PARTIAL PERSISTENT SEQUENCES AND THEIR APPLICATIONS TO COLLABORATIVE TEXT DOCUMENT EDITING AND PROCESSING

A Thesis
Presented to
The Academic Faculty

by

Qinyi Wu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2011

PARTIAL PERSISTENT SEQUENCES AND THEIR APPLICATIONS TO COLLABORATIVE TEXT DOCUMENT EDITING AND PROCESSING

Approved by:

Professor Calton Pu, Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Calton Pu, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Ling Liu
School of Computer Science
Georgia Institute of Technology

Professor Leo Mark
School of Computer Science
Georgia Institute of Technology

Professor Shamkant B. Navathe
School of Computer Science
Georgia Institute of Technology

Professor Lakshmish Ramaswamy
Department of Computer Science
University of Georgia

Date Approved: June 30, 2011

To my family

ACKNOWLEDGEMENTS

If I am asked to name a few number of people that has changed and shaped me, my advisor, Prof. Calton Pu, immediately comes to mind. I sincerely appreciate his guidance both as a researcher and as a mentor. As a researcher, Calton respects me as a young researcher by giving me independent thinking space and time to explore new things. As a mentor, Calton helps me grow in terms of professional maturity through his advice on my long-term career goals. I fully respect his advice because he always wants the best for his students. With this level of trust and understanding, I value the time we worked together during those tense moments when my research got stalled and the future looked unclear. His underlying gentleness and dedication to research will continue to have a significant impact on my future.

Thanks to members of my thesis committee: Ling Liu, Leo Mark, Shamkant B. Navathe, and Lakshmish Ramaswamy. I would like to thank Ling for her organization on DISL meetings. Through this supporting group and her advice, I have significantly broadened my knowledge scope and have become more comfortable and confident in doing research presentations. I would like to thank Sham for his consistent encouragement and recognition of my research. Thanks to Leo for watching each milestone of my pursuit of the Ph.D. degree. I felt very happy upon receiving his congratulation email for my award on the College of Computing Research Day. Doing a Ph.D. is long and sometimes lonely journey. Such an encouragement makes a difference. I would like to thank Lakshmish for his sharp and insightful suggestion for my research, which led to our successful collaboration on a new topic.

The friendship, companionship, and support of my colleagues at Gatech would be hard to replace. Special thanks to Bhuvan Bamba, Rocky Dunlap, Binh Han,

Danesh Irani and Qingyang Wang for being around in the lab. Bhuvan brought so much energy into the lab. Rocky is my best English tutor. Binh were my study buddy. Danesh is a valuable colleague to work with. Qingyang is such a good friend to hang around and discuss about research.

More importantly, I would like to acknowledge the unconditional love and support from my mom, my husband, my daughter, my sister, my grandmother, and my parents-in-law for their affection and wisdom. Finally, thanks to my dad. Even if you are in another world, your perseverance and dedication to your dream are always on the way of my life journey and lighting the road forward.

TABLE OF CONTENTS

DEDICATION . . . . **iii**

ACKNOWLEDGEMENTS . . . . **iv**

LIST OF TABLES . . . . **x**

LIST OF FIGURES . . . . **xi**

SUMMARY . . . . **xiii**

I INTRODUCTION **1**

1.1 Motivation 1

1.2 Dissertation Contributions 2

1.3 Organization of the Dissertation 4

II RELATED WORK **5**

2.1 Persistent Data Structures 5

2.2 Data Consistency Control in Collaborative Document Editing Systems 7

2.3 Document Provenance 10

2.4 Text Documents Labeling Schemes 11

2.5 Transaction Time Management in Temporal Databases 11

III PARTIAL PERSISTENT SEQUENCES **15**

3.1 Definition of PPSs 15

3.2 Two Levels of View on Collaborative Text Documents 17

3.2.1 Mapping from Physical View to Logical View 18

3.2.2 Mapping from Logical View to Physical View 19

3.3 Representing the Revision History of Documents Based on PPSs . . 21

3.3.1 Different Design Options 21

3.3.2 A Hybrid Approach 24

3.3.3 Properties of Delta Change Operator $\Box$ 26

3.3.4 Operations on a PPS View Range 30

3.4 Summary 35

IV DATA CONSISTENCY CONTROL IN REAL-TIME COLLABORATIVE TEXT DOCUMENT EDITING SYSTEMS 37

4.1 Motivation 37

4.2 DDP Consistency Model and View Synchronization Strategy 40

4.2.1 DDP Consistency Model 40

4.2.2 View Synchronization Strategy 41

4.3 System Implementation Based on PPSs 43

4.3.1 System Architecture Overview 43

4.3.2 PPS Update 45

4.3.3 Global Uniqueness of Position Stamps 47

4.3.4 PPS Re-initialization 48

4.4 Experiments 50

4.4.1 Experiment Setup 50

4.4.2 Disk Space Consumption 51

4.4.3 Updating Cost From Logical View to Physical View 53

4.4.4 Updating Cost From Physical View to Logical View 54

4.4.5 PPS Scalability 54

4.4.6 PPS Re-initialization 56

4.5 Summary 57

V MODELING AND IMPLEMENTING COLLABORATIVE TEXT DOCUMENT EDITING SYSTEMS WITH TRANSACTIONAL TECHNIQUES 59

5.1 Motivation 59

5.2 Overview of Collaborative Document Editing Systems 61

5.2.1 Existing Collaborative Editing Systems 61

5.2.2 Commentary of Existing Collaborative Editing Systems . . . 63

5.3 A Transactional Framework for collaborative editing Systems 64

5.3.1 Programming Interfaces 65

5.3.2 Synchronization Protocol Between Client and Server 67
5.4 Specifying Collaborative Editing Systems 69
5.5 Implementing the Synchronization Protocol Based On PPSs 71
5.5.1 Enforcement of the Synchronization Protocol 72
5.5.2 Revert Handling 74
5.5.3 Implementation Issues for PPSs 75
5.6 Collaborative Editing System Prototype 76
5.6.1 Oracle Berkeley DB High Availability Infrastructure 76
5.6.2 System Architecture 76
5.6.3 Implementation Modules 77
5.7 Summary 79

VI FINE-GRAINED DOCUMENT PROVENANCE MANAGEMENT ON COLLABORATIVE TEXT DOCUMENTS 81

6.1 Motivation 81
6.2 Fine-grained Document Provenance Queries 83
6.2.1 Classifying Fine-grained Document Provenance Queries from the perspective of the temporal dimension 83
6.2.2 Processing Fine-grained Document Provenance Queries based on PPSs 84
6.3 System Implementation 86
6.4 Performance Impact due to Size of PPS View Ranges 88
6.5 Experiments 91
6.5.1 Experiment Setup 91
6.5.2 Evaluating a Snapshot Document Provenance Query: Getting Authorship of Text at Selected Versions 92
6.5.3 Evaluating a Delta-change Document Provenance Query: Getting Who Modified Whose Work at Selected Versions 93
6.5.4 Evaluating Document Provenance Queries for Entire Revision History of Documents 95
6.5.5 Disk Space Usage 95
6.5.6 Document Loading Time 96

6.6 Summary 97

VII CONCLUSIONS AND FUTURE WORK 99

7.1 Dissertation Conclusion 99

7.2 Future Work 101

7.2.1 Partial Persistent Sequences 101

7.2.2 Data Consistency Control in Collaborative Text Document Editing Systems 102

7.2.3 Fine-grained Document Provenance Queries 103

VITA 105

REFERENCES 106

LIST OF TABLES

1 Statistics of sampled Wikipedia data set for evaluating the performance of the PPS-based real-time collaborative editor 51

2 Edit length distribution in the sampled Wikipedia data set 55

3 Statistics of three Wikipedia articles 92

LIST OF FIGURES

1 A simple PPS example . . . 15

2 Mapping between two levels of view . . . 19

3 Organize the revision history of a document based on PPSs . . . 22

4 Organize the revision history of a document based on PPSs by considering the REVERT operation . . . 23

5 Store the full list of visible position stamps for every version of a document . . . 23

6 Represent the revision history of a PPS in a hybrid form . . . 25

7 A real-time collaborative editing scenario . . . 38

8 System Architecture for the PPS-based real-time collaborative document editing system . . . 43

9 An example of splitting a compact record after an insert . . . 48

10 Total disk space consumption for the sampled data set . . . 52

11 Updating cost from logical view to physical view for a given number of edits . . . 52

12 Updating cost from logical view to physical view for edits at different length . . . 52

13 Updating cost from physical view to logical view for PPSs at different length . . . 52

14 Throughput measure at the server site in the PPS-based real-time collaborative editing system . . . 56

15 Re-initialization cost for the PPSs . . . 57

16 Examples of synthesized code. a)Check-in/check-out; b)Block-exclusive; c)Update-anywhere-anytime; d)Read-from . . . 69

17 The algorithm for validating $Accept^{\theta}$ for transaction t . . . 73

18 System architecture . . . 77

19 Organize the revision history of a versioned document based on delta changes . . . 82

20 System Architecture . . . 86

21 Disk space usage under different configurations for the size of PPS view ranges . . . 88

22 Impact of the size of PPS view ranges . . . 89

23 Query processing cost due to the size of PPS view ranges . . . 91

24 Compare querying cost for a snapshot document provenance query for three Wikipedia articles . . . 93

25 Compare querying cost for a delta-change document provenance query on three Wikipedia articles . . . 94

26 Compare querying cost on three Wikipedia articles . . . 94

27 Disk space usage for Wikipedia articles with different number of versions: a) disk space usage; b) ratio . . . 96

28 Document loading time for Wikipedia articles with different number of versions . . . 97

SUMMARY

In a variety of text document editing and processing applications, it is necessary to keep track of the revision history of text documents by recording changes and the metadata of those changes (e.g., user names and modification timestamps). The recent Web 2.0 document editing and processing applications, such as real-time collaborative note taking and wikis, require fine-grained shared access to collaborative text documents as well as efficient retrieval of metadata associated with different parts of collaborative text documents. Current revision control techniques only support coarse-grained shared access and are inefficient to retrieve metadata of changes at the sub-document granularity.

In this dissertation, we design and implement partial persistent sequences (PPSs) to support real-time collaborations and manage metadata of changes at fine granularities for collaborative text document editing and processing applications. As a persistent data structure, PPSs have two important features. First, items in the data structure are never removed. We maintain necessary timestamp information to keep track of both inserted and deleted items and use the timestamp information to reconstruct the state of a document at any point in time. Second, PPSs create unique, persistent, and ordered identifiers for items of a document at fine granularities (e.g., a word or a sentence). As a result, we are able to support consistent and fine-grained shared access to collaborative text documents by detecting and resolving editing conflicts based on the revision history as well as to efficiently index and retrieve metadata associated with different parts of collaborative text documents.

We demonstrate the capabilities of PPSs through two important problems in collaborative text document editing and processing applications: data consistency control and fine-grained document provenance management. The first problem studies how to detect and resolve editing conflicts in collaborative text document editing systems. We approach this problem in two steps. In the first step, we use PPSs to capture data dependencies between different editing operations and define a consistency model more suitable for real-time collaborative editing systems. In the second step, we extend our work to the entire spectrum of collaborations and adapt transactional techniques to build a flexible framework for the development of various collaborative editing systems. The generality of this framework is demonstrated by its capabilities to specify three different types of collaborations as exemplified in the systems of RCS, MediaWiki, and Google Docs respectively. We precisely specify the programming interfaces of this framework and describe a prototype implementation over Oracle Berkeley DB High Availability, a replicated database management engine. The second problem of fine-grained document provenance management studies how to efficiently index and retrieve fine-grained metadata for different parts of collaborative text documents. We use PPSs to design both disk-economic and computation-efficient techniques to index provenance data for millions of Wikipedia articles. Our approach is disk economic because we only save a few full versions of a document and only keep delta changes between those full versions. Our approach is also computation-efficient because we avoid the necessity of parsing the revision history of collaborative documents to retrieve fine-grained metadata. Compared to MediaWiki, the revision control system for Wikipedia, our system uses less than 10% of disk space and achieves at least an order of magnitude speed-up to retrieve fine-grained metadata for documents with thousands of revisions.

CHAPTER I

INTRODUCTION

1.1 Motivation

Collaborative text documents are modified by multiple users over time. To handle data consistency issue and obtain precise knowledge about their evolution, it is necessary to keep track of the revision history of collaborative text documents by recording changes of content and the metadata of those changes (e.g., username and timestamp). For instance, source control systems maintain delta changes in source code files in order to undo accidental edits. In another example, the revision history of Wikipedia pages is maintained and analyzed to combat vandalism [125, 126] or to recognize contributions of Wikipedia authors [27]. In fact, the requirement of tracking changes and their metadata for collaborative documents can be found in a variety of text document editing and processing applications such as version control systems [12, 43], wikis [19, 21], real-time collaborative editing systems [58, 87, 113], document-driven workflows [69], and deep document knowledge discovery applications [27, 67, 121].

Revision control is the only technique that has been developed to track changes and metadata of these changes for text documents [43]. Current revision control systems (e.g., RCS [117], CVS [12], Subversion [24]) manage changes of documents by maintaining a list of chronologically ordered versions. For each of these versions, these systems record necessary metadata such as user name and modification timestamp. During collaboration, these systems use some kind of locking mechanisms to detect editing conflicts of users. Revision control systems have two limitations when applied to collaborative text document editing and processing. First, they are not suitable for the recent Web 2.0 collaborative applications in which users can simultaneously

modify different parts of a shared document. The centralized locking mechanism imposes serious performance limitations for those applications. Second, revision control systems track metadata for changes at the document level. As a result, it is efficient to retrieve metadata (e.g. who committed which version at what time), but not efficient to query metadata at a sub-document granularity. In the latter case, we have to develop tools to parse the revision history of collaborative text documents and analyze their delta changes in order to collect fine-grained metadata. For documents with a long revision history, the parsing and analyzing cost becomes expensive.

1.2 Dissertation Contributions

To address the problems described in the previous section, this dissertation makes the following contributions:

In the first contribution, we design partial persistent sequences (PPSs) to track changes and the metadata of changes for collaborative text document editing and processing applications. We precisely define PPSs and establish the mapping between traditional documents editing operations and PPS editing operations. With the mapping, we are able to accurately record changes of a document for its revision history. In addition, we develop a hybrid approach to balance the tradeoff between disk space usage and efficiency of accessing the revision history of collaborative text documents.

In the second contribution, we use PPSs to track data dependencies between document editing operations and design a relaxed consistency model suitable for real-time collaborative editing systems. Compared to existing consistency models, our consistency model is more flexible in that it allows users to simultaneously work on different parts of a shared document without being constrained by causal dependencies [57]. We also introduce a view synchronization strategy for the relaxed consistency model

and prove its correctness. To demonstrate the practicality of our approach, we implement a prototype system and evaluate the performance of our system in terms of both disk space usage and access time for document updates and retrievals.

In the third contribution, we design a flexible transactional framework for the development of various collaborative editing systems. Our framework combines transactional techniques from the database technologies and the capabilities of PPSs in tracking data dependencies between editing operations. The generality of our framework is tested by its capabilities of specifying three types of collaborative editing systems RCS [117], MediaWiki [17], and Google Docs [15]. We further test its generality by using this framework to specify the behavior of a new type collaboration that is derived by combining features of Google Docs and the acceptance test in handling conflict reconciliation in replicated database management systems [64]. In addition, our framework has the advantage of saving the cost for infrastructure development because it can be implemented on the top of replicated database management systems, as demonstrated by a prototype implementation over Oracle Berkeley DB High Availability [18].

In the fourth contribution, we design both disk-economic and computation-efficient techniques to index document provenance information at fine granularities. Our approach is disk efficient because we only store a few full versions of a document and store the delta changes for the rest of the versions. Our approach is also computation-efficient because we avoid the necessity of parsing the revision history of collaborative text documents. We have built a system to manage document provenance information for millions of Wikipedia articles and compare its performance with MediaWiki, the database engine for Wikipedia. The experiments show that our system uses less than 10 percent of the disk space used by MediaWiki and achieves at least an order of magnitude speedup for common provenance queries.

1.3 Organization of the Dissertation

This dissertation first reviews the related work in Chapter 2. It provides background knowledge for persistent data structures, data consistency control in collaborative editing systems, document provenance, document labeling schemes, and transaction time management in temporal databases.

In Chapter 3, we define PPSs and establish the mapping between document editing operations and PPSs editing operations. We then introduce a hybrid approach that balances disk space usage and the efficiency of accessing the revision history of collaborative text documents.

In Chapter 4, we explain the challenges of data consistency control in real-time collaborative editing systems and define a relaxed consistency model to support more flexible real-time collaborative editing scenarios. We explain how to use PPSs to represent data dependencies between different editing operations and design a view synchronization strategy to resolve editing conflicts in real-time collaborations.

In Chapter 5, we give an overview of existing collaborative systems and discuss their potential improvements. We then describe the programming interfaces of a transactional framework and its synchronization protocol for data consistency enforcement based on PPSs. After that we illustrate the flexibility of our framework by modeling several representative collaborative models. Finally, we describe a prototype implementation over Oracle Berkeley DB High Availability and evaluate the performance of our prototype.

In Chapter 6, we explain the limitations of current revision control systems to handle fine-grained document provenance queries. We classify common document provenance queries based on the temporal dimension and provide the rule of thumb to organize the revision history of collaborative text documents based on PPSs. Finally, we describe the implementation of our system and evaluate its performance on disk space usage and query processing cost.

CHAPTER II

RELATED WORK

2.1 Persistent Data Structures

The concept of persistent data structure was first systematically analyzed by Driscoll et. al. [55]. According to their definition, "*A data structure is persistent if it supports access to multiple versions. The structure is partially persistent if all versions can be accessed but only the newest version can be modified, and fully persistent if every version can be accessed and modified*".

In order to access the revision history of a text document, we design the partially persistent form of the *sequence* data structure [46] because a text document is naturally represented as a sequence. Each element in the sequence may be a character, a word, or a line. The choice of granularities depends on different application domains. In revision control systems, the granularity is a line. In wikis, the granularity is a word [62]. In real-time collaborative editing systems, the granularity is a character. To facilitate our discussion about accessing the revision history of text documents without being distracted by the choice of element granularities, we treat a text document as a sequence of *items*. Each item is the smallest indivisible unit when we track the changes in the document. For example, an author fixed the typo error by changing "Compl**i**ment" to "Compl**e**ment". If the indivisible unit is at the word level, we say the word "Compliment" was deleted; a new word "Complement" was inserted. If the indivisible unit is at the character level, we say the character "i" was deleted; a new character "e" was inserted. When we give examples for sequences, we use the Greek letters to represent their items such as $[\alpha, \beta, \gamma, ...]$.

A sequence is *ephemeral* in the sense that any modification always destroys the

previous version. For instance, if we modify the sequence from $[\alpha, \beta]$ to $[\beta, \gamma]$, we lose the information for the deleted item α. In order to access the previous versions of a sequence, we design the partially persistent form of the sequence data structure named *partial persistent sequence*(PPS). The PPS data structure is partially persistent because it always preserves the previous version of a modified sequence and only the latest version of the sequence can be modified. This dissertation does not address fully persistent sequences and leave them as our future work.

Various persistent data structures have been proposed in the literature, including stacks, queues, and search trees [55, 81]. There are two major techniques for making a data structure partially persistent: *fat node method* and *node-copy method*. The idea of fat node method is to keep all changes happening to a node in an array of timestamp/value pairs without erasing old values. These timestamp/value pairs are sorted by their timestamps. To retrieve the value of the node at a particular time, a binary search is used to locate the right value. As a node may save an arbitrarily number of pairs, it can become very "fat", which would slow down the performance of access time by $O(log(n))$ where n is the number of pairs in a node. The node-copy method is introduced to solve this problem by controlling the number of pairs in a node. When a node becomes too fat, the node is copied to contain the latest value. All predecessors of the copied node also store pointers to the new node. The node-copy method has an $O(1)$ amortized bound on the number of nodes copied and update cost. Theoretically, we could reuse these early techniques to make sequences partially persistent because a sequence can be represented as a search tree. Items are ordered according to their offsets in the sequence. Each item in the sequence corresponds to a node in the tree. Insert and delete operations on sequences could be mapped to node insert and delete operations in the search tree. However, these early techniques are memory-based in that they require the entire revision history of a search tree represented in memory in order to update pointers to maintain the

tree structure. For text documents with long revision histories, we have to seek disk-resident techniques for making sequences partially persistent.

To design disk-resident techniques for partially persistent sequences, we have three options. The first option, called the full-version approaches used in MediaWiki [17], stores the full content of a sequence for every version. This option uses significant amount of disk space since it keeps the same number of copies of an item as the number of versions containing the item. This option performs inefficiently for large documents with only minor changes between consecutive versions. The second option is called delta-change approach as used in RCS [117]. It keeps all the changes in a log and chains them in their chronological order. To retrieve a particular version, we start with the first version and apply those delta changes until we reach the requested version. This option solves the problem of the full-version approach, but at the cost of applying delta changes on a long delta-change list. The third option is called the hybrid approach used by Subversion [42]. In this option, we store multiple full versions of a sequence at different points. For versions between these points, only delta changes are saved. Our design on PPSs use the hybrid approach to balance the tradeoff between disk space usage and access cost.

2.2 Data Consistency Control in Collaborative Document Editing Systems

Collaborative editing systems support geographically distributed users to work on a shared document. We have observed a wide spectrum of collaborations among these systems. At one end of the spectrum are version control systems that support only restricted collaboration, such as CVS [12], RCS [117], and Subversion [42]. At the other end of the spectrum are those "liberal" collaborative editing systems that support highly interactive collaboration, such as Gobby [2], Google Docs [15], SubEthaEdit [8], and Coword [133].

The implementation of current collaborative document editing systems are ad

hoc in the sense that they only cover a subset of interactions found in collaborative environments. We aim to design a unifying framework to specify and reason the entire spectrum of collaborations. Our work borrows ideas from advanced transaction models and their unifying frameworks. Our work is also influenced by the research on data consistency control in real-time collaborative editing systems. Below we overview each of them.

Advanced Transaction Models The classical transaction model guarantees the ACID properties (i.e., atomicity, consistency, isolation, and durability). This model is mainly used in OLTP applications. More and more nontraditional applications such as CAD/CAM, business processes have found that the classical transaction model is too rigid. As a result, many extended transaction models were developed to establish a theoretical foundation for specifying correctness in cooperative applications [31, 38, 68, 88, 102] to model open-ended and long-running activities. Our work makes it one step further to adapt advanced transaction models to collaborative document editing systems because PPSs make it possible to define transaction boundaries and editing conflicts in collaborative document editing systems. Especially, our work on data consistency control adopts two techniques from the earlier work. First, the handling of revert follows the compensation technique in Sagas [63]. Second, the introduction of a pivot-point to define an irreversible reference point follows the technique of handling backward recovery of transaction processing proposed by Mehrotra et.al. [89].

Our work also benefits from ACTA, the first formalizable framework developed to characterize the whole spectrum of collaborations [38]. Within the framework, two primitives to specify commit/abort dependencies between transactions were proposed as well as the interplays between read and write set of transactions. ACTA demonstrates its capabilities by specifying several advanced transaction models such as Nested Transaction [90] and Split-Join Transactions [102]. Our transactional framework for collaborative document editing systems follows the same observation as

ACTA that it is possible to specify and reason different kinds of collaborations by using primitives to specify the interaction between collaborative editing transactions. Compared to ACTA, our transactional framework is specialized for document editing applications to handle editing operations such as release of user changes, local undo, and global undo.

