we assume that a node holds \( m \) 4-byte search-key values and \((m+1)\) 4-byte pointers and the occupancy of each node being
halfway between the minimum and the maximum key-pointer pair, a three-level B+tree can hold up to tens of millions of position stamps, which is adequate for representing very large documents (up to a few megabytes). Therefore, updates to the PPS data structure can be implemented efficiently since the number of disk I/O operations will be a small number.

PPSs can incur large disk space overhead and many small disk I/Os if all position stamps are stored individually. To address these concerns, we represent the position stamps of consecutively inserted characters in a compact record. We use an integer to represent the precision part of a position stamp with an implicit integral part being an integer to represent the precision part of a position stamp consecutively inserted characters in a compact record. We use this encoding scheme, the distance between consecutive characters, and gap to represent them in a compact record.

For data set, since we are unaware of any published editing machine with Intel Pentium 4 CPU 2.80GHz and 1GB RAM. We process all documents in sequence. For each document, we use its revisions to update a PPS in their chronological order. The first experiment measures the total time to process a given number of edits. In Figure 2, the “File” system uses several times larger disk space than others because it stores redundant data for overlapped content. The ratio becomes larger as the number of revision increases because documents have the tendency of getting stabilized after certain amount of revisions. Both RCS and PPS can therefore achieve better compression ratio since they only store delta changes. The disk space measurement for PPS does not include the part for maintaining the mapping between consecutive re-balanced PPSs because the other two systems only maintain revision histories, and we want to compare them on the same feature.

A. Disk Space Consumption

We compare PPSs on managing document revision histories with a file-based system and RCS [13]. The file system stores each revision as individual copies while RCS stores the delta difference between revisions. In Figure 2, the “File” system uses several times larger disk space than others because it stores redundant data for overlapped content. The ratio becomes larger as the number of revision increases because documents have the tendency of getting stabilized after certain amount of revisions. Both RCS and PPS can therefore achieve better compression ratio since they only store delta changes. The disk space measurement for PPS does not include the part for maintaining the mapping between consecutive re-balanced PPSs because the other two systems only maintain revision histories, and we want to compare them on the same feature.

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>STATISTICS OF OUR EXPERIMENTAL DATA SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of documents</td>
<td>1,941</td>
</tr>
<tr>
<td>No. of revisions</td>
<td>275,387</td>
</tr>
<tr>
<td>Avg. revisions</td>
<td>141</td>
</tr>
<tr>
<td>Avg. document size</td>
<td>8k</td>
</tr>
<tr>
<td>Avg. edit length</td>
<td>10</td>
</tr>
</tbody>
</table>

B. Updating Cost From Logical View to Physical View

We design two experiments to measure the mapping cost from the logical view to the underlying PPS (Step 1-5 in Figure 1). We process all documents in sequence. For each document, we use its revisions to update a PPS in their chronological order. The first experiment measures the total time to process a given number of edits. In Figure 3, the total processing time increases almost linearly as the number of edits increase and takes less than a few second to complete hundreds of edits. The second experiment measures the total time to process edits at certain edit length. In Figure 4, it takes seconds to process edits with characters larger than a few kilobytes.
both experiments, the small processing time is sufficient to cope with the speed of human edits.

C. Updating Cost From Physical View to Logical View

The EPT thread described in Section III-B periodically refreshes the logical view with remote edits. During its execution, it does a sequential traversal of the PPS and concatenates all visible characters. We measure this cost by evaluating total processing time for traversing a given length of PPS in terms of the number of position stamps. The cost is equal to that of traversing all leaf nodes in its B+tree. The result, Figure 5, shows that the traversal cost is at around tens of milliseconds and does not change much as the length of PPS increases. Based on our experiment data, a typical PPS of hundreds of thousands of position stamps has only hundreds of keypoint pairs in general in its B+tree due to the optimization techniques described in Section III-C. Therefore, a large PPS can be traversed very quickly.

V. RELATED WORK

Several index structures have been proposed for creating unique identifiers for unstructured documents. One category relies a centralized server for assigning global unique identifiers to all the characters, e.g., TeNDax [14]. These tools are suitable for non real-time collaborative editing scenarios since the central server can introduce significant delay to local edits. The other category relies on local editors to generate globally unique identifiers. Oster et al. [6] proposed to generate global unique identifiers for locally inserted characters based on its site identifier and a local counter and maintains the structural information in a linked-list-like data structure, which is problematic when representing a large sequence of characters on disk and doing sequential retrieval. In both [7] and [8], the authors proposed to use concatenated labels to uniquely identify a character. Since the paths can become arbitrary long, it can incur high overhead for storage and update.

VI. CONCLUSION

We address the problem of maintaining data consistency and tracking changes of documents by proposing a new data structure, partial persistent sequence (PPS). We believe that it has a variety of applications. For example, position stamps can be used for making transclusions of parts of documents to help manage data lineage, a critical issue in scientific data documentation for obtaining an accurate picture of data sources. We will explore these possibilities as future work.

ACKNOWLEDGMENT

This research has been partially funded by National Science Foundation grants ENG/EEC-0335822, CISE/CNS-0646430, CISE/CNS-0716484, AFOSR grant FA9550-06-1-0201, NIH grant U54 RR 024380-01, IBM, Hewlett-Packard, Wipro Technologies, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

REFERENCES