Data Consistency Control in Real-time Collaborative Editing Systems Several consistency models have been proposed in the literature of computer supported collaborative work. The first well-accepted consistency model for collaborative editing systems consists of two properties *convergence* and *precedence preservation* [57]. After its proposal, people realized that an editing system could still produce an execution history leading to a document state not intended by any of the users. The reason is that the definition of the convergence property does not specify correctness criteria for the transformation between successive document states. To disallow these abnormal execution histories, Sun et. al. [113] added an additional property *intention preservation* to the earlier consistency model. This property states that the execution effect of an operation is preserved at all collaboration sites. This consistency model receives wide recognition because of its first attempt to define a consistency criterion for the intermediate document states of a co-authored document. Following this work, two other consistency models were proposed to clarify the intention preservation property. The CSM consistency model [85] defines the intention preservation effect in terms of a total order of characters based on their insertion positions. The WOOT consistency model [99] has a similar idea. In parallel to the development of data consistency models, many data consistency control algorithms have been proposed [57, 98, 99, 103, 113, 114]. Most of them are a variant of operational transformation [57], an optimistic distributed concurrency control algorithm.

None of the above algorithms adopt transactional techniques because of the general concern that serializability is too restricted to be the correctness criteria for

cooperative applications; the performance of synchronized distributed concurrency control algorithms such as two-phase commit are slow for human-centered editing applications. However, RDBMSs are going through fundamental changes in recent ten years [111]. Several famous replicated RDBMSs such as Dynamo [50] or PNUTS [44] all support relaxed consistency models in order to satisfy the availability and the performance of their applications. Based on PPSs, we adapt transactional techniques from databases to building a framework to model a variety of collaborations including real-time collaborative editing systems.

2.3 Document Provenance

Data provenance in its most general form describes the derivation history of data from its original source [71]. The research on managing data provenance has been conducted extensively in scientific workflows where the primary focus is on how to develop efficient techniques to manage metadata related to process of experiment workflows, such annotations, programs, and notes [35, 110]. For this dissertation, we particularly focus on data provenance management for editing processes of documents. We use the term *document provenance* to describe how a document was updated over time. In multi-user editing systems, the importance of document provenance was recognized two decades ago in improving the awareness of group activities [54]. Nowadays, enterprise have started the efforts of improving wiki sustainability by using document provenance to recognize contributions of employees [48, 66]. In general, applications exploits provenance in tasks ranging from document processing in business processes [69] to deep knowledge discovery for text documents [27, 49]. Although fine-grained document provenance is valuable, storing and querying provenance can be expensive. Current revision control systems manage provenance data at the document level, which is too coarse-grained to retrieve provenance data at a finer granularity such as a word or a sentence. The focus of this dissertation is to design

efficient techniques to track provenance information at the sub-document granularity.

2.4 Text Documents Labeling Schemes

Our work on PPSs is closely related to the literature of text document labeling schemes in the sense that both PPSs and the early work focus on how to create unique and persistent identifiers for the items of a text document. Treedoc [100] and Logoot [124] use path-based labeling scheme to assign unique identifiers for items of a document. For the path-based labeling schemes, the space and computation overhead are always major concerns, especially when we need to create identifiers for items at a very small granularity (e.g., a word). TeNDax [84] is another work that assigns unique identifiers for items in a document. The difference is that TeNDax uses a monotonically increased counter to assign identifiers for new items. As a result, the identifiers are not ordered. To determine the right position of an item, the identifier of its previous item and the identifier of its next item need to be explicitly maintained in a linked-list-like data structure. Text document labeling schemes by themselves do not address how to access previous versions of documents. We choose rational numbers as our labeling domain due to its space and computation efficiency.

2.5 Transaction Time Management in Temporal Databases

Temporal database management is an active field of research on managing time-varying data in applications that are temporal in nature such as accounting, banking, and scheduling [74]. In general, temporal databases consider three types of time: *valid time*, *transaction time*, and *user-defined time*. According to the glossary of temporal database concepts [72], "*Valid time of a fact is the time when the fact is true in the modeled reality. Transaction time of a database fact is the time when the fact is current in the database. User-defined time is an un-interpreted attribute domain of date and time.* Different temporal data models were proposed to address these time dimensions [104]. We particularly review the DM/T data model [73, 75]

that was proposed to address the type of transaction time in temporal databases because this dissertation manages transaction time for document editing and processing applications. Even though our work targets the unstructured data model, which is different from the relational data model in the early work, we face similar research issues including change history representation, differential computing, and timeslice computing over change histories.

The DM/T data model organizes the updating history of an ordinary database relation in an append-only relation, called the *base relation*. Each tuple in the base relation corresponds to an insert, a delete, or an update database operation. The tuple records the type of the operation, the transaction timestamp, and the changes. The base relation can be queries using a special timeslice operator to compute any previous state of the original database relation. To guarantee the efficiency of computing both the past and recent timeslices, an I-tree data structure is designed to index the transaction time attribute in the base relation. An I-tree is a sparse B+-tree designed for append-only relations. Given a transaction timestamp, we can use an I-tree to quickly locate the disk page containing the transaction record within logarithmic cost. The cost of the timeslice operator can be further reduced if we cache previously computed timeslices and apply either an incremental or a decremental computation to the cached timeslices depending on the number of pages that must be read to process a request.

Similar to the DM/D model, we have a timeslice-like operator to compute a previous state of a document. We also maintain differential changes to conduct either incremental or decremental computation to calculate timeslices. However, we face the new challenge of creating unique, persistent, and ordered identifiers for the items of a text document. In a relational database, tuples are either indexed by a given primary key or are treated the same if they are equal pair-wise on all the attributes. In addition, tuples in a relational database are defined over bags and therefore there are

no ordering relationships between tuples. However, all these characteristics are not available in documents, which are defined over the unstructured data model. First, there is no given primary key for items of documents. This creates problems in differential computing because we need to associate update operations for a particular item. Second, we could not depend on the content of an item to create a unique key because two items with the same content but located at different offsets of a document are treated as different items. Finally, items of a document have spatial relationships conforming to the sequential structure of the document. This ordering is required for many document queries, e.g., we want to create a range locking on a paragraph to keep other users from simultaneously modifying the same area in a multi-user concurrent editing scenario. Part of the contribution of this dissertation is to create unique, persistent, and ordered identifiers over the domain of rational numbers. Note that we could not choose the common solutions of using a counter or creation timestamps of items because identifiers created this way are not ordered. By using rational numbers, we essentially translate an unstructured document into a list of key/value pairs and open the door of bringing temporal database technologies for document editing and processing.

Compared to the DM/T data model, this dissertation focuses on a narrower scope in that our techniques are specialized for key/value based data model, not the relational data model. As a result, our techniques are applicable to retrieve metadata of a particular item, but not support advance queries such as selection, projection, and join on data related to different items. Another difference is that we use a static approach to materialize the timeslices of a document at strategic points of the revision history in order to reconstruct the state of the document at about constant time. In our static approach, we maintain an array of key/value pairs with the key being the timestamp of a timeslice and the key being a pointer to the materialized data on disk. To construct the state of the document at a particular time, we do

a binary search on the array based on the keys to choose the closest timeslice for the requested document state. Based on our experiments, our current design suffices the requirements of document versioning systems that manage documents with tens of thousands of revisions and support key/value based metadata lookup. More sophisticated techniques are required if we want to index very large document revision histories and support rich metadata document queries. We will study the strength of different advanced data structures to improve transaction record indexing such as I-trees, MB-trees [60] and the AP trees [107] and query processing and optimization techniques such as state transition network [75] as our future work.

CHAPTER III

PARTIAL PERSISTENT SEQUENCES

3.1 Definition of PPSs

Conceptually, a PPS contains a list of items indexed by *rational numbers*. Since rational numbers are defined over a dense domain, we are able to index the items with *unique*, *persistent*, and *ordered* indexes. We call these rational indexes as *position stamps*. Figure 1 shows a simple PPS example. At Step 1, the PPS has four items whose position stamps are 0, 0.2, 0.8, and 1 respectively. ϕ is a special item used to mark the beginning and the end of a PPS. It is invisible to the represented document (as indicated by the gray color). At Step 2, an insert happens between α and γ. As a result, a new item β is inserted with a new position stamp 0.5. At Step 3, a delete happens on item α. The PPS handles the delete by marking the item invisible instead of physically removing it from the data structure.

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

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

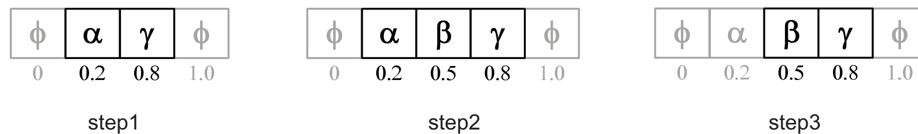


Figure 1: A simple PPS example

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 $S_0 = \{0, 1\}$, $M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\}$. The PPS is updated by a sequence of parameterized operations that take form in one of the kinds: ADD and $HIDE$. An ADD operation adds a new position stamp into S_k and a new mapping into M_k. A HIDE operation changes the mapping in M_k. Each ADD and HIDE operation transforms (S_k, M_k) to (S_{k+1}, M_{k+1}). These two operations are defined as follows:

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

  $S_{k+1} = S_k \bigcup \{s_{new}\}$

  $M_{k+1} = M_k \bigcup \{s_{new} \mapsto x\}$

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

  $S_{k+1} = S_k$

  $M_{k+1} = M_k - \{s_i \mapsto x\} \bigcup \{s_i \mapsto \phi\}$.

A HIDE operation does not change the set of position stamps in a PPS. But an ADD operation will add a new element, s_{new}, into the set. The value of s_{new} must fall within the range between s_k and s_{k+1} defined in S_k. This is important in

order to maintain the uniqueness of each position stamp and the right position of the newly inserted item in S_{k+1}. The algorithm that computes the value of s_{new} is called an *labeling scheme*. Partial persistent sequences leave the freedom of choosing a particular labeling scheme. For example, we can compute s_{new} in dyadic rational numbers, which halving the interval between s_i and s_{i+1} into two, or in Farey rational numbers, which choose mediant of s_i and s_{i+1} [119].

3.2 Two Levels of View on Collaborative Text Documents

From a user's perspective, a document consists of a sequence of items. If a new item is inserted, a portion of the sequence will be shifted right to vacate the space for the new item. Correspondingly, if a item is deleted, a portion of the sequence will be shifted left to reclaim the space. On the other hand, the underlying editing system keeps the items of the document in a selected data structure, such as an array and a linked list [13]. 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*. Characters in the logical view are identified by their offsets to the first item. These offset-based identifiers are volatile in that they keep changing with new edits. To create persistent identifiers for items, we need to maintain their identities in the physical view and make them persistent on disk. In this section, we define precisely how we do the mapping between the two views.

At the logical view, a document is defined by a sequence of items $E = \langle c_i \in 2^{\Sigma^c}, 1 \leq i \leq n, n \in \mathbb{N} \rangle$, where Σ^c is the alphabet of the document. $\langle\rangle$ denotes an empty sequence, $E[i]$ the i-th element of E, $E[i,j]$ the subsequence $\langle c_i, c_{i+1}, ..., c_j \rangle$, $|E|$ the length of E.

When a document is first created, it is initialized to an empty sequence of items $E_0 = \langle\rangle$. Each INSERT or DELETE operation transforms the document from E_i to E_{i+1}. We use E_k to denote the revision of E as transformed by a sequence of

$INSERT$ and $DELETE$ operations of cardinality k. An INSERT operation adds a new item into the sequence. A DELETE operation removes a item from the sequence. These two operations are defined as follows:

- $INSERT(p, x)$: $p \in \mathbb{N}$, $x \in \Sigma^c$. It adds the item x at the p-th position in E_i. This operation updates E_i to E_{i+1} such that

  $E_{i+1} = E_i[1, p-1] \circ \; x \circ \; E_i[p, n]$, where $n = |E_i|$. $\circ$ denotes concatenation of sequence.

- $DELETE(p)$: $p \in \mathbb{N}$. It removes the item at the p-th position in E_i. This operation updates E_i to E_{i+1} such that

  $E_{i+1} = E_i[0, p-1] \circ \; E_i[p+1, n]$, where $n = |E_i|$.

3.2.1 Mapping from Physical View to Logical View

The mapping from the physical view to the logical view is defined by a $PIECE$ operation that returns the sequence of items whose position stamps are not mapped to ϕ. We call the items returned by PIECE *visible items* of the PPS. The concatenation of an item sequence seq with ϕ is still a item sequence such that $seq \circ \phi = seq$. The concatenation of two null items is still the null item such that $\phi \circ \phi = \phi$. PIECE is defined as below:

- $PIECE(S_k[s_i, s_j], M_k) = M_k(s_i) \circ M_k(s_{i+1}) \circ, ..., \circ$
  $M_k(s_j)$, where $s_i, s_{i+1} ..., s_j \in S_k$.

$PIECE(S_k[s_i, s_j], M_k)$ returns the sequence of visible items whose position stamps falls with the range of s_i and s_j (inclusive). $PIECE(S_k[s_1, s_n], M_k)$, where $n = |S_k|$, returns the sequence that contains all visible items. For brevity, we use $PIECE(S_k, M_k)$ to denote $PIECE(S_k[s_1, s_n], M_k)$ without writing the range explicitly. If $PIECE(S_k$ $[s_i, s_j], M_k) = \phi$, then it denote the empty sequence $\langle\rangle$ from a user's point of view.

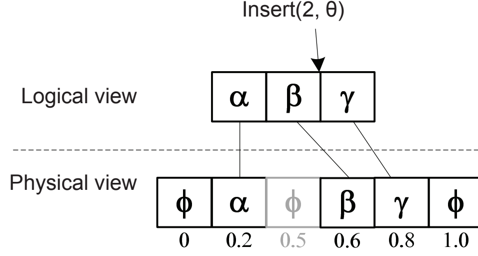


Figure 2: Mapping between two levels of view

3.2.2 Mapping from Logical View to Physical View

Next we show how to map the logical view to the physical view. As discussed in Section 3.2, users edit a document by issuing editing operations defined at its logical view, which need to be mapped into the forms on its physical view. The mapping rules are defined below:

- Rule 1: an $INSERT(p, x)$ on E_k is mapped to $ADD(s_{i-1}, s_i, x)$ on (S_k, M_k), where s_i is the smallest position stamp satisfying $p = |PIECE(S_k[s_1, s_i]\ , M_k)|$ if $E_k \neq \langle\rangle$ or $s_i = 1$ if $E_k = \langle\rangle$.

- Rule 2: a $DELETE(p)$ on E_k is mapped to $HIDE(s_i)$ on (S_k, M_k), where s_i is the smallest position stamp satisfying $p = |PIECE(S_k[s_1, s_i], M_k)|$.

Figure 2 gives an example for Rule 1. When a user issues an operation at offset p, Rule 1 locates the smallest position stamp s_i that satisfies $p = |PIECE(S_k[s_1, s_i], M_k)|$. In this example, it is the position between 0.6 and 0.8. This constraint establishes the correspondence between the item indexed by s_i in the physical view and its counterpart indexed at p on the logical view. An exception is when $E_k = \langle\rangle$. In this case, we require s_i to be the rightmost position stamp, which is 1.

Lemma 1. *Given the initial document version $E_0 = \langle\rangle$ and the initial physical document version $S_0 = \{0, 1\}$, $M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\}$, let $H = o_0 o_1 ... o_n$ be the logical-view*

editing history and $\tilde{H} = \tilde{o}_0\tilde{o}_1...\tilde{o}_n$ *be the physical-view editing history, obtained by applying Rule 1 and Rule 2. We have* $E_n = PIECE(S_n, M_n)$.

Proof by induction:

1. *Upon initialization,* $E_0 = \langle\rangle$, $PIECE(S_0, M_0) = M(0) \circ M(1) = \phi \circ \phi = \phi$.

2. *Assume the lemma holds for the k-th update* o_k, *such that* $E_k = PIECE(S_k, M_k)$

3. *For the* $k+1$*-th update,*

   - *suppose* $o_{k+1} = INSERT(p, x)$ *and its mapped form* $\tilde{o}_{k+1} = ADD(s_{i-1}, s_i, x)$ *by applying Rule 1. Based on the definition of INSERT in Section 3.2, we have* $E_{k+1} = E_k[0, p-1] \circ x \circ E_k[p, n]$, *where* $n = |E_k|$. *On the other hand, after applying* $ADD(s_{i-1}, s_i, x)$ *to the physical view* (S_k, M_k), *we have* $PIECE(S_{k+1}, M_{k+1}) = M_{k+1}(s_1) \circ ... \circ M_{k+1}(s_{i-1}) \circ M_{k+1}(s_{new}) \circ M_{k+1}(s_i) \circ ... \circ M_{k+1}(s_m)$, *where* $m = |S_k|$. *Based on the definition of ADD in Section 3.1, we know that* $\tilde{o}_{k+1}$ *does not change the mappings in* M_k. *Therefore, we get* $PIECE(S_{k+1}, \ M_{k+1}) = M_k(s_1) \circ ... \circ M_k(s_{i-1}) \circ M_k(s_{new}) \circ M_k(s_i) \circ ... \circ M_k(s_m)$. *Based on the assumption of* $E_k = PIECE\ (S_k, M_k)$ *and* $p = |PIECE(s_1, si)|$, *we know that* $E[1, p-1] = M_k(s_1) \circ ... \circ M_k(s_{i-1})$ *and* $E[p, n] = M_k(s_i) \circ ... \circ M_k(s_m)$. *Therefore,* $PIECE(S_{k+1}, \ M_{k+1}) = E[1, p-1] \circ x \circ E[p, n] = E_{k+1}$.

   - *suppose* $o_{k+1} = DELETE(p)$ *and its mapped form* $\tilde{o}_{k+1} = HIDE(s_i)$ *by applying Rule 2. Based on the definition of DELETE in Section 3.2, we have* $E_{k+1} = E_k[0, p-1] \circ E_k[p+1, n]$, *where* $n = |E_k|$. *On the other hand, after applying* $HIDE(s_i)$ *to the physical view* (S_k, M_k), *we have* $PIECE(S_{k+1}, M_{k+1}) = M_{k+1}(s_1) \circ ... \circ M_{k+1}(s_{i-1}) \circ M_{k+1}(s_i) \circ M_{k+1}(s_{i+1}) \circ ... \circ M_{k+1}(s_m)$, *where* $m = |S_k|$. *Based on the definition of HIDE in Section 3.1, we know that* $\tilde{o}_{k+1}$ *does not change the mapping of any position stamp in* M_k *except* s_i. *Therefore, we get* $PIECE(S_{k+1}, M_{k+1}) = \ M_k(s_1) \circ ... \circ M_k(s_{i-1}) \circ \phi \circ M_k(s_{i+1}) \circ ... \circ M_k(s_m) = M_k(s_1) \circ ... \circ M_k(s_{i-1}) \circ M_k(s_{i+1}) \circ ... \circ M_k(s_m)$. *Since* $E_k = PIECE(S_k, M_k)$

and $p = |PIECE(s_1, si)|$, *we know that* $E[1, p-1] = M_k(s_1) \circ ... \circ M_k(s_{i-1})$ *and* $E[p+1, n] = M_k(s_{i+1}) \circ ... \circ M_k(s_m)$. *Therefore,* $PIECE(S_{k+1},\ \ M_{k+1}) = E[1, p-1] \circ x \circ E[p+1, n] = E_{k+1}$.

The above lemma guarantees that whenever the document is updated by a history defined on its logical view, the mapping rules will correctly map the operations to their physical forms such that the version of the persistent sequence produced by the mapped history will maintain the same view as the one from users' perspective.

3.3 Representing the Revision History of Documents Based on PPSs

3.3.1 Different Design Options

In this section, we look at different design options to represent the revision history of a document based on PPSs. The revision history of a document is represents as a list of versions. Each version represents the state of the document at a particular time. In order to retrieve any of these versions, we use PPSs to track all the inserted and deleted items in a sequence and keep adequate timestamp information in order to tell whether an item belongs to a particular version.

Specifically, we maintain two timestamps for each position stamp: one for ADD timestamp and one for HIDE timestamp. For position stamps that are visible, the value of their HIDE timestamps is empty. These timestamps can be organized in any of the search tree data structures by using position stamps as keys. Figure 3 illustrate the organization. For simplicity, we draw the search tree as a sorted array with each element pointing to their timestamps. The first version of the PPS has two items with the timestamp being t_1 (We do not keep the timestamp for the position stamps 0 and 1 since they are not considered in any version.). In the second version of the PPS, α was deleted and its HIDE timestamp is updated. In the third version, γ was inserted with the timestamp being t_3. To construct a particular version of the

document, we traverse the array and examine the timestamp field of each position stamp to determine whether it belongs to a particular version. For example, to retrieve $version_2$ of the document, we find the item indexed by 0.2 does not belong to $version_2$ because its HIDE timestamp is equal to t_2. The item indexed by 0.5 belongs to $version_2$ because its ADD timestamp is smaller than t_2 and its HIDE timestamp is empty. The item indexed by 0.8 does not belong to $version_2$ because its ADD timestamp is larger than t_2. This organization, though simple, has a performance penalty for documents with a large number of items because all of them need to be examined.

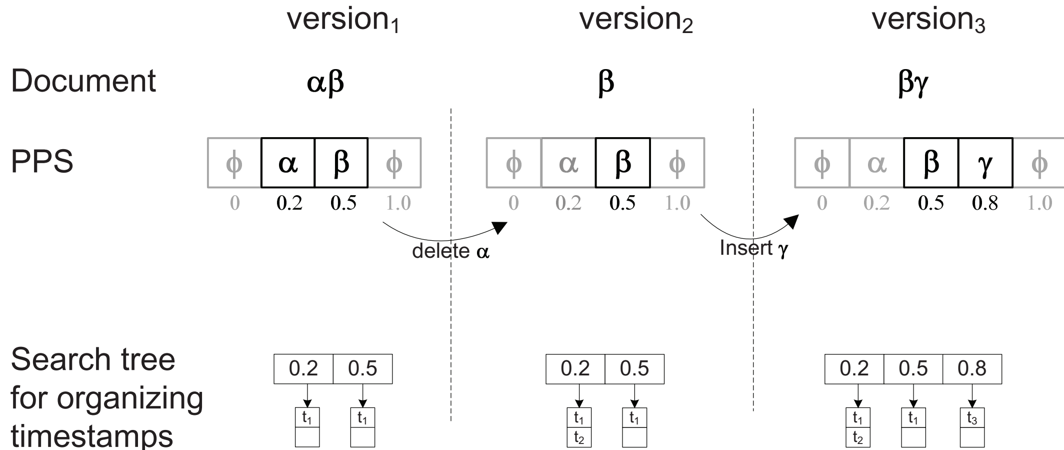


Figure 3: Organize the revision history of a document based on PPSs

Another issue of the above organization arises when we consider the REVERT operation. A REVERT operation restores the latest state of a document to one of the previous versions in its revision history. The introduction of REVERT is necessary because we cannot simply treat it as a DELETE followed by an INSERT. If that were the case, we would lose the correct insert timestamp of all restored items, which causes a problem when we come to the problem of metadata management discussed in Chapter 6. PPSs handle REVERT by comparing the position stamps between the latest version and the position stamps in the version that are reverted to. For items that are restored, their position stamps are marked visible again. For items that are

deleted, their position stamps are marked invisible. The REVERT operation adds new challenges to maintain timestamp information because we need to maintain a list of timestamps for each item. The visibility of an item could switch between the state of visible and invisible multiple times. The situation is illustrated in Figure 4. Now the item α has three timestamps. For documents that were reverted many times, we end up with a long list of timestamps for some of these items.

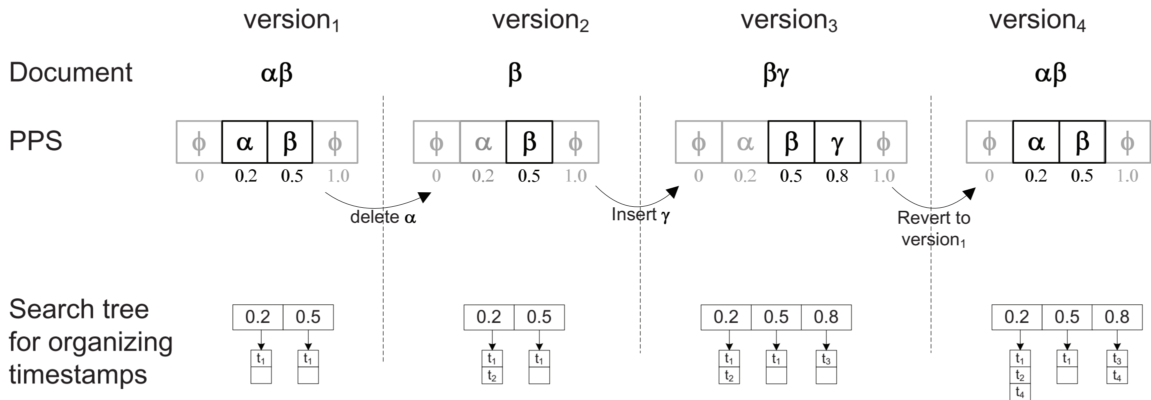


Figure 4: Organize the revision history of a document based on PPSs by considering the REVERT operation

An alternative way to organize the revision history of a document is to store all visible position stamps of the PPS for each version of the document as shown in Figure 5. In this way, we can retrieve a version directly without traversing the search tree. However, this approach uses up disk space quickly, especially in the situation that only minor changes happened between consecutive versions.

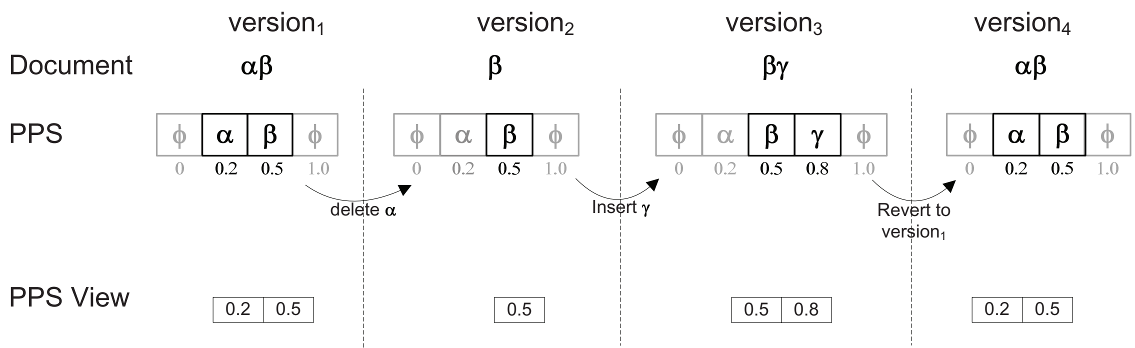


Figure 5: Store the full list of visible position stamps for every version of a document

3.3.2 A Hybrid Approach

We use a hybrid approach to represent the revision history of a PPS. The approach is hybrid because we maintain the full list of visible position stamps at multiple points. Between those points, only delta changes are maintained. The advantage of a hybrid approach is that it allows us to balance the tradeoff between disk space usage and access time. In the rest of this section, we first give an overview of our hybrid approach. Then, we introduce a delta-change applying operator and its properties. After that we describe several algorithms used to access the revision history of a document both at a particular time and for the delta changes between different versions.

The revision history of a PPS consists of as a list of PPS *views* $V_1, V_2, ..., V_n$. Each PPS view V_i consists of the full list of visible positions stamps in the PPS at t_i. We use $\Delta_{i,i+1} = (X_{i,i+1}, Y_{i,i+1})$ to denote the delta changes between the two PPS views V_i and V_{i+1}, where $X_{i,i+1}$ denotes all the position stamps that were marked visible and $Y_{i,i+1}$ that were marked invisible. For example, given three views of a PPS $V_1 = (0.1, 0.3)$, $V_2 = (0.1, 0.4)$, and $V_3 = (0.1, 0.3, 0.4)$, we have $\Delta_{1,2} = (X_{1,2}, Y_{1,2}) = (\{0.4\}, \{0.3\})$ and $\Delta_{2,3} = (X_{2,3}, Y_{2,3}) = (\{0.3\}, \emptyset)$

The hybrid form for the revision history of a PPS consists of a mixed sequence of PPS views and delta changes $V_1, \Delta_{1,2}, \Delta_{2,3}, ..., V_i, \Delta_{i,i+1}, \Delta_{i+1,i+2}, ...V_j$. We call the sub-list starting from a PPS view (inclusive) to the next PPS view (exclusive) a PPS *view range*, each of which covers parts of the revision history. In order to locate the PPS view range that a PPS view belongs to, we additional maintain an array structure, called *view pivot array*, containing the version identifiers for all PPS views in the hybrid form. The whole structure is illustrated in Figure 6.

To represent a PPS view range in a compact form as well as quickly locate the delta changes between any two PPS views, we use the technique of Compressed Sparse Row (CSR), a standard technique to save sparse matrices. The basic idea of CSR is to use three one-dimensional arrays [*val*, *col*, *row*] to save the content of a matrix.

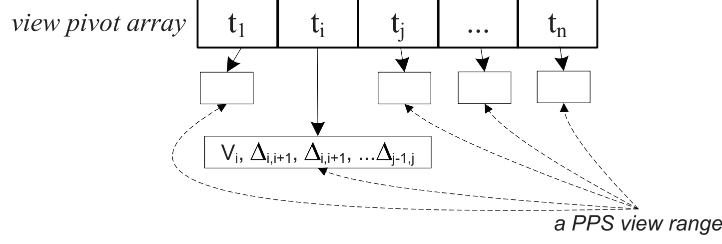


Figure 6: Represent the revision history of a PPS in a hybrid form

val saves the elements in the matrix from left to right and top to bottom. *col* saves the column indexes of the elements in the same order. *row* saves the last column index of each row. Let us look at an example to understand how we use CSR to store a PPS view range. Given the three views mentioned at the beginning of this section, we can represent their delta changes in a matrix-like structure as following:

$$\begin{array}{c} \\ \Delta(1,2) \\ \Delta(2,3) \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 0.1 & 0.3 & 0.4 & 1 \end{array} \\ \left(\begin{array}{ccccc} & & 0 & 1 & \\ & & 1 & & \end{array} \right) \end{array}$$

In general, each row contains the information for the delta changes between two consecutive views. For example, if the PPS view is V_1, the first row contains the delta changes between V_1 and V_2, the second row the delta changes between V_2 and V_3 and so on. The indexes of columns are rational numbers corresponding to position stamps. The values in the matrix-like structure are Booleans. If a position stamp s_u was marked invisible in V_{i+1}, the cell at (i, s_u) has value 0. Otherwise, the cell has value 1. So the CSR form for the above matrix is

$$val = [0, 1, 1]$$

$$col = [0.3, 0.4, 0.3]$$

$$row = [1, 2]$$

To retrieve the delta changes between V_{i+j} and V_{i+j+1} in a PPS view range with the first view being V_i, we follow three steps: 1) locate the starting point of the delta

changes with $row[j-1]$; 2) calculate the length of the delta changes by $|\Delta_{j,j+1}| = row[j] - row[j-1]$; 3) obtain the delta changes $val[row[j-1]+1, row[j-1]+2, ..., row[j]+|\Delta_{j,j+1}|$ and $col[row[j-1]+1, row[j-1]+2, ..., row[j]+|\Delta_{j,j+1}|]$

3.3.3 Properties of Delta Change Operator $\square$

Due to the uniqueness property of position stamps, we are able to calculate delta changes between two PPS views by using the classical set operators. Vice versa, we are also able to calculate a PPS view if we know its delta changes to another PPS view. Given the delta change $\Delta_{1,2}$ between V_1 and V_2, we use $X_{1,2}$ to denote position stamps that newly become visible in V_2 and $Y_{1,2}$ to denote position stamps that are visible in V_1, but newly become invisible in V_2. For example, given two PPS views $V_1 = (0.1, 0.3)$ and $V_2 = (0.1, 0.4)$. The delta changes between V_1 and V_2 are $X_{1,2} = V_2 \setminus V_1 = \{0.4\}$ and $Y_{1,2} = V_1 \setminus V_2 = \{0.3\}$. On the other hand, we have $V_2 = V_1 \cup X_{1,2} \setminus Y_{1,2} = (0.1, 0, 3) \setminus \{0.3\} \cup \{0.4\}$. Here we overload the set operators for sorted sequences. Their definitions are almost identical to set operators except that the elements in the set are sorted.

We use $\square$ to denote the functional composition of applying some delta changes to a view and define it below:

Definition 1. $V_1 = V_2 \square (X_{1,2}, Y_{1,2}) = (V_1 \cup X_{1,2}) \setminus Y_{1,2}$

$\square$ has several properties important to calculate PPS views and delta changes between two PPS views in a PPS view range. Next we first introduce these properties and prove their correctness and then explain the usage of these properties in the next section.

Lemma 2. ***(Properties of*** $\square$***)*** *Let V_1, V_2, and V_3 are three consecutive views,* $\square$ *has the following properties:*

1. $V_2 = V_1 \square (X_{1,2}, Y_{1,2}) \implies X_{1,2} \not\subset V_1 \wedge Y_{1,2} \subseteq V_1 \wedge X_{1,2} \cap Y_{1,2} = \emptyset$

2. $V_2 = V_1\Box(X_{1,2}, Y_{1,2}) \iff V_1 = V_2\Box(Y_{1,2}, X_{1,2})$

3. $V_2 = V_1\Box(X_{1,2}, Y_{1,2}) \wedge V_3 = V_2\Box(X_{2,3}, Y_{2,3}) \implies V_3 = V_1\Box(X_{1,3}, Y_{1,3})$, *where* $X = (X_{1,2} \setminus Y_{2,3}) \cup (X_{2,3} \setminus Y_{1,2})$ *and* $Y = (Y_{1,2} \setminus X_{2,3}) \cup (Y_{2,3} \setminus X_{1,2})$

Proof. 1. The first property can be directly inferred from the definition of $\Box$.

2. For the second property, we show the proof " $\implies$ " only. The proof of "$\impliedby$" is similar.

$$
\begin{aligned}
V_2\Box(Y_{1,2}, X_{1,2}) &= V_2 \cup Y_{1,2} \setminus X_{1,2} && (1)\\
&= (V_1\Box(X_{1,2}, Y_{1,2})) \cup Y_{1,2} \setminus X_{1,2} && (2)\\
&= (V_1 \cup X_{1,2} - Y_{1,2}) \cup Y_{1,2} \setminus X_{1,2} && (3)\\
&= (V_1 \cup X_{1,2} \cup Y_{1,2}) \setminus (Y_{1,2} \setminus Y_{1,2}) \setminus X_{1,2} && (4)\\
&= (V_1 \cup X_{1,2} \cup Y_{1,2}) \setminus X_{1,2} && (5)\\
&= (V_1 \cup Y_{1,2}) && (6)\\
&= V_1 && (7)
\end{aligned}
$$

Step(1) and (3) are based on the definition of $\Box$. Step(2) is based on the given precondition. Step(4) is based on the standard set complement preposition $(B \setminus A) \cap C = (B \cup C) \setminus (A \setminus C)$. Step(6) holds because $X_{1,2} \not\subset V_1$ and $X_{1,2} \cap Y_{1,3} = \emptyset$ based on the first property. Step(7) holds because $Y_{1,2} \in V_1$ based on the first property.

3. We prove $V_3 = V_1\Box(X_{1,3}, Y_{1,3})$ by proving two implications:

(a) $u \in V_3 \implies u \in V_1\Box(X_{1,3}, Y_{1,3})$, and

(b) $u \notin V_3 \implies u \notin V_1\Box(X_{1,3}, Y_{1,3})$

We prove (a) by showing that $u \in V_1 \cup X$ and $u \notin Y$. Based on the precondition and the first property, we know that if $u \in V_3$, then

$$
\begin{aligned}
u &\in (V_1 \Box (X_{1,2}, Y_{1,2})) \Box (X_{2,3}, Y_{2,3}) && (1)\\
&\in (((V_1 \cup X_{1,2}) \setminus Y_{1,2}) \cup X_{2,3}) \setminus Y_{2,3} && (2)\\
&\in (V_1 \cup X_{1,2} \cup X_{2,3}) \setminus (Y_{1,2} \setminus X_{2,3}) \setminus Y_{2,3} && (3)
\end{aligned}
$$

Since $u \in V_1 \cup X_{1,2} \cup X_{2,3}$ must hold, we know that $u \notin (Y_{1,2} \setminus X_{2,3})$ and $u \notin Y_{2,3}$. Therefore,

$$
\begin{aligned}
u &\notin (Y_{1,2} \setminus X_{2,3}) \cup Y_{2,3} && (4)\\
&\notin (Y_{1,2} \setminus X_{2,3}) \cup (Y_{2,3} \setminus X_{1,2}) = Y_{1,3} && (5)
\end{aligned}
$$

This finishes our proof for $u \notin Y$. We prove $u \in V_1 \cup X$ by contradiction. Assume $u \notin V_1 \cup X$, then

$$
\begin{aligned}
u \notin V_1 \cup (X_{1,2} \setminus Y_{2,3}) \cup (X_{2,3} \setminus Y_{1,2}) \implies && (6)\\
u \notin V_1 \wedge u \notin (X_{1,2} \setminus Y_{2,3}) \wedge u \notin (X_{2,3} \setminus Y_{1,2}) && (7)
\end{aligned}
$$

Since we know that $u \in V_1 \cup X_{1,2} \cup X_{2,3}$, we know that $u \in X_{1,2} \cup X_{2,3}$ must hold. We consider two cases:

- If $u \in X_{1,2}$, then $u \in Y_{2,3}$ must hold based on the assumption $u \notin (X_{1,2} \setminus Y_{2,3})$. This contradicts with the conclusion (4) drawn from the precondition. If $u \in X_{2,3}$, then $u \in Y_{2,3}$ must hold. However, we know that $Y_{2,3} \in V_1$ based on the first property. This leads to the conclusion that $u \in V_1$, which contradicts with our assumption. Therefore, $u \in V_1 \cup X$ holds.

- If $u \in X_{2,3}$, based on $u \notin (X_{2,3} \setminus Y_{1,2})$, we have $u \in Y_{1,2}$. Based on the first property, we know $Y_{1,2} \in V_1$, which contradicts with our assumption. Therefore, $u \in V_3$.

Now we prove (b). Based on the precondition, we know that if $u \notin V_3$, then $u \notin (V_1 \cup X_{1,2} \cup X_{2,3}) \setminus (Y_{1,2} \setminus X_{2,3}) \setminus Y_{2,3}$. Since we know $u \in V_1 \cup X_{1,2} \cup X_{2,3}$, then we get $u \in (Y_{1,2} \setminus X_{2,3}) \cup Y_{2,3}$. We consider two situations:

- If $u \notin X_{1,2}$, then we get $u \in ((Y_{1,2} \setminus X_{2,3}) \cup (Y_{2,3} \setminus X_{2,3})) = Y$. Therefore, $u \notin (V_1 \cup X_{1,3}) \setminus Y$.
- If $u \in X_{1,2}$, then we have $u \notin Y_{2,3}$ based on the first property. Therefore, $u \in Y_{2,3}$ must hold. Therefore $u \notin X_{1,2} \setminus Y_{2,3}$. Furthermore, based on the first property, we know that if $u \in X_{1,2}$, then $u \notin V_1$. Also if $u \in Y_{2,3}$, then $u \notin X_{2,3}$. Therefore, we have $u \notin V_1 \wedge (X_{1,2} \setminus Y_{2,3}) \wedge (X_{2,3} \setminus Y_{1,2}) = X$, which leads to the conclusion that $u \notin (V_1 \cup X_{1,3}) \setminus Y$.

This finishes our proof for both (a) and (b). Finally, we need to show that X and Y satisfy the first property. That is $X \not\subset V_1$, $Y \subseteq V_1$, and $X \cap Y_{1,3} = \emptyset$.

- Based on the preconditions the first property, we know

$$X_{2,3} \not\subset V_2 = (V_1 \cup X_{1,2}) \setminus Y_{1,2} \implies$$
$$X_{2,3} \not\subset (V_1 \setminus Y_{1,2}) \cup X_{1,2} \implies$$
$$X_{2,3} \not\subset (V_1 \setminus Y_{1,2})$$

  Since $X_{2,3} \setminus Y_{1,2} \not\subset Y_{1,2}$ holds trivially, we have

$$X_{2,3} \cap (X_{2,3} \setminus Y_{1,2}) \not\subset (V_1 \setminus Y_{1,2}) \cup Y_{1,2} \implies (X_{2,3} \setminus Y_{1,2}) \not\subset V_1$$

  Furthermore, since $X_{1,2} \not\subset V_1$, we have $X_{1,2} \setminus Y_{2,3} \not\subset V_1$. Therefore

$$X_{1,3} = (X_{2,3} \setminus Y_{1,2}) \cup (X_{1,2} \setminus Y_{2,3}) \not\subset V_1$$

- Based on the first property, we have

$$Y_{2,3} \subseteq V_2 = (V_1 \cup X_{2,3}) \setminus Y_{2,3} \implies$$
$$Y_{2,3} \subseteq (V_1 \cup X_{2,3}) \implies$$
$$Y_{2,3} \setminus X_{2,3} \subseteq V_1$$

Since $Y_{1,2} \subseteq V_1$ holds based on the first property, $Y_{1,2} \setminus X_{2,3} \subseteq V_1$ holds trivially. Therefore,

$$Y_{1,3} = (Y_{2,3} \setminus X_{2,3}) \cup (Y_{1,2} \setminus X_{2,3}) \subseteq V_1$$

- We have

$$\begin{aligned} X \cap Y_{1,3} &= ((X_{1,2} \setminus Y_{2,3}) \cup (X_{2,3} \setminus Y_{1,2})) \cap (Y_{1,2} \setminus X_{2,3}) \cup (Y_{2,3} \setminus X_{1,2}) \\ &= \underbrace{(X_{1,2} \setminus Y_{2,3}) \cap (Y_{1,2} \setminus X_{2,3})}_{(1)} \cup \underbrace{(X_{2,3} \setminus Y_{1,2}) \cap (Y_{1,2} \setminus X_{2,3})}_{(2)} \\ &\cup \underbrace{(Y_{1,2} \setminus X_{2,3}) \cap (Y_{2,3} \setminus X_{1,2})}_{(3)} \cup \underbrace{(X_{2,3} \setminus Y_{1,2}) \cap (Y_{2,3} \setminus X_{1,2})}_{(4)} \end{aligned}$$

It is easy to show that each of the items (1)-(4) is empty based on the properties of $X_{1,2} \cap Y_{1,2} = \emptyset$ and $X_{2,3} \cap Y_{2,3} = \emptyset$.

□

3.3.4 Operations on a PPS View Range

We describe how to compute either a view or delta changes between two views in a PPS view range based on the three properties of $\Box$ introduced in the previous section.

3.3.4.1 Applying Delta Changes to a View

Given a view V_1 and its delta changes $\Delta_{1,2}$ to V_2, we explain how to compute the view V_2. Based on Property 1 in Lemma 2, we know that a position stamp $s \in \Delta_{1,2}$ will be added into V_2 if s is not in V_1 and a position stamp $s \in \Delta_{1,2}$ will be removed from V_1 if it is in V_1. Based on this observation, we can use the algorithm APPLY-DELTA-TO-VIEW to apply delta changes to a view in $O(n + m)$ where n is the number of position stamps in V_1 and m the number of position stamps in $\Delta_{1,2}$. The algorithm APPLY-DELTA-TO-VIEW takes two parameters. V is a sorted double

Algorithm 1 APPLY-DELTA-TO-VIEW(V, S)

```
1: i ← 0
2: j ← 0
3: k ← 0
4: A ← an empty array
5: while i < length[V] and j < length[S] do
6:    if S[i] < S[j] then
7:       k ← k + 1
8:       A[k] ← V[i]
9:       i ← i + 1
10:   else if S[i] = S[j] then
11:      i ← i + 1
12:      j ← j + 1
13:   else
14:      k ← k + 1
15:      A[k] ← S[i]
16:      j ← j + 1
17:   end if
18: end while
19: while i < length[V] do
20:   k ← k + 1
21:   A[k] ← V[i]
22:   i ← i + 1
23: end while
24: while j < length[S] do
25:   k ← k + 1
26:   A[k] ← S[i]
27:   j ← j + 1
28: end while
29: return A
```

array containing position stamps in V_1. S is a sorted double array containing position stamps in $\Delta_{1,2}$. This algorithm merges the two sorted array into a single sorted array, which is the same as the merge step of a merge-sort algorithm [45] except that it drops elements that exist in both arrays. The code maintains two indexes pointing to the next elements to compare in each array. The main loop in line 5-18 compares the elements in the two arrays from left to right. For elements that exist in both arrays (line 10-12), they are not copied into the output array A. For the elements that only exist in V, they are copied into the output array A (line 7-8) and the index i is updated to pointing to the next element in V. The elements that only exist in S is processed similarly (line 14-16). At the end of the execution, APPLY-DELTACHANGE-TO-VIEW returns a sorted double array A containing the position stamps in V_2.

3.3.4.2 Calculating Transitive Delta Changes

Algorithm 2 CALCULATE-TRANSITIVE-DELTA(S_1, F_1, S_2, F_2)

1: $i \leftarrow 0$
2: $j \leftarrow 0$
3: $k \leftarrow 0$
4: $S \leftarrow$ an empty double array
5: $F \leftarrow$ an empty boolean array
6: **while** $i < length[S_1]$ and $j < length[S_2]$ **do**
7:     **if** $S_1[i] < S_2[j]$ **then**
8:         $k \leftarrow k + 1$
9:         $S[k] \leftarrow S_1[i]$
10:         $F[k] \leftarrow F_1[i]$
11:         $i \leftarrow i + 1$
12:     **else if** $S_1[i] = S_2[j]$ **then**
13:         $i \leftarrow i + 1$
14:         $j \leftarrow j + 1$
15:     **else**
16:         $k \leftarrow k + 1$
17:         $S_1[k] \leftarrow S_2[i]$
18:         $F[k] \leftarrow F_2[i]$
19:         $j \leftarrow j + 1$
20:     **end if**
21: **end while**
22: **while** $i < length[S_1]$ **do**
23:     $k \leftarrow k + 1$
24:     $S[k] \leftarrow S_1[i]$
25:     $F[k] \leftarrow F_1[i]$
26:     $i \leftarrow i + 1$
27: **end while**
28: **while** $j < length[S_2]$ **do**
29:     $k \leftarrow k + 1$
30:     $S[k] \leftarrow S_2[i]$
31:     $F[k] \leftarrow F_2[i]$
32:     $j \leftarrow j + 1$
33: **end while**
34: **return** (S, F)

Given the delta changes $\Delta_{1,2}$ between V_1 and V_2 and the delta changes $\Delta_{2,3}$ between V_2 and V_3, the algorithm CALCULATE-TRANSITIVE-DELTA calculate the delta changes between V_1 and V_3 in $O(n + m)$ where n is the number of position stamps in $\Delta_{1,2}$ and m the number of position stamps in $\Delta_{2,3}$. Based on the Property 3 in Lemma 2, we know that a position stamp $s \in \Delta_{1,2} \cup \Delta_{2,3}$ will not belong to either $\Delta_{1,3}$ if it exists in both arrays. For position stamps that exist in only one of the arrays will have the same boolean values as in their original delta change arrays. The CALCULATE-TRANSITIVE-DELTA algorithm takes four parameters. S_1 and S_2 are sorted arrays containing position stamps in $\Delta_{1,2}$ and $\Delta_{2,3}$ respectively. F_1 and F_2

are two boolean arrays. F_1 stores the visibility information for the position stamps in $\Delta_{1,2}$ while F_2 stores visibility information for the position stamps in $\Delta_{2,3}$ Similar to APPLY-DELTA-TO-VIEW, the code in CALCULATE-TRANSITIVE-DELTA merges the two sorted arrays into a single sorted array and drops all the elements that exist in both arrays. In addition, the algorithm copies the values in both boolean arrays into the new boolean array F.

3.3.4.3 *Calculating Delta Changes between Two PPS Views Within A PPS View Range*

Algorithm 3 CALCULATE-DELTA-WITHIN-RANGE(val, col, row, u, v)

1: $i \leftarrow row[u-1]+1$
2: $j \leftarrow row[u] - row[u-1]$
3: $S_1 \leftarrow col[i, ...j]$
4: $F_1 \leftarrow val[i, ...j]$
5: **for** $k = u+1 \rightarrow v$ **do**
6: $\quad i \leftarrow row[k-1]+1$
7: $\quad j \leftarrow row[k] - row[k-1]$
8: $\quad S_2 \leftarrow col[i, ...j]$
9: $\quad F_2 \leftarrow val[i, ...j]$
10: $\quad (S_1, F_1) \leftarrow$ CALCULATE-TRANSITIVE-DELTA(S_1, F_1, S_2, F_2)
11: **end for**
12: **return** (S_1, F_1)

We describe how to compute delta changes between any two views within a PPS view range. Based on Property 3 in Lemma 2, we can calculate the delta changes between any two views by the composition of $\Box$ for all the delta changes between two PPS views. The algorithm is described in CALCULATE-DELTA-WITHIN-RANGE. Let $n = j - i + 1$, the complexity of this algorithm is lower bound by

$$\begin{aligned}
&cost(\text{CALCULATE-DELTA-WITHIN-RANGE}) \\
&= (n-1)|\Delta_{i,i+1}| + (n-2)|\Delta_{i+1,i+2}| + ...|\Delta_{j-1,j}| \\
&\geq \frac{(n-1)*n}{2} min(|\Delta_{i,i+1}|, \Delta_{i+1,i+2}|, ..., \Delta_{j-1,j}|) \\
&= O(n^2 \Delta_{min})
\end{aligned}$$

Therefore, the complexity of CALCULATE-DELTA-WITHIN-RANGE will be proportional to n^2.

3.3.4.4 Calculating a PPS view

Given a PPS view range, we have two options to calculate a PPS view. We can start with the full PPS view in the PPS view range and repeatedly apply the delta changes by calling APPLY-DELTA-TO-VIEW to produce the requested PPS view. The complexity of this approach is $(n-1)|V| + (n-2)|\Delta_{i,i+1}| + ... + |\Delta_{j-1,j}|$. An alternative is to calculate the composite delta changes between the full PPS view and the request PPS view by calling CALCULATE-DELTA-WITHIN-RANGE and then call APPLY-DELTA-TO-VIEW to obtain the requested view. The complexity of this approach is $|V| + (n-1)|\Delta_{i,i+1}| + ... + |\Delta_{j-1,j}|$. Since the size of V is normally much larger than a delta change, the second option is much better. The algorithm CALCULATE-VIEW takes five parameters. The V is the full PPS view in a PPS view range. *val*, *col*, and *row* are the delta changes. u is the version identifier for the requested PPS view.

Algorithm 4 CALCULATE-VIEW(V, val, col, row, u)

1: $(S, F) \leftarrow$ CALCULATE-DELTA-WITHIN-RANGE(*val*, *col*, *row*, 1, u)
2: **return** APPLY-DELTA-TO-VIEW(V, S)

3.3.4.5 Calculating Delta Changes between Any Two PPS Views

To calculate the delta changes between any two PPS views, we first check whether the two views are within the same PPS view range based on the pivot array. The BINARY-SEARCH function takes an integer array and a search key and returns the index of the key that is largest among all the keys smaller than the search key. If the answer is yes (line 3), we can call CALCULATE-DELTA-WITHIN-RANGE to get the result. Otherwise, we use CALCULATE-VIEW to obtain the two PPS views first and then calculate their delta changes by CALCULATE-TRANSITIVE-DELTA.

Algorithm 5 CALCULATE-DELTA(R, p, u, v)

```
1: i ← BINARY-SEARCH(R.pivot, u)
2: j ← BINARY-SEARCH(R.pivot, v)
3: if i=j then
4:     return
5:     CALCULATE-DELTA-WITHIN-RANGE(R.range[i].val, R.range[i].col, R.range[i].row, u, v)
6: else
7:     V1 ← CALCULATE-VIEW(R.range[i].V, u)
8:     V2 ← CALCULATE-VIEW(R.range[j].V, v)
9:     F1 ← allocate a boolean array with size length[V1]
10:    F2 ← allocate a boolean array with size length[V2]
11:    for k = 1 → length(V1) do
12:        F1[k] ← FALSE
13:    end for
14:    for k = 1 → length(V2) do
15:        F2[k] ← TRUE
16:    end for
17:    return CALCULATE-TRANSITIVE-DELTA(V1, F1, V2, F2)
18: end if
```

Given two PPS views V_1 and V_2, based on the definition of $\square$, we can write them as:

$$V_1 = V_0 \square (V_1, \emptyset)$$

$$V_2 = V_0 \square (V_2, \emptyset)$$

where V_0 is an empty PPS view. Based on Property 2 in Lemma 2, we have $V_0 = V_1 \square (\emptyset, V_1)$. Then we can calculate the delta changes between V_1 and V_2 based on Property 3 in Lemma 2. The algorithm CALCULATE-DELTA takes three parameters. R is a PPS view range. We use *R.pivot* to denote its pivot array and *R.range* to denote its PPS view range array. u and v are version identifiers for the two PPS views.

3.4 Summary

In this chapter, we introduce partial persistent sequences, the partially persistent form of the sequence data structure. Conceptually, a PPS consists of a list of items indexed by rational numbers, called position stamps. PPSs have two important features. First, items are never removed from the data structure. Deleted items are only marked invisible to editing applications. By keeping necessary timestamp information for the

items, we are able to access the revision history of a document at any time. Second, PPSs create unique, persistent, and ordered identifiers for all the items. These three properties make it easy to track changes of items as documents get modified. In order to balance the tradeoff between disk space usage and access time for previous versions, we design a hybrid approach to represent and access the revision history of documents based on PPSs. In the hybrid approach, each version of a document is represented by the position stamps of the containing items. We only maintain the full list of position stamps for a few versions at different points of the revision history. For versions between these points, only their delta changes are maintained. In addition, we define a delta-change operator to operate over position stamps in different versions. This delta-change operator has several important properties, which makes it possible to access the revision history of documents by using the classical set operators. Based on the delta-change operator, we design efficient algorithms to reconstruct a version at any time as well as computing the delta changes between any two versions.

CHAPTER IV

DATA CONSISTENCY CONTROL IN REAL-TIME COLLABORATIVE TEXT DOCUMENT EDITING SYSTEMS

4.1 Motivation

With the technological advance in collaborative editing tools, more and more documents are co-authored. We have observed a proliferation of collaboration tools including Gobby [2], Google Docs [15], SubEthaEdit [8], and Coword [133]. To provide quick response time for the edits of local users and timely updates for the edits of remote users, most collaborative tools adopt the architecture that a shared document is replicated at multiple sites connected by communication networks. The user at each site can update his/her local replica by insertions and deletions anytime and anywhere. Local edits are executed immediately for quick response time. The underlying editing system is responsible for view synchronization among all replicas by propagating local updates (i.e. delta changes) to other sites either through a dedicated central server [15] or in a peer-to-peer fashion [133]. One site acts as a server that manages user membership in the editing session and keeps the document durable for future access. To recover from damaging edits, the server site also maintains the editing history as a sequence of revisions. A collaborative editing example is shown in Figure 7. Three users work on a shared document with initial content "*abcd*". The vertical lines represent the elapse of time. Circles represent locally generated operations, which are executed immediately. Arrows represent the propagation of local operations to other sites. In this scenario, $user_1$ inserts 'e' at offset 0. Simultaneously $user_2$ inserts 'f' at offset 2. After executing o_1 and o_2's updates, $user_3$'s document

replica is modified to "$eabfcd$". Then o_3 deletes 'c' at offset 4. In this example, the three operations are executed in different orders at the three sites. Communication between users must be carefully coordinated due to concurrent editing conflicts.

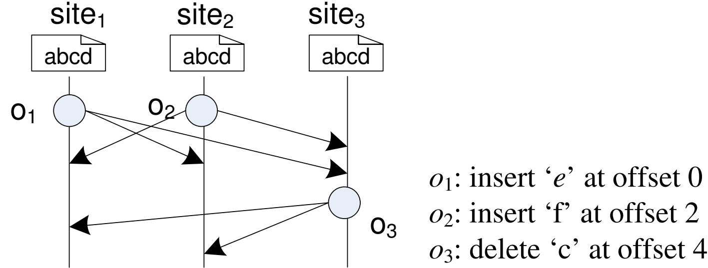


Figure 7: A real-time collaborative editing scenario

The requirement of maintaining data consistency in collaborative editing scenarios has been well-studied in the literature. Several consistency models have been proposed [57, 87, 98, 113]. Essentially, they require the following two properties: 1) all document replicas eventually converge to the same state; 2) execution of editing operations conforms to the happen-before precedent order defined by Lamport's Clock Condition [82]. These earlier consistency models can be unnecessarily restricted in editing scenarios in which users want to edit different parts of the document without interference and synchronize only when they edits the same area. For example, two accountants work together on a financial report for different departments. There is no overlap between their edits when they work on different departments. The second issue we address in this chapter is how to relax the current consistency models and make it suitable for the targeted scenarios. Our contributions in this chapter are summarized below:

- Proposal of a relaxed data consistency model. Our data consistency model has two properties: *eventual consistency* and *data-dependency precedent order preservation*. Concretely, a data dependency is defined on a pair of operations if they modify overlapped or contiguous characters. Edits happening on different parts of a document are not data-dependency related and can be executed

concurrently without synchronization. The second property is more relaxed than the property of happen-before precedent order preservation. Therefore, we expect that it can be enforced more efficiently. We will explain how we use the PPS data structure to track data-dependencies between editing operations to guarantee the relaxed data consistency model.

- An experimental evaluation on performance of PPSs in terms of both disk space consumption and access time for document update and retrieval. We choose a Wikipedia data dump [5] as our experimental data set to generate real world collaborative editing traces. Experimental results show that PPSs can be updated within several seconds for thousands of edits and kilobytes of characters. We also compare the disk space consumption of PPSs with two kinds of content management systems to store documents and their corresponding revisions. The first system stores revisions of documents as individual files without compressing overlapped content between consecutive revisions. The second system is RCS [117], which only stores delta differences between consecutive revisions. Our experimental result shows that our PPS-based collaborative system achieves a compression ratio much closer to RCS as compared to the first system.

Roadmap. In the rest of the chapter, we first define a data consistency model based on the PPS data structure and describe its corresponding synchronization strategy in Section 4.2. We then give technical details for a prototype implementation of a collaborative editor in Section 4.3. Experimental results are presented in Section 4.4.

4.2 DDP Consistency Model and View Synchronization Strategy

4.2.1 DDP Consistency Model

A data consistency model defines correctness criteria in collaborative editing systems (CESs). We propose a data-dependency preservation (DDP) consistency model consisting of two properties:

- *convergence property*: it states that the replicas of a shared document converge to the same logical view after executing all updates if no new update arrives, and if all nodes are connected. The resulting view contains all generated updates.

- *data-dependency precedence preservation property*: it states that if one operation o_j data-depends on o_i, then o_i should be executed before o_j at all the sites. We say o_j data-depends on o_i, denoted as $o_i \rightarrow o_j$, if

  - o_j deletes the character inserted by o_i.

  - o_j inserts a character contiguous to the character inserted by o_i.

The convergence property requires that all document replicas converge to the same value. This is different from *semantic consistency*, which demands that the converged value is also meaningful in the application context [53]. Semantic consistency requires domain specific knowledge, which is hard to verify by relying on pure system approaches. An example is two users trying to fix a grammar error in the sentence "*There should be student*" at the same time [113]. One inserts '*a*' after "*be*" while the other inserts a '*s*' after "*student*". After the modification, the sentence becomes "*There should be a students*", which is not semantically correct. Current CESs enforces converged views and leave semantic consistency to the interpretation of end users.

The data-dependency precedence preservation property guarantees that editing operations are executed within their surrounding context. For example, if a user

writes "*a day*", then inserts a "*nice*" between these two words, he expects the execution of the insertion for the phrase "*a day*" is executed first, the insertion for the word "*nice*" second. Here the phrase "*a day*" is the context for the word "*nice*". Another example is if the user inserts a word first, then deletes it, he also expects the execution of the insertion first and the deletion second at all the sites. Editing operations not data-dependency related are allowed to be executed in any order. Therefore, users working on different portions of the document can collaborate efficiently without any interference. In the next section, we will precisely define the data-dependency in terms of position stamps.

4.2.2 View Synchronization Strategy

A view synchronization strategy resolves editing conflicts when users simultaneously edit the overlapped or contiguous characters. A CES is defined by a triple $CES =< U, D, \tilde{O} >$, where

- U: a set of unique site identifiers. $U = \{u_i, 1 \leq i \leq n, s_i \in \mathbb{N}, n \in \mathbb{N}\}$
- D: a set of PPSs. $D = \{ps_i, 1 \leq i \leq n, n = |U|\}$. ps_i is the PPS at u_i.
- $\tilde{V}$: a set of parameterized editing operations. $\tilde{V} = \{\tilde{v}_i, 1 \leq i \leq \mathbb{N}\}$, where $\tilde{v}_i$ is one of the kinds
  - $(ADD(s_i, s_{i+1}, x), u_k)$: an ADD generated by u_k.
  - $(HIDE(s_i), u_k)$: a HIDE generated by u_k.

An execution of a CES at a particular site is modeled by an editing history $H = \tilde{v}_1\tilde{v}_2...\tilde{v}_n$. We use $op(H)$ to denote the set of operations in H and $<_H$ to denote their ordering. Starting with the initial version (S_0, M_0), we use (S_H, M_H) to denote the version produced by history H.

Next we show that if every site executes the same set of operations in an order that preserves their data-dependencies, the PPS at each site will converge to the same

value. We first define data-depends relationships, and then define data-dependency preserving histories.

Definition 2. *Data Depends Relation* $\rightarrow$. *Given two operations* $\tilde{v}_p$ *and* $\tilde{v}_q$*, we say* $\tilde{v}_q$ *depends on* $\tilde{v}_p$*, denoted as* $\tilde{v}_p \rightarrow \tilde{v}_q$*, if one of the following conditions is satisfied:*

1. $\tilde{v}_p = (ADD(s_i, s_{i+1}, x), u_m)$ and $\tilde{v}_q = (HIDE(s_j),\ u_n)$. Let s_{new} be the position stamp generated for x by $\tilde{v}_p$. We have $s_{new} = s_j$. In other words, $\tilde{v}_q$ maps the character inserted by $\tilde{v}_p$ to ϕ.

2. $\tilde{v}_p = (ADD(s_i, s_{i+1}, x), u_m)$ and $\tilde{v}_q = (ADD(s_j, s_{j+1},\ y), u_n)$. Let s_{new} be the position stamp generated for x by $\tilde{v}_p$. We have either $s_{new} = s_j$ or $s_{new} = s_{j+1}$. In other words, $\tilde{v}_q$ inserts a character next to the one inserted by $\tilde{v}_p$.

3. $\exists \tilde{v}_x, \tilde{v}_p \rightarrow \tilde{v}_x$ and $\tilde{v}_x \rightarrow \tilde{v}_q$.

Definition 3. *Data-dependency-preserving History. A history* $H = \tilde{v}_1\tilde{v}_2...\tilde{v}_n$ *is said to be data-dependency-preserving if* $\tilde{v}_i \rightarrow \tilde{v}_j$ *then* $\tilde{v}_i <_H \tilde{v}_j$*.*

Definition 4. *Data-dependency Equivalence. Let* H *and* H' *be two histories.* H *and* H' *are called data-dependency equivalent, denoted as* $H \approx_d H'$*, if the following holds:*

1. $op(H) = op(H')$*, and*

2. H *and* H' *are data-dependency preserving.*

Theorem 1. *If* H *and* H' *are data-dependency equivalent, then starting with the same initial empty PPS, we have* $S_H = S_{H'}$ *and* $M_H = M_{H'}$*.*

Theorem 1 guarantees that if each site executes all updates (both local and remote) in their data-dependency precedent order, the final versions of PPSs at each site will converge to the same value. Since data dependencies between edits can be precisely captured by position stamps encoded in editing operations, a view synchronization strategy can easily check them and maintain their precedent orders when executing edits at each user site.

4.3 System Implementation Based on PPSs

4.3.1 System Architecture Overview

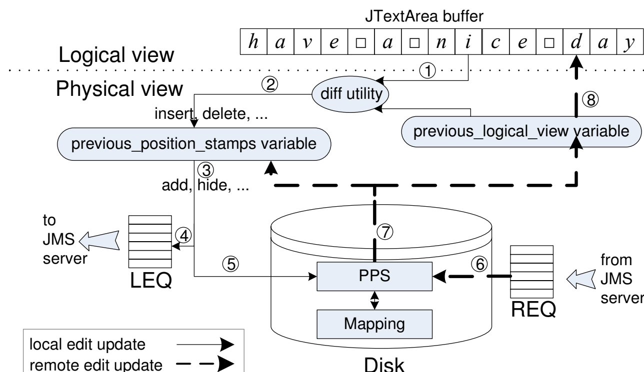


Figure 8: System Architecture for the PPS-based real-time collaborative document editing system

We have prototyped a collaborative editor based on the PPS data structure. Figure 8 illustrates the system architecture for a local editor. It communicates with other sites through a Java Message Service (JMS) server [1]. JMS is a messaging standard that allows application to send and receive messages in a loosely coupled environment. We use the publish/subscribe feature of the JMS sever in which each document is published as a topic. Users involved in an editing session all subscribe to this topic. Each site publishes their local edits to the JMS server, which will relay them to all other sites.

The logical view of a document is implemented by the javax.swing package [4]. We use an instance of JTextArea class to handle text display and screen scrolling. When edits (i.e. insert and delete) occur, the JTextArea instance updates the document content in its own buffer and refreshes the logical view instantly. This guarantees quick response time for the local user. There are two queues: *local edit queue* (LEQ) and *remote edit queue* (REQ). LEQ stores local edits to be sent to the JMS server. REQ stores remote edits sent from the JMS server.

A background thread, called edit processing thread (EPT), is responsible for both processing local edits and refreshing the logical view with remote edits. EPT runs at

a configurable interval or user specified moments, e.g. clicking a save button. Note that EPT never blocks a local user from editing the logical view. The frequency of its execution will only impact how fast local edits will be propagated to other user sites. EPT maintains two variables in memory: *previous_logical_view* (*prev_lv*) variable and *previous_position_stamps* (*prev_ps*) variable. The first variable stores the logical view seen in its previous check. The second variable stores the position stamps of characters in the logical view. During its execution, EPT first checks the occurrence of new local edits since its last check. If new local edits are detected, it updates the underlying PPS following the steps of (1)-(5) in Figure 8. These steps are to: 1) get the current logical view from the JTextArea instance buffer; 2) compute the changes between the previous logical view and the current logical view; 3) compute update operations on the PPS based on the position stamps of the previous logical view; 4) put update operations into LEQ; 5) update the PPS on disk. If no local edits are detected, it proceeds to check REQ for the existence of any remote edits. If REQ is not empty, it processes remote edits following the steps of (6)-(8) in Figure 8. These steps are to: 1) de-queue remote edits from REQ and apply them to the PPS; 2) compute the visible character sequence and their corresponding position stamps to update *prev_lv* and *prev_ps* respectively. The new character sequence is the result of the PIECE function defined in Section 3.2; 3) reset the logical view to the new character sequence containing the recent remote updates. When EPT executes remote edits, it also needs to enforce their execution order consistent to their data dependencies. This can be easily done by a lookup in the PPS. If the position stamp an edit depends on already exists, the edit will be executed as usual. Otherwise, it will be temporarily put in a waiting queue, which will be checked again after executing new edits.

Special attention needs to be paid for processing remote edits. Even though the procedure for remote updates runs very fast (it takes less than a second for a hundred edits based on our experiment results in Section 4.4), there is a small chance that

the user issues new local edits during the time EPT processes remote edits. In this situation, EPT would update the PPS in a way independent of the logical view, which creates inconsistency between the logical view and the physical view. EPT handles this situation in a simple way. Before it sets the logical view with the new character sequence, it first checks the existence of new local edits. If no local edits are detected, it resets the logical view as described in the above steps. If new local edits exist, it gets the current logical view from the JTextArea buffer, merges it with the new character sequence, and finally updates the logical view. The merge can be correctly handled based on the values of *prev_lv* and *prev_ps*. We omit the detail here due to its simplicity.

4.3.2 PPS Update

It is necessary that EPT finishes both local and remote updates within a short interval. In some real-time collaborative editors, the interval can be as short as a few seconds. Furthermore, it is important that the PPS data structure is stored economically on disk, especially for the server site that maintains revisions of documents.

To meet the above requirements, the PPS data structure needs to support efficient insertions and deletions. Instead, they An insertion adds new position stamps between two existing ones. It looks up the PPS to locate the position where after insertion the sorted order of the data structure is still maintained. A deletion hides some existing position stamps. It starts with looking up the PPS to locate these position stamps and then mark them invisible. The data structure also needs to support efficient sequential retrieval for refreshing the logical view with remote updates.

Our first optimization technique is to represent PPSs in search tree data structures with keys being position stamps and values being the characters. In the concrete implementation, we use Berkeley DB [95] to store PPSs on disk with the access method B+tree. Berkeley DB is an embeddable database providing high performance

for managing key/value data structures. The PPS data structure inherits the cost of B+tree, which is $O(tlog_t n)$ for lookup, insertion and deletion of records, where t is the upper bound on the number of keys in a B-tree node and n the number of position stamps. We choose the default page size for a B-tree node, which is 4096 bytes. Each position stamp takes 4 bytes. If we assume that a node holds m 4-byte search-key values and $(m+1)$ 4-byte pointers, it can hold $m \leq 511$ key-pointer pair at maximum according to $4*m+4*(m+1) \leq 4096$. If we further assume that each node has occupancy halfway between the minimum and the maximum key-pointer pair, it holds 384 key-pointer pairs on average. For a two-level B+-tree, it can hold up to $384^2 = 147,456$ position stamps. For a three-level B+tree, it can hold up to $384^3 = 56,623,104$ position stamps, which is adequate for representing megabyte documents. The level of B+-tree will be $2 \sim 3$ for most documents. Therefore, updates to the PPS data structure can be implemented efficiently since the number of disk I/O operations will be restricted to a small number.

Storing PPSs in B+trees itself is inadequate to achieve our goals. Storing all position stamps individually would incur lots of small disk I/Os because a sequential retrieval of all visible characters needs to access many small leaf nodes in the B+tree. Furthermore, in our implementation, a position stamp is a 4-byte integer. We use all 32 bits to represent the significant precision of a rational number with an implicit integral part being 0. If we represent a document by storing all the position stamps for the characters it contains, it would take four times larger disk space than storing it in characters.

Our second optimization technique is to represent both PPSs and the position stamps in the $prev_p s$ variable in compact records with the help of a simple labeling scheme. In our implementation, we group consecutively inserted or deleted characters as one edit. Given two position stamp s_i and s_j, if we insert m characters $c_1 c_2 ... c_m$ between them, the m characters will evenly distribute the space $d_{ij} = s_j - s_i$ under

the condition of $d_{ij} \geq (m+1)$. (We address in Section 4.3.4 when the condition is not satisfied.). Based on this labeling scheme, the m characters will be assigned position stamps $s_i + gap, s_i + 2 * gap, s_i + m * gap$ respectively, where $gap = \frac{d_{ij}}{m+1}$. Instead of maintaining each of them individually, we represent them in a compact record $< s_i + gap, gap, m >$, where $s_i + gap$ is the position stamp of the left-most character, gap the distance between consecutive characters, and m the length of the character sequence. Since each field in the record is a 4-byte integer, we only need 3*4 = 12 bytes to represent the position stamps of m characters. Correspondingly, we will insert only one entry into the B+tree with the key being $s_i + gap$ and the value being $c_1c_2...c_m$. This compact representation helps save disk space and speed up the update and lookup performance because it processes position stamps in batch.

The compact representation needs to consider situations when old records are updated. For example, a user inserts characters $x_1x_2...x_u$ into the above m characters or deletes parts of them. In this case, we need to split the old record into sub-records. Let us assume that $x_1x_2...x_u$ is inserted after c_k. The record $< s_i + gap, gap, m >$ is split into two records as $< s_i+gap, gap, k >$ and $< s_i+(k+1)*gap, gap, m-k >$. For the character sequence $c_1c_2...c_kx_1x_2...x_uc_{k+1}...c_m$, its position stamps are represented as $< s_i+gap, gap, k >$, $< s_i+k*gap+gap_1, gap_1, u >$, and $< s_i+(k+1)*gap, gap, m-k >$, where $gap_1 = \frac{gap}{u+1}$. Correspondingly, in the PPS the entry $< s_i+gap, c_1c_2...c_m >$ is updated to $< s_i + gap, c_1c_2...c_k >$ and $< s_i + (k+1) * gap, c_{k+1}c_2...c_m >$. Figure 9 illustrates the state of $prev_ps$ and the PPS after the insertion. Processing of deletes is similar.

4.3.3 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.

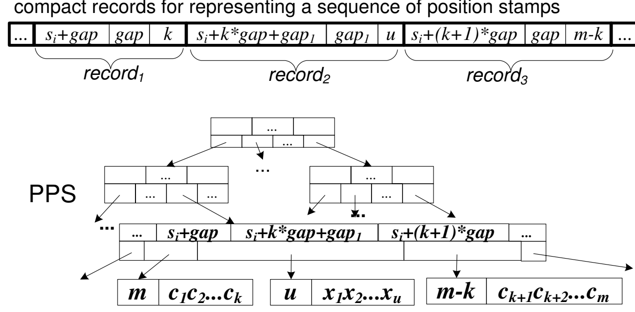


Figure 9: An example of splitting a compact record after an insert

Example 1. *Given a PPS with* $P_0 = \{0, 1\}$ *and* $M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\}$ *at two sites,* $site_1$ *executes* $ADD(0, 1, a)$*, while* $site_2$ *executes* $ADD(0, 1, b)$ *simultaneously. If we use dyadic fraction labeling scheme by halving the interval, both 'a' and 'b' would be assigned* 0.5*.*

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 $d_{i,i+1}$ between two position stamps s_i and s_{i+1}, it is pre-divided into n sub-ranges $(s_i, s_i + \frac{d_{i,i+1}}{n})$, $(s_i + \frac{d_{i,i+1}}{n}, s_i + \frac{2*d_{i,i+1}}{n})$, ..., $(s_i + \frac{(n-1)*d_{i,i+1}}{n}, s_j)$. For site $site_p$, it assigns new position stamps for its local characters within the range of $(s_i + \frac{(p-1)*d_{i,i+1}}{n}, s_i + \frac{p*d_{i,i+1}}{n})$. Taking the above example, $site_1$ will use the space in the range $(0, 0.5)$. $site_2$ 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 essentially uses site identifier to break ties for simultaneous edits happening at the same location. This modified labeling scheme is the one we are currently considering. There may be other options as well. In the rest of the paper, we assume the global uniqueness of position stamps without repeating this property.

4.3.4 PPS Re-initialization

Since users can insert infinite length of character sequence between two characters, a PPS can run out of precision bits if its position stamps are represented in native

machine word. It is established that an *immutable* labeling scheme, in which identifiers never change over document modification, will take $\Omega(N)$ bits per identifier, where N is the document length [40]. Such long identifiers incur high computation cost and disk space consumption, which is unacceptable in collaborative editing scenarios. Many labeling schemes [109] handle this problem by reassigning identifiers when running out of precision bits. The identifiers assigned for the same item will be associated with a unique and immutable identifier. We employ the similar idea for the PPS data structure through a re-initialization procedure.

Essentially, there are two sets of identifiers. The first set contains the position stamps that help us support efficient document update and retrieval and maintain the structural information for characters. The second set contains immutable identifiers that help us associate position stamps for the same character. In a distributed setting, we can use site id and an incremental counter at each site as a pair to create a unique identifier. Each time a character is inserted, it is assigned both a position stamp and an immutable identifier. When re-initialization procedure happens, it first reassigns position stamps for all the characters that have not been deleted from the logical view. The visible characters will re-distribute the space in the range of $[0, 1]$. The procedure also maintains the mappings both the pre-initialization position and after-initialization position stamp of a character with its immutable identifier. The mapping table in Figure 8 stores this information. Immutable identifiers are used to associate meta-information. When users ask queries on editing histories, we first use the position stamps of a document to find their immutable identifiers and finally look up their meta-information.

The re-initialization procedure can be triggered at several moments: 1) when the editing session is over; 2) when no editing is detected; 3) when running out of precision bits. The handling of case 1) can be simply handled by the server site. The practicality of start re-initialization in case 2) and case 3) is based on three conditions.

First, quiescent moments are common in editing scenarios. This is confirmed by various empirical studies [94, 115] showing that collaborative editing scenarios involve a large amount of time for coordination, discussion, and thoughts organization, and it normally happens within a small group of people. The second condition is that re-initialization procedure completes quickly. The third condition is that the re-initialization procedure does not occur frequently in an editing session, and therefore it is likely for the system to detect a quiescent moment before a PPS runs out of precision bits.

4.4 Experiments

4.4.1 Experiment Setup

Hardware configuration All experiments are conducted on a 32-bit GNU/Linux machine with Intel Pentium 4 CPU 2.80GHz and 1GB RAM.

Data set To precisely evaluate the performance of PPSs at various metrics, it requires the availability of real-world collaborative editing traces because the measurements will be impacted by many factors such as the distribution of edit locations and the length of edits. However, we are unaware of any published editing traces. Alternatively, we construct realistic traces by using real-world co-authored documents and their revision histories. The content of a co-authored document is contributed by many users at different points in time. A new revision is created when a user save his edits. With its full revision history, we can compute delta changes between consecutive revisions and replay these changes in their chronological order. We expect the editing traces are able to approximate user editing patterns in reality. We choose co-authored documents from Wikipedia as our data source. All Wikipedia web pages and their revision histories can be downloaded from their website [5]. We used the data dump taken snapshot on March 14, 2008. There are around 200 thousands documents. We do a simple random sampling on a subset of the documents to analyze

their editing traces. Their statistics are shown in Table 1.

Table 1: Statistics of sampled Wikipedia data set for evaluating the performance of the PPS-based real-time collaborative editor

number of documents	1,941
average document size (byte)	8,536
number of revisions	273,587
average revisions per document	141
total number of edits	2,005,468
total edit length[a]	178,339,776
average edits per revision	7
average length per edit	10

[a] edit length means the number of characters modified by an edit.

4.4.2 Disk Space Consumption

In this experiment, we measure the disk space consumption of representing documents and their revision histories in the PPS data structure. We compare PPS with two other systems. The first system, named "File", represents document revisions as individual files disregarding knowledge of any overlapped content. This gives us a measure of the total amount of disk space to manage all document revisions. The other is to represent document revisions in RCS [117]. RCS is a content management system that only stores the delta difference between consecutive revisions to compress data. For each document, we calculate the total number of revisions and the amount of disk space taken by each systems.

Figure 10 shows the result. The x-axis represents the documents with different number of revisions. Each line represents the disk space consumption for a particular system. A point is the disk space used (y-axis) to store a document with a particular number of revisions (x-axis) at that system. It can be seen that "File" system uses several times larger disk space than the two other systems because it does not consider the overlapped content between consecutive revisions. The ratio becomes larger as the number of revision increases. This is because the contents of documents have the tendency of getting stabilized after certain amount of revisions. RCS achieves a

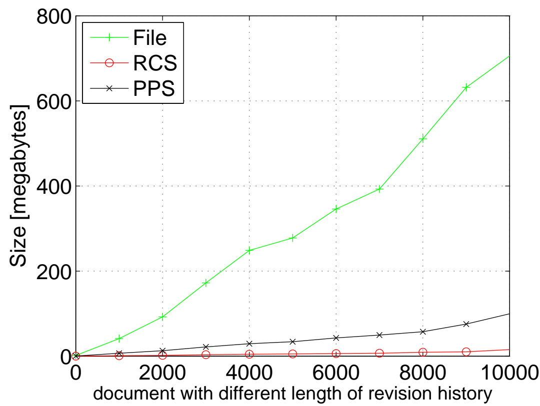


Figure 10: Total disk space consumption for the sampled data set

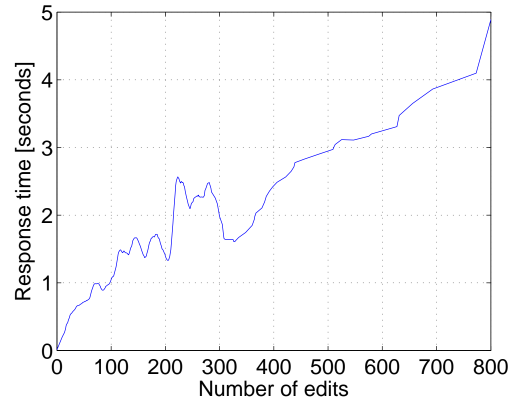


Figure 11: Updating cost from logical view to physical view for a given number of edits

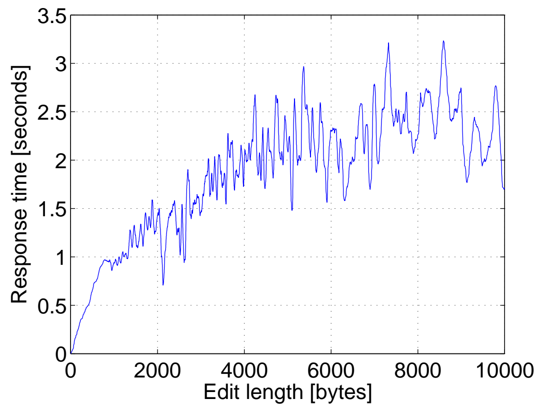


Figure 12: Updating cost from logical view to physical view for edits at different length

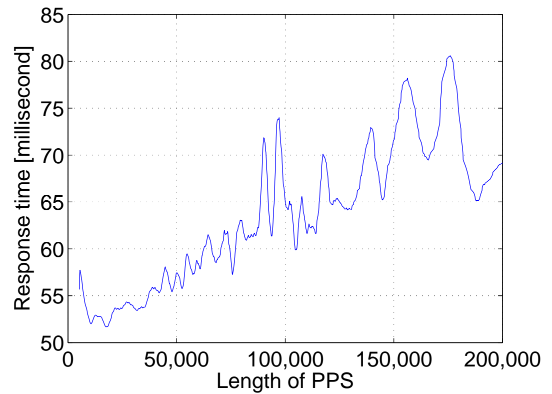


Figure 13: Updating cost from physical view to logical view for PPSs at different length

higher compression ratio because the percentage of overlapped content between consecutive revisions becomes higher. PPS can also benefit from the increased percentage in overlapped content because a majority of edits will be minor or small refining edits and decrease the chance of doing re-initialization. Overall, PPS achieves a compression ratio close to that of RCS with the additional benefit of persistently identifying characters across different revisions.

4.4.3 Updating Cost From Logical View to Physical View

Local edits update a physical view, represented in a PPS, based on a user' edits at the logical view. We use two experiments to measure the performance of this updating cost. The first experiment measures the total amount of time to process a given number of edits (both insert and delete). The total amount of time includes three parts: 1) the time to compare the difference between two consecutive revisions. We use the diff utility [3], which implements Myer's diff algorithm [91]; 2) the time to compute update operations on the PPS; and 3) the time to update the PPS on disk. We collect the experiment data through two steps.

- For a given document, we get all its revisions from our data set and use them to update the PPS in the chronological order of the revisions. For each revision, we measure the number of edits between the current revision and the previous revision and how much time it takes to update its PPS.

- We repeat the first step for all the documents in our sampled data set.

The experiment result is shown in Figure 11. The total processing time increases almost linearly as the number of edits increase. This is because a larger number of edits result in more modifications to their underlying PPSs and also cause more records to be split. It can be seen that the mapping cost is small. Even for hundreds of edits, it takes less than a few second to complete. The processing time is more than sufficient to cope with the speed of human edits.

The first experiment does not consider the factor of edit length. For an insertion, the edit length means the length of inserted character sequence. For a deletion, the edit length means the length of deleted character sequence. The higher the edit length, the longer it will take to process the edit because it involves more I/O and more record processing. The second experiment measures the total amount of time to process edits at certain edit length. The experimental data is collected in a way

similar to the first experiment. The difference is in step 1. For each revision, we measure the edit length of each insert or delete operation and sum them together to calculate the total edit length for each revision update. Figure 12 shows the result. Again the processing time is small. Even for edit length larger than a few kilobytes, it takes around several second to finish. The small processing time on large edit length is important, which means that the editor performs well in the situations that users insert large segment of text through copy&paste or delete a big portion of the document.

4.4.4 Updating Cost From Physical View to Logical View

At each user site, the EPT thread described in Section 4.3.1 periodically refreshes the logical view with remote edits. During its execution, it needs to do a sequential traversal of the PPS and concatenates all visible characters. We measure this updating cost by evaluating total processing time for traversing a given length of PPS. The length of a PPS is the number of position stamps it contains. The traversing cost is equal to that of traversing all leaf nodes in its B+tree. The result is shown in Figure 13. We can see that the updating 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 contains hundreds of thousands of position stamps, but has only hundreds of key-point pairs in general in its B+tree due to the optimization technique of compact records described in Section 4.3.2. Therefore, a large PPS can be traversed very quickly. We also observe that the cost is dominated by disk seek time, not the number of visited leaf nodes because the sizes of B+trees are small. That explains why the cost does not change too much as the length of PPS increases.

4.4.5 PPS Scalability

In collaborative editing scenarios, scalability of PPSs does not raise a concern because updating speed of users is inherently constrained by the users' typing speed, which

on average is around 19 words per minute for composition [77]. The experiments in Section 4.4.3 show that PPSs are by far adequate to process both local and remote updates. However, this may not be the case for the server site in charge of administrative role of an editing session. In realistic settings, we expect the existence of a server site in charge of administrative roles of many collaborative editing sessions running simultaneously. The server site will accept updates from different editing sessions and apply them to their corresponding documents. It is therefore important that the server site can scale up to large workloads. We prototype a text document management system (TDMS) for the server site. TDMS uses a thread pool to process simultaneous edits. Each edit is either an insertion or a deletion, updating one of the co-authored documents. Based on the analysis of our data set in Section 4.4.1, the length of an edit follows a zipf distribution. In other words, the majority of edits modify a few characters. The distribution of edit length in our experiment is shown in Table 2. For each insertion, we use the dummy text generated from Lorem Ipsum [7] to construct a character sequence at a given length. We also implement a workload generator that sends update requests at a configurable rate (i.e. number of requests per second).

Table 2: Edit length distribution in the sampled Wikipedia data set

edit length [byte]	percentage	edit length [byte]	percentage
$[1, 10]$	53%	$[11, 50]$	22%
$[51, 200]$	12.8 %	$[201, 1000]$	6.6%
$[1001, 5000]$	3.4%	$[5001, 10000]$	0.4%
$[10001, 20000]$	0.05%		

We evaluate the throughput of PPSs by simulation. We configure TDMS to maintain 500 documents simultaneously. Each document starts with an initial size of one kilobytes. The simulation runs in cycles. In each cycle, the workload generator sends edits at a gradually increasing rate. We run the workload generator for two minutes and collect the processing time of the requests in the second minute to compute the

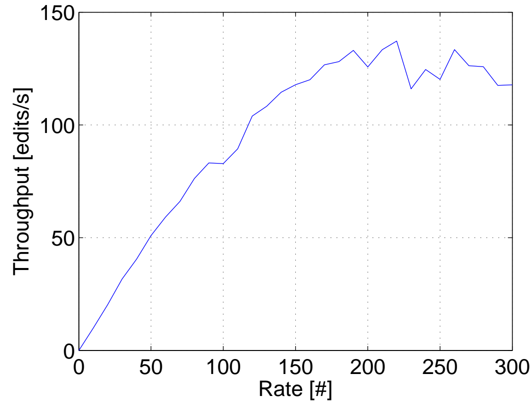


Figure 14: Throughput measure at the server site in the PPS-based real-time collaborative editing system

throughput at a given rate. Figure 14 is the measurement of throughput. TDMS gets saturated at around 125 edits/second. The bottleneck of the system is disk I/O. Berkeley DB stores its tables as flat files on disk. In our implementation, the PPS of each document is stored as an individual file on disk. Berkeley DB has a limited cache size. A higher frequency of workload causes more pages flushed back to disk, which causes more disk head movement. Currently, we use the default cache size at 256KB per PPS. If we assume that all 500 editing sessions are active and update the documents at similar frequencies, it can accommodate the update frequency at every $4 = \frac{500}{125}$ seconds per editing session, which should be fast enough to handle many realistic settings.

4.4.6 PPS Re-initialization

A PPS needs to be re-initialized for recycling position stamps. There are two major steps: 1) obtain the global unique identifiers of all the characters from disk; 2) reassign new position stamps for all the visible characters; 3) store the mapping between their global unique identifiers and their new position stamps on disk. The experiments in this section evaluate the cost of this procedure. To collect the data, we follow similar steps in Section 4.4.3. Each time when a re-initialization procedure happens, we

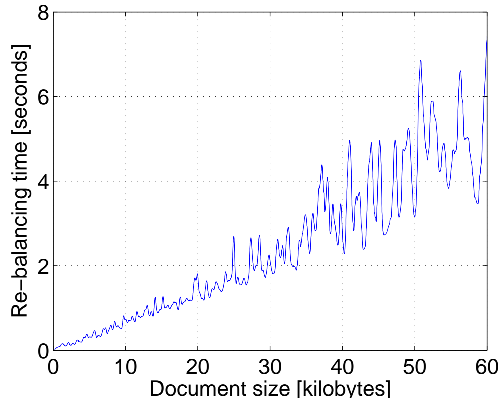


Figure 15: Re-initialization cost for the PPSs

measure the size of the PPS and the total amount of time for finishing the procedure. The result is shown in Figure 15. The re-initialization cost is linear to the length of documents. This is not a problem for documents that are smaller than tens of kilobytes because it takes a few seconds to finish the procedure. However, for large documents, better techniques are required to shorten the re-initialization time, which will be the direction of future work.

4.5 Summary

In this chapter, we address the problem of how to use PPSs to enforce data consistency in real-time collaborative editing systems. A PPS represents a document by creating an order-based index structure for all the characters occurring in its entire editing history. We apply two techniques to make a PPS both disk-space economic and computation efficient. The first technique is to represent the PPS data structure in a B+tree to support efficient random update and sequential retrieval. The second technique is to represent PPSs and their revision histories in compact records. We choose to represent position stamps in native machine words because long position identifiers can have a big impact on system performance. When a PPS runs out of precision bits for newly inserted characters, we re-initialize the whole data structure and reassign position stamps for characters that have not been deleted by users. The

re-initialization procedure can happen at several moments such as the end of editing session or system quiescent time. We conduct a set of experiments to demonstrate the performance of PPSs and the practicality of doing PPS re-initialization based on real-world collaborative editing traces from Wikipedia.

CHAPTER V

MODELING AND IMPLEMENTING COLLABORATIVE TEXT DOCUMENT EDITING SYSTEMS WITH TRANSACTIONAL TECHNIQUES

5.1 Motivation

Collaborative editing systems support geographically distributed users to work on a shared document. They are responsible for coordinating edits of users to guarantee data consistency because users may simultaneously update a replica of the shared documents anywhere and anytime. These systems in general have specialized implementations and only cover a subset of interactions found in collaborative environments. While it is tempting to develop new algorithms and infrastructures to cover the missing points in the full spectrum of collaborations, any such work will lead to ad hoc implementations and substantial investment of resources.

We have developed a transactional framework to model and implement the whole spectrum of collaborations. This new framework has two advantages. First, it provides primitives to program common editing actions (e.g., insert and delete) as well as to specify permissible interactions between users (e.g., cancel the effect of another user). These primitives allow us to conceptually specify different types of collaborations and reason about their behaviors in terms of granularity of sharing, time to release of individual edits to public, notification of editing conflicts, and conflict reconciliation strategy. The generality of our framework is tested by its capability of specifying three types of collaborative editing systems RCS [117], MediaWiki [17], and Google Docs [15]. We further test its generality by using this framework to specify the behavior of a new type of collaboration that is derived by combining features of

Google Docs and the approach of acceptance test in handling conflict reconciliation in replicated database management systems (DBMS) [64].

In the second advantage, the framework can be entirely layered on the top of a modern database management system to reuse its transaction processing capabilities for data consistency control in both centralized and replicated editing systems. In centralized collaborative systems, a document is stored at a central server. Users take turns to modify the document [58]. In more recent collaborative editing systems, a document is replicated at geographically distributed sites. Each site is used by one user to modify its local copy. Users can simultaneously modify the document and read the changes of others. Due to network latency, users may modify different versions of the shared document. An important role of replicated editing systems is to bring all divergent document copies into a convergent and consistent state [57, 113]. Though successful, these early techniques require specialized implementations and only handle a subset of collaborations. Our framework supports the entire spectrum of collaborations by reusing the built-in database techniques in concurrency control, crash recovery, and automatic replica synchronization.

Within our framework, we use PPSs to represent documents and manage them within a database management system. With the help of PPSs, we take the first initiative to define editing conflicts and establish a correctness criterion for collaborative editing systems based on the theory of serializability and the approach of acceptance test for data reconciliation. We also explain the usage of PPSs to support document processing and their implementation issues. We demonstrate the practicality of our framework by building it over Oracle Berkeley DB High Availability [18], a replicated transactional data management system.

In the rest of this chapter, we start with an overview of existing collaborative systems and discuss their potential improvements in Section 5.2. We describe the programming interfaces of the proposed framework and its synchronization protocol

for data consistency guarantees in Section 5.3. In Section 5.4, we illustrate the flexibility of our framework by modeling a variety of collaborative models. Then we explain the application of PPSs to data consistency guarantees in Section 5.5. After that, we describe a prototype implementation over Oracle Berkeley DB High Availability in Section 5.6. The related work is discussed in Section 2.2.

5.2 Overview of Collaborative Document Editing Systems

We observe a wide spectrum of collaborative editing systems. At one end of the spectrum are version control systems that support only restricted collaboration [43]. At the other end of the spectrum are those "liberal" collaborative editing systems that support highly interactive collaboration [57]. In this section, we first describe three collaborative editing systems to give a brief coverage for the type of collaboration available in practice in Section 5.2.1. For each system, we characterize it in terms of granularity of sharing, time to release of individual edits to public, notification of editing conflicts, and conflict reconciliation strategy. After that, we suggest potential improvements to these systems in Section 5.2.2.

5.2.1 Existing Collaborative Editing Systems

Existing collaborative editing systems unanimously adopt the client-server architecture. The server node holds a persistent copy of a shared document. Each client node stores a copy of the shared document. A user at a client node updates the shared document through the local copy. All updates are synchronized to other users through the server node. Below, we describe three collaborative editing systems in the order of their restrictiveness on collaboration.

RCS. It is a version control system. In RCS, a user modifies a document through an explicit check-out step. The document can be checked out by multiple users. Editing conflicts occur if a user attempts to check in a new version whose modifications are

based on a stale version. The granularity of sharing is the whole document. A user releases her edits through an explicit check-in step. RCS uses a locking mechanism to detect editing conflicts and notifies impacted users through diagnostic messages. Even though traditionally being used to handle source code in software development, RCS has been recently used to support wiki applications, e.g., Twiki [19].

MediaWiki. It supports fine-grained collaboration among a group of users who simultaneously edit a shared document. Users edit different parts of a document without interference. Editing conflicts occur if more than one user simultaneously edits the same paragraph. A user releases her edits by manually clicking a *save* button. MediaWiki automatically merges users' changes by *diff3* [11], provided that changes happened in different parts of the document. Otherwise, impacted users are notified with diagnostic messages. MediaWiki is the underlying engine for the largest online encyclopedia, Wikipedia [21].

Google Docs. It supports fine-grained collaboration among a group of users who may simultaneously edit a shared document and at the same time read updates made by other users. Editing conflict occurs if more than one user simultaneously updates the same sentence. A user's updates are automatically synchronized to other users at a fixed time interval (about tens of seconds). Google Docs uses the *differential-synchronization* algorithm [14] to automatically merge changes from different users. The basic idea is similar to *diff3*, but in a streaming fashion. If an automatic merge fails, Google Docs notifies impacted users through diagnostic messages.

In the rest of this section, we refer the collaboration type supported by RCS as the *check-in/checkout* model and the collaboration type supported by MediaWiki the *block-exclusive* model. Finally, we refer to the collaboration type supported by Google

Docs as the *update-anywhere-anytime* model.

5.2.2 Commentary of Existing Collaborative Editing Systems

We comment on existing systems from five aspects. We make it clear if an aspect is only pertinent to certain types of collaborative editing systems. The aspect list is by no means complete. Other aspects such as access control are not addressed in this paper since they are orthogonal to the problem of data consistency.

Atomicity of grouped operations. There are many cases that a user wants to release a sequence of changes in an atomic step, e.g., a *cut* operation followed by a *paste* operation. Current collaborative editing systems have already included or planned to include this feature in some form of *block edits* that allow users to release her edits in a batch. For example, the next release of Google Docs will enhance the current keystroke-by-keystroke synchronization mode with a block-edit mode. However, the block-edit mode is not atomic in the real sense in that it simply buffers a user's edits and sends them to other users in a batch. It is still possible that the buffered edits are only partially executed at remote sites due to system crash or network intermittence.

Undo An undo operation allows a user to go back to a previously edited document state. In a single-user setting, the implementation of undo can be done by logging adequate information for the pre-image and post-image of a document transformed by each editing operation. In a multi-user setting, two problems arise. First, the choice of which operation to undo becomes ambiguous. When a user issues an undo, it is unclear whether the user intends to undo the last operation or undo the last operation received from other users. The problem becomes more difficult if the user wants to undo a sequence of changes which may be interleaved with operations from different users. Second, no standard techniques exist to evaluate and inform users of

the impact of undo. In some situations, an undo may produce dangling text that was inserted into a paragraph which would disappear later on. In some other situations, undo can lead to loss of data. We cannot emphasize more in a collaborative environment the importance of making undo predictable and recoverable. For example, in Wikipedia, if a user replaces the current version of an article with one of its previous versions, some edits between these two versions may get lost.

Infrastructure development The three collaborative editing systems described previously differ a lot in the level of restrictiveness on collaboration. Therefore, it is not surprising that each of them uses different implementation techniques. For example, RCS uses a locking mechanism, while Google Docs uses operational transformation [57] for data consistency guarantees. However, it is important to avoid re-investing new resources each time a new type of collaboration comes out.

Automatic merging in a controlled manner. Collaborative editing systems that fall at the update-anywhere-anytime end of the collaboration spectrum normally do automatic merging of updates at best efforts. Even though this can minimize manual reconciliation from users, automatic merging may produce unintended results which may not get noticed immediately. It is therefore important for the system to be able to limit the amount of inconsistency introduced during a merging procedure.

5.3 A Transactional Framework for collaborative editing Systems

We describe a transactional framework for modeling and implementing collaborative editing systems. Our framework is based on standard transaction services in database management systems such as two-phase locking concurrency control, predicate locking, and write-ahead logging. This framework is applicable to documents consisting

of a sequence of data objects. These objects can be instantiated to suit the requirement of a particular application domain. For example, a data object can be a word in a text document or be a XML element in a serialized XML document. Henceforth, we choose text documents to explain our ideas due to its commonality. But the presented ideas and techniques are applicable to all kinds of documents that bear sequential structures. We first describe the programming interfaces of our framework in Section 5.3.1 and then describe the synchronization protocol for the replicas of a shared document in Section 5.3.2.

5.3.1 Programming Interfaces

There are two sets of programming interfaces for implementing a certain type of collaboration. The interfaces in the first set are used for interacting with a shared document, as described below:

- $Insert(pos, x)$: it inserts a new item 'x' at position pos.
- $Delete(pos)$: it deletes the item at position pos.
- $Read(pos_x, pos_y)$: it reads a range of text between the two items indexed at pos_x and pos_y respectively.

Insert and *Delete* are standard editing operations. Sometimes we call them *write* operation without differentiation. The *Read* operation is new since a user may not explicitly tell the underlying collaborative editing system the dependent data items of new changes. However, the knowledge of the data items in a read operation can be obtained either automatically or manually. In an automatic approach, a collaborative editing system either infers the dependent data items based on application-specific knowledge or uses the standard technique *implicit locking* [92] to locate the area where the user's most recent editing activities took place. For example, in the check-in/check-out model, the read set is the whole document. In the block-exclusive model,

the read set is the paragraph that contains the modified text. In the manual approach, a user selects a block of text and marks them as being read through a Graphical User Interface (GUI) menu entry.

The programming interfaces in the second set are used to instruct our framework to take transaction-related actions, as described below:

- *Release*: it releases a user's changes to other users since the last release point. All the changes are bracketed within a transaction whose execution is guaranteed with the ACID properties.
- *Save*: it saves the current state of the document and returns with a save-point identifier for later references. The *Save* operation triggers the execution of a *Release* as well.
- *SavePivot*: it saves the current state of the document and returns with a pivot-point identifier for later references. The *SavePivot* operation triggers the execution of a *Release* as well.
- *Cancel*: it cancels the last write operation (i.e., insert or delete) that has not been released to other users.
- *Revert*: it changes the current state of the document to a state identified by either a save-point or a pivot-point identifier.

A *Release* operation is useful in controlling the frequency of synchronization with other users. For example, Google Docs may issue a *Release* command each time a timeout event happens for starting the next round of synchronization with the server.

Both a *Save* and a *SavePivot* operation force the framework to save a persistent state of the shared document. These persistent states serve as reference points for a user to undo her changes. They are also useful to reduce the amount of work that a user has to redo during a collaborative editing system failure or a system crash. The

difference is that *SavePivot* sends the framework an additional message that all edits occurring before this point will not be undone by this user. Usually, *Save* is used to commit intermediate edits while *SavePivot* is used to commit milestone edits.

Our framework explicitly differentiates two types of *Undo* operations. A *Cancel* undoes the last operation by the local user. Since it has not been released to other users, the last operation can be simply removed from the messaging sending queue of the client. However, a *Revert* operation requires synchronizations with other users since it may undo the changes on which other users' edits depend. The save-points and pivot-points created by a user are globally visible, which means a user can bring the state of a shared document back to a point saved by other users as well. However, any save-point before the last pivot-point of a user becomes unavailable.

5.3.2 Synchronization Protocol Between Client and Server

Our framework uses an optimistic synchronization protocol based on the two-tier replication scheme in [64]. The server hosts the master copy of a shared document. Each client node hosts a copy of the shared document. The master copy reflects the most recent committed updates from all the users. The client copy may be the latest or an old version of the master copy. All transactions committed at the client nodes are *tentative*. They are sent to the server and executed under single-copy serializability in the order in which they are committed at the client node. A tentative transaction becomes a *base* transaction if it is committed at the server node and its effects are integrated into the master copy. The write set of all base transactions are sent to the client nodes and update their replicas in the order they are committed. Since the server node determines a global serializable order for all tentative transactions, document replicas converge to the same state and each of them has a consistent view of the document state.

Regarding the choice of concurrency control algorithm for enforcing the single-copy serializability at the server node, we choose the approach of *acceptance criterion test* in [64] instead of multiversion concurrency control algorithms. Under the two-tier replication scheme, it is possible for a tentative transaction to see a very stale version of the shared document. For example, a user may exit an editing session, edit offline, and re-join days later. During the user's absence, the shared document has gone through many rounds of revisions and many tentative transactions have already committed. To determine serializability for the tentative transactions the user committed offline, a multi-version scheme needs to check both active and committed transactions. The examination cannot simply be done by usual lock conflict check because these committed transactions no longer hold their locks.

The idea of acceptance criterion test is to check whether the result produced by a tentative transaction based on the version at the server node is within an acceptable threshold. We take the first initiative to define such a criterion for collaborative editing systems. In our acceptance criterion, a tentative transaction is considered to be *acceptable* if the difference between the set of data items that it reads at the client node and the set of data items that it reads at the server node is within a configurable threshold θ. We assume that a *write* operation is always proceeded by a *read* operation. There are no blind writes. Therefore, we can use the read set of data items to quantify the divergence between these two versions. A quantitative definition of $Accept^{\theta}$ is given in Section 5.5.1 after introducing the PPS data structure.

We use $Accept^{\theta}$ to mean the acceptance criterion is passed if the difference is within θ. $Accept^{0}$ means that a tentative transaction must read exactly the same set of data items at the server node. $Accept^{\infty}$ means a tentative transaction can tolerate arbitrary divergence between the data items read at the client node and those at the server node. Of course, there are cases that a write operation totally lost its context and cannot be applied at all. For example, a delete operation attempts to remove

an already deleted item. We will come to this issue in Section 5.5 and show that all write operations can be precisely defined with the help of PPSs.

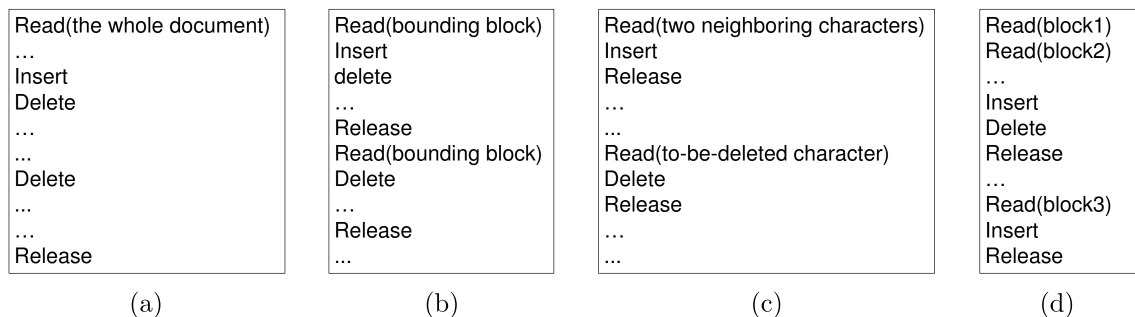


Figure 16: Examples of synthesized code. a)Check-in/check-out; b)Block-exclusive; c)Update-anywhere-anytime; d)Read-from

5.4 Specifying Collaborative Editing Systems

In this section, we demonstrate the usage of our framework in modeling three editing models described in Section 5.2.1. To demonstrate the flexibility of our framework, the modeling of an artificial editing model is also described.

Check-in/Check-out Model. In this model, a user modifies a shared document through a sequence of editing operations and releases new changes through a check-in step. We synthesize this model as in Figure 16a. The acceptance criterion of the server node is configured to be $Accept^0$. Therefore, if someone modifies the shared document and creates a new version, this transaction will be aborted. In the synthesized code, there is only one *Release* operation, which is the last operation within an editing session. In a standard check-in/check-out model, a user may save multiple versions before issuing the *Release* command. These intermediate versions are not visible to other users. They are different from those versions created through *Save* and *SavePivot* operations. We assume that these intermediate versions are created in a private space of the user and are handled completely by a standard text editor.

Block-exclusive Model In this model, a user's edits are sent to the server either at a fixed time interval or through a manual click of a "send" button. Both events cause the execution of a *Release* command. Users do not interfere unless they work on the same part of a document. We synthesize this model as in Figure 16b. A *Read* operation is followed by a sequence of write operations that updated the text within the range of the *Read* operation. A *bounding block* consists of the read text. Its content is application specific. For example, in MediaWiki it is the paragraph where these write operations took place. In Google Docs, it is the sentence. The acceptance criterion is set to be $Accept^0$.

Update-anywhere-anytime Model. In this model, users update the shared document without any restriction. All editing conflicts are automatically reconciled. We synthesize this model as in Figure 16c. Every write operation is followed by a *Release* to synchronize the document replica at the frequency of every keystroke. Each transaction is essentially reduced to a read operation followed by a write operation. For an *Insert*, its read set contains only the two characters neighboring the insertion point. For a *Delete*, its read set is exactly the character to be deleted. The acceptance criterion is configured to be $Accept^\infty$. Since θ is set to be ∞, the framework essentially enforces read-committed isolation [123] because each tentative transaction only reads the data written by committed transactions based on our synchronization protocol described in Section 5.3.2. Under read-committed isolation, transactions are susceptible to lost updates and phantom problems. More specifically, it is possible that two users simultaneously delete the same data item or insert new items at the same location. In Section 5.5.1, we explain in detail how our framework is able to produce the same result as that of operational transformation when $\theta = \infty$. Since all document replicas are updated in a global serializable order and all tentative transactions are applied in the order they committed at the client nodes, both the convergence property and the causality preservation property are preserved.

Read-from Model. We introduce a new editing model to demonstrate the flexibility of our transactional framework. In this model, a user can select blocks of text by the mouse in different parts of a shared document and notify the system that the follow-up changes depend on them. The user releases new changes at a fixed time interval or the click of a "send" button. This model is synthesized as in Figure 16d. When the server merges the user's new edits, the user is willing to accept the result if the text the user read is only slightly different from the original. In this case, the θ is set to be a small positive integer. This model has two distinct features. First, a user can monitor the changes in other parts of the document without blocking other users from editing. Second, the model is able to quantify the discrepancy between what a user has viewed and what is actually produced. This feature is useful because it creates a smoother editing environment since the user will not be asked for manual reconciliation if other users only did minor changes to the text such as grammar or spelling corrections. Meanwhile, the user has the assurance of being notified for big changes.

5.5 *Implementing the Synchronization Protocol Based On PPSs*

Partial persistent sequence (PPS) is a data structure that always preserves the previous version of a sequence when it is modified, but only the latest version can be modified [128]. We start by a background introduction for PPSs and then explain how to use it for document processing. After that we explain the usage of PPSs to realize the synchronization protocol of replicated collaborative editing systems and the handling of reverts. Finally, we discuss the implementation issues of PPSs.

5.5.1 Enforcement of the Synchronization Protocol

The PPS data structure has two important properties which make it an attractive candidate for enforcing data consistency in collaborative editing systems. First, position stamps are unique and consistent to the sequential structure of a document. Therefore, they can be used as primary keys to store a document in a DBMS. All editing operations can be represented as standard database operations and executed by the DBMS in a conventional way. Second, a PPS never deletes any data items. This property makes it possible to reconstruct any version of the PPS to detect editing conflicts in a replicated setting. In this section, we explain how to efficiently validate the acceptance criterion mentioned in Section 5.3.2 based on PPSs.

Given a tentative transaction t defined on the version (S_u, M_u) of a document replica, let the version of the master copy at the server be (S_v, M_v). The acceptance criterion test checks whether the editing distance between the data items read on (S_u, M_u) and the data items read on (S_v, M_v) exceeds the threshold θ, as defined below:

Definition 5. ***Acceptance criterion*** *$Accept^\theta$. Given a transaction t defined on (S_u, M_u), we say that t passes the acceptance criterion of $Accept^\theta$ on (S_v, M_v) if*
$\sum_{read(s_i,s_j)\in t} Diff(LV([s_i, s_j]_u), LV([s_i, s_j]_v)) \leq \theta$,
where $Diff$ is a difference algorithm.

Since each $Read(s_i, s_j)$ only contains the position stamps at the two end points for the range of text a transaction read, it does not provide adequate information for correct validation. For example, in Figure 1 the logical view of PPS_1 is "*ab*" and the logical view of PPS_2 is "*abc*". They have different views between $[0.3, 0.6]$. With only $Read(0.3, 0.6)$, it is unsure whether they have the same set of visible data items. However, it turns out we can design a correct validation algorithm by introducing some version information.

```
AcceptTest(t, V_client, θ)
1  diverge ← 0
2  FOR each Read(s_i, s_j) ∈ t DO
3      A ← read all position stamps between s_i and s_j
4          FOR each s_x ∈ A DO
5              IF V_server(s_x) > V_client THEN
6                  diverge ← diverge+1
7              IF diverge > θ THEN
8                  abort
9  FOR each write operation o ∈ t DO
10     execute o
```

Figure 17: The algorithm for validating $Accept^{\theta}$ for transaction t

In the client-server synchronization protocol, the server maintains a version counter V_{server}. We use $V_{server}(s_x)$ to represent the version that s_x was last written by a committed transaction. Each client maintains a local version counter V_{client}. When a tentative transaction is sent to the server node, it includes the value of V_{client} as well. The server validates all tentative transactions by the algorithm $AcceptTest$ in Figure 17. The $AcceptTest$ checks whether any position stamps within $[s_i, s_j]$ are updated by transactions committed after V_{client}. Each time it detects a new update, it increases the variable $diverge$ (line 5-6). If $diverge$ exceeds θ, the whole transaction is aborted (line 7-8). Otherwise, the transaction will be executed as normal (line 9-10).

$AcceptTest$ is executed as a standard transaction by the DBMS. In the prototype of our framework, position stamps are implemented by the access method B+-tree within the DBMS. Therefore, the range scan procedure (line 3-8) can be done atomically, which guarantees that the correctness of the acceptance criterion test is not compromised.

$AcceptTest$ provides a sufficient, but not necessary condition for validating $Accept^{\theta}$. It is possible that $AcceptTest$ aborts a transaction, which turns out to be acceptable by $Accept^{\theta}$. As shown in Figure 1, PPS_1 and PPS_3 have the same view, but $AcceptTest$ will abort a transaction if it reads $Read(0.3, 0.6)$ under $Accept^{0}$. However,

AcceptTest provides a practical solution because it adds negligible network communication overhead for *Read* operations.

When $\theta \neq 0$, the editing system admits non-serializable interleaving of transactions. For example, a transaction tries to delete data items that have been deleted or do an insert at a position containing unseen items inserted by previously committed transactions. Our framework handles these situations as follows. For a *Delete*, it will be executed as normal because a *Delete* operation is mapped to $SetState(s_x, false)$. In the PPS, it is mapped to write the *state* of s_x to $false$ multiple times. From a user's perspective, the data item is deleted exactly once. When it is an $Insert$, the server first checks whether there are any items between s_i and s_{i+1}. If no new position stamps are present, it does the $ADD(s_i, s_{i+1}, x)$ by inserting a new position stamp s_{new} as usual. Otherwise, the server will query the DBMS to get the next position stamp s_k greater than s_i and does $ADD(s_i, s_k, x)$ instead.

5.5.2 Revert Handling

A *Revert* operation reverts the state of a shared document to a previous *save-point* or *pivot-point*. When the server receives a *Revert* operation, it checks its log entries and locates all the transactions committed after that point. If the revert point is located before the most recent *pivot-point* in the server's log, the server will abort this transaction and respond back to the client along with the identifier for the most recent *pivot-point*. The client can optionally resubmit the revert request with this new reference point. Let $o_1 o_2 ... o_n$ be the sequence of operations that need to be reverted. The compensating transaction is constructed as $\overline{o_n}\ \overline{o_{n-1}} ... \overline{o_1}$ based on the following rules:

- if o_i is a *Read*, its compensating operation is $\overline{o_i} = \phi$, which is simply ignored.

- if o_i is a $SetState(s_x, state)$, its compensating operation is $\overline{o_i} = SetState(s_x, \overline{state})$;

The compensating transaction undoes, from the user's perspective, any operations that are performed by the transactions committed after the reverted point. A big advantage of handling *Revert* based on PPSs is that the construction of a compensating transaction is completely operational.

5.5.3 Implementation Issues for PPSs

The previous discussion for PPS assumes that data items are never removed and a machine has unbounded precision bits for representing position stamps. While this is valid from a theoretical point of view, which enables us to explain the framework in a concise way, it is rare in practice that collaborative editing systems allow its data to grow unbounded. Therefore, a garbage collection algorithm is used to periodically rebalance the PPS data structure and reassign visible data items with new position stamps.

The server starts the garbage collection process when any of the three events happens: 1) the data storage for the PPS exceeds a threshold; 2) the PPS runs out of precision bits; 3) all users exit an editing session. The server starts a distributed consensus algorithm such as two-phase commit to coordinate the garbage collection process. The server maintains the pre-image and post-image of a PPS at the end of the process and maintains the mapping between the old position stamps and the new position stamps for visible data items. Therefore, if a client node submits a transaction based on an old PPS, the server can use the mapping to determine the right data items to update. Each rebalanced PPS is uniquely identified by a *rebalance-identifier*. All document replicas maintain the *rebalance-identifier* for its local PPS and will include it in all the transactions sent to the server.

Even though the garbage collection process uses a distributed synchronization algorithm, we do not expect it to raise much concern. A user is able to continue her regular edits since all transactions are tentatively committed on its local copy. The

garbage collection only delays the time of synchronizing new changes to the replicas of other users.

5.6 Collaborative Editing System Prototype

We have implemented our transactional framework over Oracle Berkeley DB High Availability. In this section, we first provide a background description for this replicated DBMS in Section 5.6.1 and give an overview of our system architecture in Section 5.6.2. We then explain different modules of our framework in Section 5.6.3.

5.6.1 Oracle Berkeley DB High Availability Infrastructure

Oracle Berkeley DB High Availability enables replication of a database across a collection of nodes. These nodes form a replication group. Within the group, one node is elected to be the master, while the rest of the nodes are referred to as replica. The master node accepts both read and write transactions, while the replica nodes accept read-only transactions. A replica node communicates with the master node through a logical replication stream that contains a description of the logical changes of the master node. The stream is replayed at the replica using an internal replay mechanism. In our implementation, a client node maintains the state of the shared document in a replica node, while the server node maintains the state of the shared document in a master node.

5.6.2 System Architecture

In our implementation, a shared document is replicated across a collection of client nodes and one server node. Each client node is used by one user to modify the shared document. The server node is responsible for integrating changes from all client nodes and replay these changes to all replicas. Figure 18 shows the system architecture between a client node and a server node. When a user issues new edits, the user sees their effect immediately. Meanwhile, these edits are wrapped in the

form of transactions and forwarded to the server node. The server node processes each transaction in two steps: 1) run it against an acceptance test; and 2) execute the transaction in the master node if it passes the acceptance test, otherwise abort the transaction. Meanwhile, the changes at the master node streams to all replica nodes. Each client node periodically refreshes its document copy based on the latest state of its replica.

Oracle Berkeley DB High Availability provides several benefits for developing collaborative editing systems. First, atomicity is a given-in property in transactions. Second, our synchronization protocol can be completely implemented based on the available concurrency control algorithm. Third, the replicated DBMS simplifies recovery. If a client node restarts after a crash, its replica is automatically brought to the latest state of the master node. Finally, the DBMS handles durability automatically for a collaborative editing system. The update of a user is guaranteed to be persistent as soon as it commits at the master node.

5.6.3 Implementation Modules

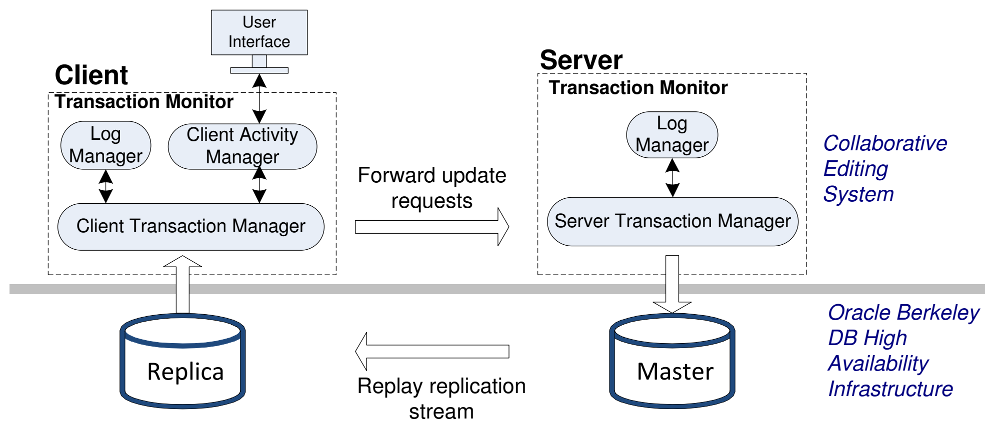


Figure 18: System architecture

We have implemented a transactional monitor at both the client side and the server side to synchronize distributed editing activities. The interaction of these modules is illustrated in Figure 18. Below we describe each of them.

Client Activity Manager (CAM). It receives a sequence of operations from the text editor. When it sees an operation of *Insert*, *Delete* or *Read*, CAM appends it to a buffer. Otherwise, it takes the following actions:

- For a *Release*, CAM wraps all the operations in the buffer and brackets them within the two control operations *Begin-transaction* and *End-transaction* and sends it to the underlying transaction manager. Then CAM empties the buffer. The *Begin-transaction* and *End-transaction* are used to indicate the beginning and the end of a classic transaction.

- For a *Save* or a *SavePivot*, CAM takes an action similar to the handling of *Release*, except that it additionally includes a *Save* or *SavePivot* as the last operation within the transaction.

- For a *Cancel*, CAM removes the last entry from its buffer.

- For a *Revert*, CAM brackets this operation parameterized with its *Save* or *SavePivot* within *Begin-transaction* and *End-transaction* and sends the transaction to its underlying module.

Client Transaction Manager (CTM). It is responsible for forwarding transactions received from CAM to the server and monitoring their progress. CTM maintains all pending transactions in a queue and waits for responses from the server. CTM assumes that the server responds to pending transactions in the order they are sent. On receiving a response from the server, it removes the transaction from the head of the queue. If the response is a commit, it takes no action since the transaction has committed at the master node and is going to be replayed at its local replica. If the response is an abort, it generates a diagnostic message to the user. The abort a transaction may cause the abort of subsequent pending transactions that read the results

of the aborted transaction. If a cascading abort happens, all the aborted transactions are removed from the queue and their states will be included in the diagnostic message.

Server Transaction Manager (STM). It is responsible for processing all client transactions under single-copy serializability. Upon receiving a transaction, STM forwards the *Begin-transaction* and *End-transaction* as well as the document editing operations to its underlying DBMS where the transaction is processed in a conventional way. Due to simultaneous editing, a client transaction may see a different version of the shared document and produces different results. To quantitatively measure the divergent distance, STM runs all client transactions against the *acceptance criterion test* introduced in Section 5.5.1. If passed, the transaction is committed, otherwise get aborted. STM then returns its state to its corresponding client node.

Log Manager (LM). It maintains log entries for the execution history of transactions. Each log entry contains the read and write set of a transaction. To support *Cancel*, the log entries of a transaction are backward chained to identify operations within a transaction. LM also maintains a special *save-point* or *pivot-point* log entry as a marker in its log for handling *Revert* operations.

5.7 Summary

We propose a transactional framework for modeling and implementing collaborative editing systems. Our framework demonstrates its advantages in two ways. First, it provides a conceptual framework to specify the entire spectrum of collaborations for document editing systems. We demonstrate its generality and flexibility through its capabilities of specifying three types of collaborative editing systems and a new collaboration model. In the second advantage, our framework can be layered on the top of a database management system to reuse its transactional techniques for

data consistency guarantee in both centralized and replicated collaborative editing systems. This is demonstrated through a prototype implementation over Berkeley DB High Availability, a replicated database management system.

CHAPTER VI

FINE-GRAINED DOCUMENT PROVENANCE MANAGEMENT ON COLLABORATIVE TEXT DOCUMENTS

6.1 Motivation

Document provenance describes how a document was updated over time. Current revision control systems manage provenance data at the document level, which is too coarse-grained to retrieve provenance information at a finer granularity such as a word or a sentence. An example of fine-grained provenance query is 'Return who contributed which part of a document". This query is useful when we want to know the authorship of a collaborative document such as a Wikipedia article or a source code file. More sophisticated fine-grained provenance queries are "Return all the text that has even been contributed from John" or "Find who deleted the text that John inserted on last Monday". In general, *fine-grained document provenance* describes how *a portion of a document* was updated over time, which is the focus of this chapter.

Fine-grained document provenance is valuable for many applications ranging from document processing in business processes [69] to deep knowledge discovery for text documents [27, 49]. Although fine-grained document provenance is valuable, storing and querying provenance can be expensive. Current revision control systems keep track of the revision history for every committed version in terms of its content, its creation timestamp, and its creation author. Some systems choose sequential files and only store the delta changes in a new version, e.g., Subversion [24]. Some systems use a database to save the full content of every version, e.g., MediaWiki [17]. Regardless of

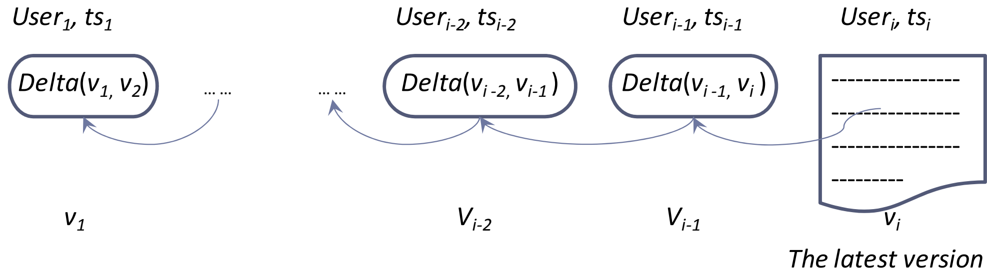


Figure 19: Organize the revision history of a versioned document based on delta changes

their implementations, these systems unanimously track provenance at the document level as shown in Figure 19. To answer any fine-grained document provenance query with such an organization of provenance data, we have to parse the revision history of documents and analyze delta changes of consecutive versions, which could become expensive for documents with a large number of versions. Take the Wikipedia article titled "iPod" as an example. The article was around 52 kilobytes by January, 2010 and had more than 13000 versions.

To avoid the cost of parsing long revision histories, we may choose to store provenance data side by side with different pieces of text for every version of a document. In this way, we can quickly retrieve document provenance for every version. However, we face the problem of overuse of disk space due to the redundancy of saving provenance information for the same piece of text multiple times in different versions of a document. Obviously there is a trade-off between disk space usage and query processing cost. It is always tempting to trade disk space for the performance of query processing. After all, hard drives are cheap and text documents are usually small. However, the trade-off becomes less obvious when we have to deal with millions of documents, some of which have thousands of versions.

We use PPSs to design both disk-economic and computation-efficient techniques to manage fine-grained document provenance. Our approach is disk-economic because we only save a few number of PPS views at different points in the revision history

of documents. For versions between those points, only their delta changes are saved. Our approach is also computation efficient because we build indexes for provenance data at the content-level and avoid the necessity of parsing the revision history of documents. Based on PPSs, we build a system to manage document provenance for millions of Wikipedia articles and evaluate both disk space usage and querying cost for several common document provenance queries. Our experiments show that our system uses less than 10% of the disk space compared to MediaWiki [17], the database engine for Wikipedia. For query processing, we compare our approach with the on-the-fly approach which read and analyze revision histories into main memory in order to answer fine-grained document provenance queries. Experiments show that our approach outperforms the on-the-fly approach on documents with more than hundreds of versions. For documents with more than thousands of versions, our PPS-based approach is able to achieve at least an order of magnitude speed-up on common document provenance queries.

In the rest of this chapter, we first describe fine-grained document queries and explain how to process these queries based on PPSs in Section 6.2 and then describe our system implementation in Section 6.3. In Section 6.4, we evaluate the impact of the size of a PPS view range to disk space cost and query processing cost. Finally we present the experiment results in Section 6.5.

6.2 Fine-grained Document Provenance Queries

6.2.1 Classifying Fine-grained Document Provenance Queries from the perspective of the temporal dimension

Basic document provenance data include what type of a change was (e.g., an insert or a delete), who made the change, and when was the change made. Some applications need to consider additional provenance data. For example, Wikipedia records IP addresses of authors as well. In mobile applications, geographical locations of users may be considered. There are two different kinds of document provenance queries to

retrieve these provenance data.

- **Snapshot Document Provenance Queries**: they produce provenance information for a document at a particular time. An example of this type is to "Return the text authorship of a document in the i-th version.

- **Delta-change Document Provenance Queries**: they produce provenance information for a document related to the state change of any two versions. An example of this type is "Return the author name for the text that was deleted in the newly committed version". Sometimes we may want to examine delta changes for non-contiguous versions as well. For instance, in software products, we want to evaluate the changes of source code files at different milestones.

With the above two types of provenance queries, we are able to answer different kinds of composite provenance queries. For example, Halfaker et. al. [67] uses the provenance data related to the delta changes of every consecutive versions to analyze the neutrality of peer review in Wikipedia. In this example, we repeatedly issue a delta-change document provenance query to get the information. In some situations, we need to further process the result through aggregation and filtering. For example, Adler et. al. [27] evaluate user reputation evaluation based on their past contributions. In this example, we first obtain provenance data for all those items that are modified within the interested period and then aggregate the provenance by the user attribute.

6.2.2 Processing Fine-grained Document Provenance Queries based on PPSs

PPSs play two important roles in processing fine-grained document provenance queries. First, we use PPSs to represent the revision history of documents. Second, we use position stamps to index provenance information of items, which contains a list of data such as author identifier, insert timestamp, and delete timestamp. Correspondingly, there are two steps to process a fine-grained document provenance query. In

the first step, we use PPSs to locate relevant position stamps by using the algorithms described in Section 3.3 . In the second step, we use the identified position stamps to look up provenance information for these items. For snapshot document provenance queries, we use the algorithm CALCULATE-VIEW to retrieve all position stamps visible at a particular time. For delta-change document provenance queries, we use CALCULATE-DELTA to retrieve all position stamps related to the delta changes between two versions.

Depending on the length of a document's revision history, the PPS view range array can become too large to fit into the available memory. As a result, we have to store PPS view ranges on disk and read them into memory as needed. In our design, the revision history a document is represented as an array of fixed size PPS view ranges. Within each PPS view range, we put as many delta changes as we can. If no adequate space to save new delta changes, a new PPS view range is created and the view pivot array is updated to store a pointer to this new PPS view range. For each PPS, only its view pivot array is kept in memory to speed up locating the right PPS view range. The size of PPS view pivot array is normally small. Based on our experiences with Wikipedia articles, for documents with hundreds of versions, their view pivot arrays have about several tens of entries. For documents with thousands of versions, their view pivot arrays have about a few hundreds of entries. Therefore, we are able to keep view pivot arrays for a large number of PPSs. After identifying the right PPS view range, the entire PPS view range is read from disk into memory for further processing. PPS view range is always read from disk and write to disk as one unit. Therefore, we need to be very careful in choosing its size. If its size is too small, only a few number of delta changes can be put into it. We end up with saving too many PPS views, which cause the problem of overuse of disk space. If its size is very large, we end up with saving a long list of delta changes within a single PPS view range, whose size could become too big to fit into the available memory. In addition,

the computation cost for some of these algorithms described in Section 3.3.4 could become too expensive. As we have already shown in Section 3.3.4.3, the computation cost on delta changes takes processor time proportional to $O(n^2 l)$, which could exceed disk I/O for a large n.

6.3 System Implementation

To efficiently process the fine-grained document provenance queries, we face three challenges. First, we need to quickly reconstruct the state of a document as well as the change of states at any time. Second, we need to quickly locate the provenance data related to the state or the change of states. Third, we need to aggregate or filter the provenance data efficiently. Corresponding to these three challenges, we design three modules to process fine-grained document provenance queries as illustrated in Figure 20.

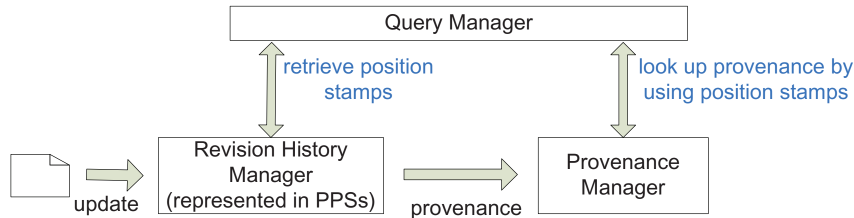


Figure 20: System Architecture

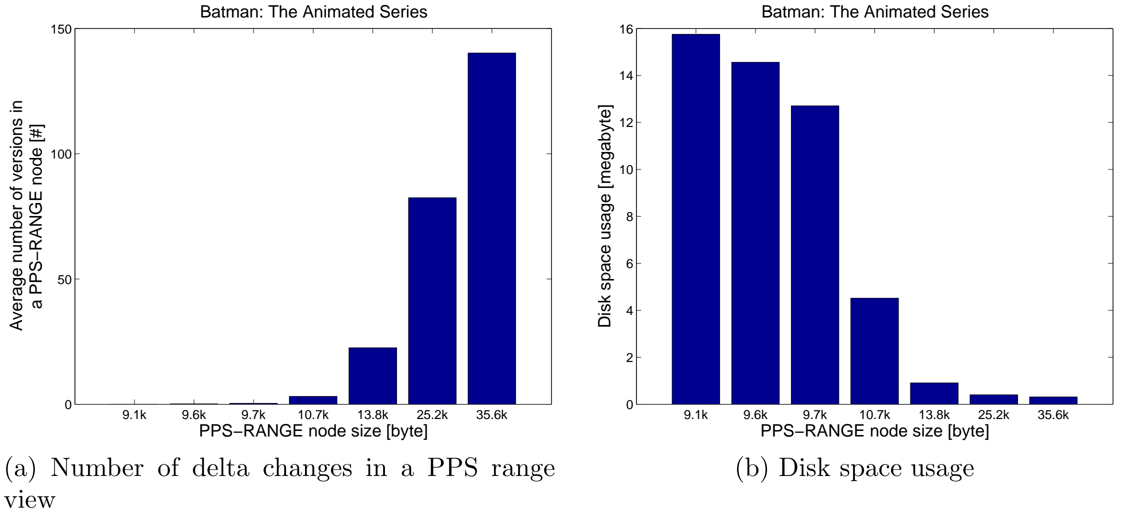
.

Revision History Manager (RHM) It manages the revision history of documents with two responsibilities. The first responsibility of RHM is to store revision histories of documents in PPSs. Given a new version of document, RHM uses diff utilities to identify the changes and represent them in a PPS. These changes are indexed by position stamps and sent to Provenance Manager for further processing. The second responsibility of RHM is to query PPSs to return all those position stamps relevant to provenance queries. In order to lower the querying cost on revision histories, we design an auxiliary data structure based on PPS to speed up the querying process, which will be detailed in the next section.

Provenance Manager It manages provenance of documents. For each document, it uses the position stamps of items to index provenance data in the form <position stamp, [data_1, data_2, ...data_n]>. Since a document may contain many number of items and large set of provenance data, we create several buckets to store a subset of these pairs. Each bucket stores its assigned pairs in a hash table that is serialized or de-serialized between memory and disk to feed provenance data. After receiving all the identified position stamps from the previous module, MM load corresponding hash tables into memory, look up the required provenance by using these position stamps as keys and output the result to the next module.

Query Manager It interacts with RHM and MM in steps and processes provenance data to transform them into required forms. If all provenance data can be processed in memory, we use common main-memory data structures such as hash tables or balanced tree to do aggregation and filtering. Otherwise, we have to rely on on-disk algorithms to process large data set.

When designing these modules, one important design decision we made is to manage metadata by ourselves instead of using a general database engine. The reason is that documents generate many items especially at very fine granularity such as a word. Use Wikipedia as an example, a regular size of a Wikipedia article currently has more than ten thousands of words. If we put all of these words into a database and index their provenance data by position stamps, for tens of millions of documents, we end up with billions of records in database, which cause serious performance problem especially when we attempted to build multiple secondary indexes over the metadata. Another problem we faced when using a database is to the maintenance of the schema. Since the provenance data have unfixed number of entries, managing the schema will become a problem if we are going to handle document provenance from different application domains.



(a) Number of delta changes in a PPS range view

(b) Disk space usage

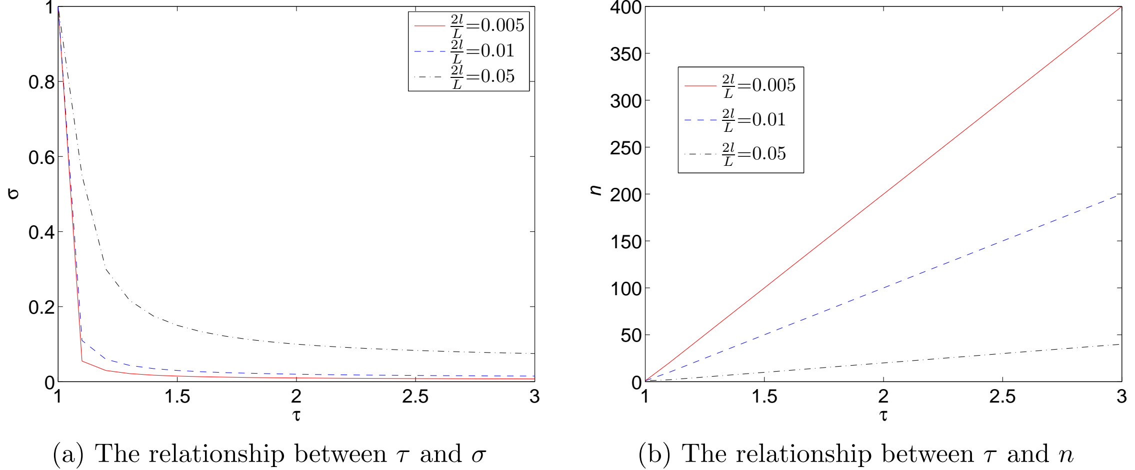
Figure 21: Disk space usage under different configurations for the size of PPS view ranges

6.4 Performance Impact due to Size of PPS View Ranges

In this section, we study the impact of the size of PPS view ranges to disk space usage and query processing cost. We first look at the disk space usage and processing cost for a regular Wikipedia article and draw some rule of thumb for setting size of PPS view ranges based on the case study.

We choose the Wikipedia article titled "*Batman: The Animated Series*" because many Wikipedia articles have similar characteristics. This article had about 2000 version by January 31, 2010. On average, each version has around 2200 items. The delta changes between consecutive versions are about 45 items.

Figure 21(a) shows the number of PPS views we are able to put into a PPS range view under different configurations for the size of PPS view ranges. It shows that when the size is less than 10 kilobyte, the number of delta changes that can be put in a PPS view range are very small. The reason is that the article "*Batman: The Animated Series*" has about 2200 items at round 9 kilobyte. Therefore, we are able to put only a few number of delta changes in a PPS view range. As we increase the size configuration to up to 36 kilobytes, the number of delta changes



(a) The relationship between τ and σ

(b) The relationship between τ and n

Figure 22: Impact of the size of PPS view ranges

increase significantly. Figure 21(b) shows the disk space usage under different size configurations. As expected, as the number of delta changes in a PPS view range get increased, the amount of disk space usage drops significantly. From these two figures, we conclude that we should always set the size of a PPS view range large enough to put one full PPS view and at least tens of delta changes. For example, when we set the size of PPS view range to 13 kilobyte, we already save 90 percent of disk space compared to the approach of saving each PPS view individually.

Next let us derive some general guidance for setting the size of PPS view ranges. Let N be the total number of PPS views, L the average number of position stamps per PPS view and l the average number of position stamps in the delta changes of two consecutive versions. Let M be the size of a PPS view range and n the number of delta changes within the PPS view range. The disk space used to store the N number of PPS views in PPS view ranges is $\frac{NM}{n}$. Let σ be the ratio between the disk space used by all PPS view ranges and the disk space used by all PPS views and $\tau = \frac{M}{L}$ be the ratio between M and L, we have

$$\sigma = \frac{NM}{nNL} = \frac{M}{nL} = \frac{\tau}{n} \implies n = \frac{\tau}{\sigma}$$

Also We have

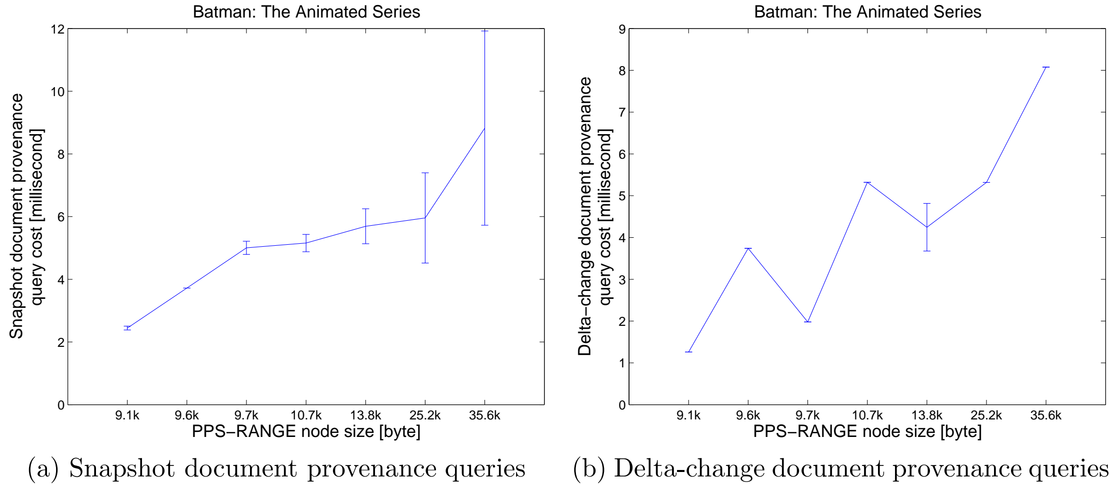
$$M = L + 2n * l \implies n = \frac{M - L}{2l} = \frac{(\tau - 1)L}{2l}$$

By putting above two equations together, we get

$$\sigma = \frac{2l}{L} \cdot \frac{\tau}{\tau - 1}$$

$$n = \frac{L}{2l} \cdot (\tau - 1)$$

Figure 22(a) shows the relationship between τ and σ under three different values $\frac{2l}{L}$. When $\frac{2l}{L}$ is small, by setting the size of PPS view ranges to be $1.5 \sim 2$ more than the average size of PPS views, we are able to save more than 95% of disk space already. For larger $\frac{2l}{L}$, the saving is less significant as the smaller $\frac{2l}{L}$, but still very impressive at 80% with $\tau = 1.5$. Another important observation is that the most significant saving happens at the beginning as we increase the value of τ. After that, we still save more disk space, but at a much lower rate. On the other hand, an increase in τ will lead to more delta changes put into a PPS view range. Figure 22(b) shows the relationship between τ and n. It shows that the smaller the ratio $\frac{2l}{L}$ is, the more delta changes we can put into a PPS view range. When the ratio is between $1.5 \sim 2$, the value of n is at about a hundred. With such a small n, even though the querying cost is proportion to $O(n^2 l)$, we are not going to see a significant increase on the performance. However, when τ increases to a few hundred, we will start to see an impact on the querying cost. For instance, for $l = 10$, the cost would be about $O(10^6)$, which is at the same magnitude for the ratio between disk I/O and CPU processor time for arithmetic operations. Figure 23 confirms our observation by showing the query costs for a snapshot document provenance query and a delta-change document provenance query. The query cost is the amount of time to retrieve the requested position stamps from a PPS. From both figures, we can see that the query costs get increases as we increase the size of PPS view ranges due to the computation on delta changes. In addition, the increases become more significant when the size of PPS



(a) Snapshot document provenance queries (b) Delta-change document provenance queries

Figure 23: Query processing cost due to the size of PPS view ranges

view ranges get larger. Based on the case study on the Wikipedia article *Batman: The Animated Series*, we can obtain a rule of thumb regarding the setting for PPS view range size based on the following observation. We obtain the most significant disk space saving by setting the size of PPS view ranges to be 1.5 $\sim$ 2 for a wide range of ratio between $\frac{2l}{L}$. In addition, the query processing cost does not increase too much because with such a small increase in the size of PPS view ranges, we can only put at most a few hundreds of delta changes into a PPS view range, which is going to have an unnoticeable impact on query processing cost due to the dominance of disk I/O. Since most Wikipedia have the ratio between 0.005 $\sim$ 0.01, we choose $\tau = 2$ for the experiments in Section 6.5.

6.5 Experiments

6.5.1 Experiment Setup

Hardware configuration All experiments are conducted on a 32-bit GNU/Linux machine with Intel Pentium 4 CPU 2.80GHz and 8GB RAM.

Data set We use Wikipedia data dump timestamped on Jan 30, 2010 to evaluate the performance our system. The data dump contains around 19376810 articles and more than three hundreds millions of revisions. To provide an in-depth investigation

for the performance of our system, we choose three Wikipedia articles different in the length of their revision histories: *Elementary algebra*, *Devil May Cry (video game)*, and *IPod*. Some related statistics for them are shown in Table 3.

Table 3: Statistics of three Wikipedia articles

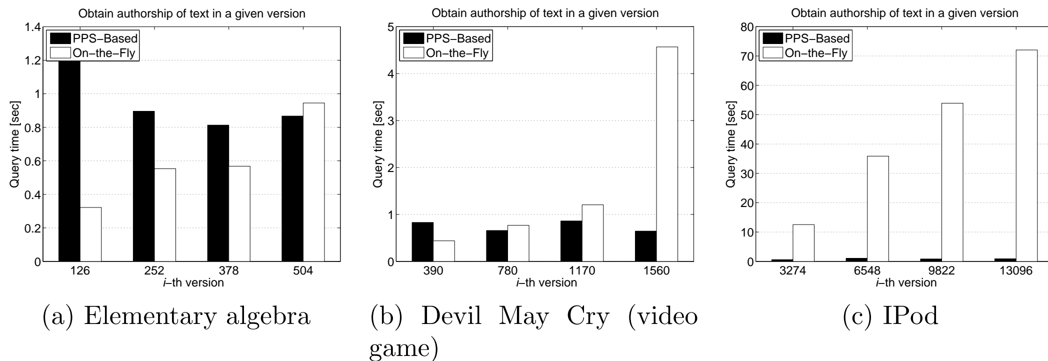
Article Name	versions [#]	Average items per version [#]	Average items in delta changes [#]
Elementary algebra	504	2882	36
Devil May Cry (video game)	1560	1395	27
IPod	13097	2801	3

The article titled *Elementary algebra* represents articles with a short revision history. The article titled *Devil May Cry (video game)* represents middle-sized articles, while the article *IPod* represents articles with long revision histories. For the experiments in the next section, we compare the querying cost for each of these articles to analyze the impact of length of revision histories on fine-grained document provenance queries.

6.5.2 Evaluating a Snapshot Document Provenance Query: Getting Authorship of Text at Selected Versions

We compare the querying time between the approaches PPS-Based and On-the-Fly on three Wikipedia articles to retrieve authorship of text at selected versions. Let us first look at the querying time for the On-the-Fly approach in Figure 24(a-c). In both articles, it is clear that the querying time increases monotonically as we process more recent versions. This is expected as the On-the-Fly approach has to do three tasks to retrieve the metadata: 1) read all the previous versions up to a given version; 2) reason the delta differences between all consecutive versions; and 3) associate the authorship metadata with different parts of the text. As the number of versions increase, so the querying time. By comparison, the querying cost for the PPS-based approach is not sensitive to the position of a version in the revision history. For example, the querying

time for the PPS-Delta approach (the black bars) in Figure 24(a) varies marginally at different versions. The same observation can be observed in Figure 24(b) as well, but less obvious due to the scale of Y-axis. The reason is that the major cost for the PPS-based approach is the time to load the PPS data structure and metadata into the memory. Once this step is done, there is no much difference to look up the metadata for different versions since the look-up costs for different versions are similar.



(a) Elementary algebra (b) Devil May Cry (video game) (c) IPod

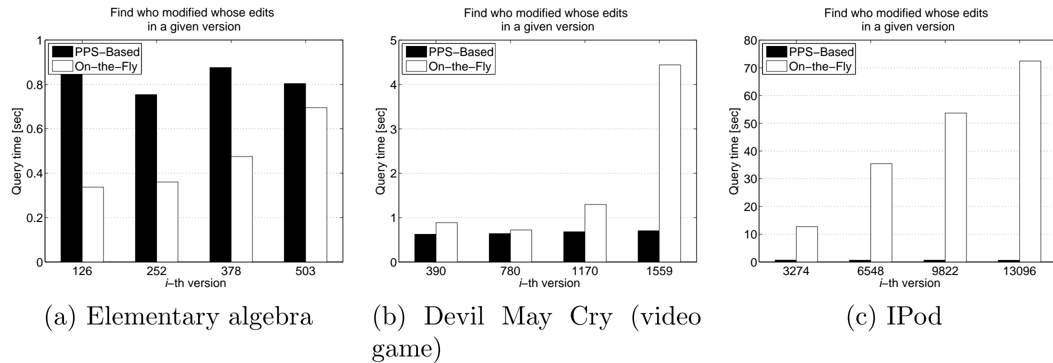
Figure 24: Compare querying cost for a snapshot document provenance query for three Wikipedia articles

The second observation is that the PPS-based approach outperforms the On-the-Fly approach for articles with more than thousands of versions. For articles with only a few hundreds of versions, the PPS-based approach performs slightly worse due to the overhead in de-serializing the PPS view ranges from disk and looking up provenance information in hash tables. The overhead becomes less significant as we progress articles with longer revision histories because the querying cost is dominated by disk I/O and the reasoning for the delta changes between the consecutive versions. In Figure 24(c), the difference between the PPS-based approach and the On-the-Fly approach are almost two orders of magnitude faster.

6.5.3 Evaluating a Delta-change Document Provenance Query: Getting Who Modified Whose Work at Selected Versions

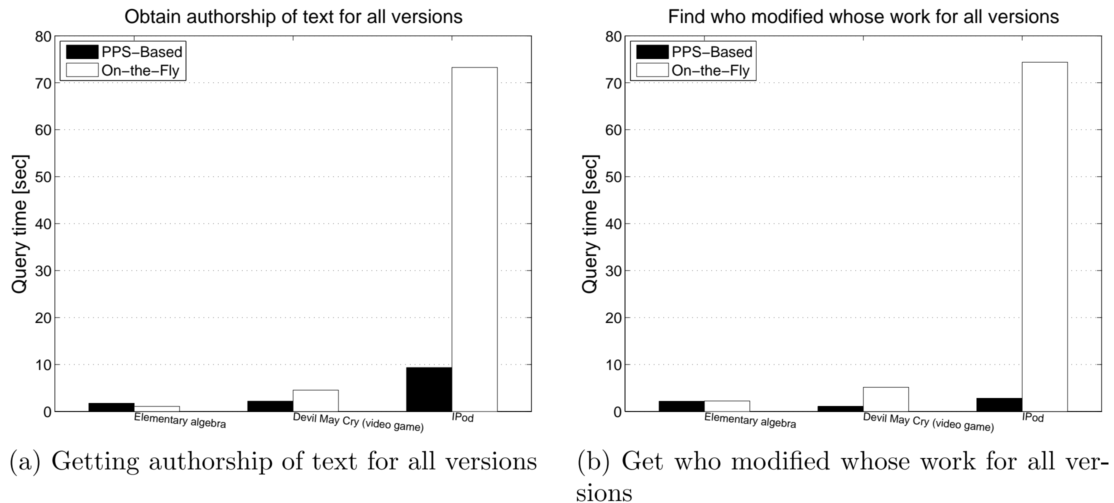
For delta-change document provenance queries, we evaluate the query "Who modified whose work at selected versions". We collect the querying time for the three Wikipedia

articles at different points in their revision histories. From Figure 25, we get the same observation as the snapshot document provenance query "Get authorship of text at selected versions". The most important observation we have from both type of queries is that the PPS-based approach is insensitive to the length of revision histories. We are able to query provenance information at any point with stable response time, which is a significant improvement over the current revision control systems.



(a) Elementary algebra (b) Devil May Cry (video game) (c) IPod

Figure 25: Compare querying cost for a delta-change document provenance query on three Wikipedia articles



(a) Getting authorship of text for all versions (b) Get who modified whose work for all versions

Figure 26: Compare querying cost on three Wikipedia articles

6.5.4 Evaluating Document Provenance Queries for Entire Revision History of Documents

For many data analytics applications, it is critical to obtain provenance information for the entire revision history of documents. we use two experiments to compare our PPS-based approach and the On-the-Fly approach. The first experiment evaluates the query "Getting Authorship of Text for All Versions", which retrieves authorship of text for all PPS views. For each PPS view, it looks up the author name and output the results. For the On-the-Fly approach, we read all the versions one by one. For each version, we use a diff utility to locate the changes and output the results. We see that for the article with a short revision history, the PPS-based performs slightly worse than the On-the-Fly approach, but outperforms On-the-Fly for articles with long revision histories. The performance gain becomes more significant as the length of revision histories gets longer.

The second experiment evaluates the query "Getting Who Modified Whose Work for All Versions". Figure 26(b) shows the amount of time to get the provenance information for who modified whose work for all the versions on the three Wikipedia articles. We obtain similar observation as the snapshot-range provenance query evaluated in the previous section except that delta-change-range query takes less time to process. This matches our expectation because it does not involve any computation on delta changes. As a result, the PPS-based approach performs as well as the On-the-Fly approach even on articles with short revision histories.

6.5.5 Disk Space Usage

To measure the disk space usage, we compare our PPS-based approach with the Flat-file approach, which stores all the versions of a document in a flat file one after another. To evaluate the impact of revision history length to both approaches, we put articles into different buckets. Each bucket holds articles whose revision length is within a certain range. For articles falling within the same bucket, we make an average

on the amount of disk space they use. The result is collected over 5000 Wikipedia articles, randomly sampled from the Wikipedia data dump. From Figure 27(a), we see that the PPS-based approach uses much less disk space compare to the Flat File approach. For articles with thousands of versions, the Flat-File approach uses almost two orders of magnitude more disk space than the PPS-based. This is an indication that Wikipedia articles have lots of overlapped content between consecutive versions. The decision of saving these overlapped content in every individual version to speed up version retrieval should really be revisited. Figure 27(b) shows the ratio between these two approaches. we can see that as the length of revision histories increase, the PPS-based approach save more than 95% of disk space.

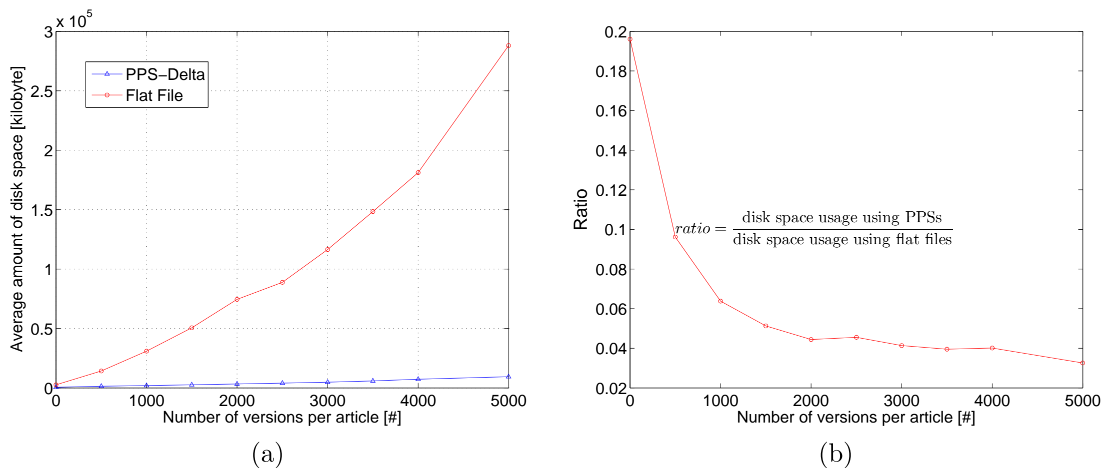


Figure 27: Disk space usage for Wikipedia articles with different number of versions: a) disk space usage; b) ratio

6.5.6 Document Loading Time

The time to represent a document in a PPS includes six parts: 1) the time to read all the versions from the disk; 2) the time to compute the delta changes between consecutive versions; 3) the time to represent these delta changes in PPS view ranges; 4) the time to associate provenance information with different position stamps; 5) the time to write these PPS view ranges to disk; and 6) the time to write provenance data to disk. From Figure 28, we see that for documents with short revision histories (less

than 1000), it takes a few seconds to load the document into its PPS representation. For documents with long revision histories, it takes tens of seconds to finish at about 10 millisecond per version. Therefore, it is practical to use PPSs for large scale document processing.

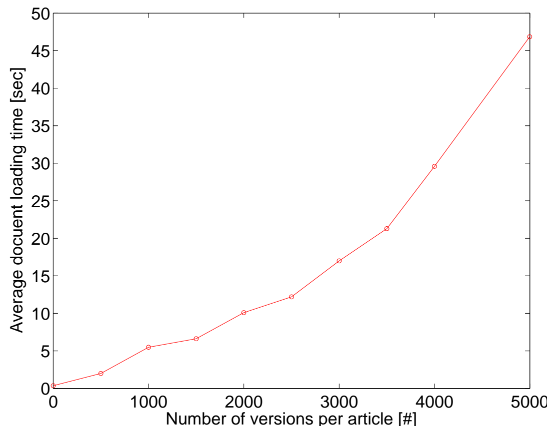


Figure 28: Document loading time for Wikipedia articles with different number of versions

.

6.6 Summary

Fine-grained document provenance describes how a portion of a document was modified over time, which enables us to obtain precise knowledge for documents at the content-level. Current revision control systems track provenance information at the document level. As a result, we have to parse revision histories and analyze delta changes between consecutive versions in order to collect provenance information at the content-level, which is inefficient especially for documents with long revision histories. We use PPSs to represent the revision history of documents and use position stamps to index provenance at fine granularities. Since PPSs represents revision histories by saving only a few full versions and delta changes for the rest, our approach is disk economic. Our experiments on processing Wikipedia articles show that our approach saves more than 90% of disk space for Wikipedia articles with thousands of versions on average. In addition, we are able to quickly reconstruct the state of

a document at any time as well as the delta changes between any two versions with stable response time. In addition, due to the uniqueness and persistent properties of PPSs, we can easily associate provenance data at arbitrary dimensions to each item and look up these data quickly. We compare the performance of our system with the On-the-Fly approach which parses revision histories at run time to process document provenance queries. The experiments show that for documents with short revision histories, our PPS-based approach performs slightly worse than the On-the-Fly approach due to the overhead in managing the data structure. But for documents with thousands of versions, the PPS-based approach outperforms the On-the-Fly approach and achieves at least an order of magnitude speedup on fine-grained metadata query processing.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 Dissertation Conclusion

We describe PPSs, a partial persistent data structure, to track changes and metadata of changes for collaborative document editing and processing applications. PPSs have two distinct features. First, they never remove any items from the data structure. By keep necessary timestamp information, we are able to access any previous version in the revision history of a document. In the second unique feature, PPSs create unique, ordered, and persistent identifiers for all the items. These identifiers are called position stamps, which can be used to index various metadata associated with each item. In order to balance the tradeoff between disk space and access cost to revision histories, we have developed a hybrid approach to represent revision history of collaborative documents. In the hybrid approach, all the document versions are represented by the position stamps of their items. The approach is hybrid because we only keep the full list of position stamps at a few points in the revision history. For versions between those points, only their delta changes are maintained. We further show that after representing the revision history of a collaborative document in our PPS data structure, we are able to access its revision history by executing classical set operators such as union, intersect, and minus. Based on this observation, we have designed efficient algorithms to access not only any previous version but also the delta changes between any two versions.

We demonstrate the capabilities of PPSs by applying them to the problem of data consistency control in collaborative editing systems. We approach this problem in two steps. In the first step, we have defined a relaxed consistency model more

suitable for real-time collaborative editing applications. By using PPSs to capture data dependencies between editing operations, a view synchronization strategy for the relaxed consistency model has been implemented and evaluated for its practicality. In the second step, we extend our scope to the entire spectrum of collaborative editing applications and design a transactional framework to specify different kinds of collaborations. Our framework is expressive as demonstrated by its capabilities to specify several representative collaborative editing systems such as CVS, MediaWiki, Google Docs. Within our transactional framework, we use PPSs to track dependencies between editing operations and quantify the amount of inconsistency in editing transactions. By varying transaction boundaries and inconsistency a transaction could import, we are able to design a general data consistency control algorithm for collaborative editing systems. We demonstrate the practicality of our framework by a prototype implementation over Berkey DB High Availability, a replicated database management engine.

We further demonstrate the capability of PPSs by using them to track and index document provenance information at fine granularities. Current revision control systems maintain document provenance at the document level, which is too coarse-grained to answer provenance queries at the content level. PPSs are a natural candidate data structure to solve this problem. By representing the revision history of documents in PPSs, we can efficiently access any previous version as well as the delta changes between any two versions in the revision history. In addition, the uniqueness and persistence properties of position stamps allow us to index various provenance data for a particular item. To process a fine-grained document provenance query, we first query the PPS structure to obtain relevant position stamps and then use these position stamps to look up requested provenance information. We have built a system and apply it to manage provenance information for millions of Wikipedia article. We compare the performance of our system with MediaWiki, the database engine for

Wikipedia. Our experiments show that our system uses less than 10% of disk space compared to MediaWiki. In addition, we show that PPSs are most useful to handle collaborative documents with a long revision history. For documents with thousands of versions, PPSs are capable of achieving stable response time for both snapshot and delta-change document provenance queries and at least an order of magnitude speedup for documents with tens and thousands of versions.

7.2 Future Work

7.2.1 Partial Persistent Sequences

PPSs use a labeling scheme to assign position stamps for new items. The choice of labeling schemes has a significant impact on the rebalancing frequency of PPSs. Since rebalancing is an expensive operation, we want a labeling scheme to be conscious about on-going editing activities such that it will reserve more space for areas that are under intensive modification and less space for areas that have relatively stable content. Currently, we use the dyadic labeling scheme that halves the space between two rational numbers. A better approach is to collect statistics for editing intensity on different areas of a document. During the rebalancing stage, these statistics are used to determine the ratio of space for different areas in the document. However, we need to be careful for the over fitting problem. It is possible that an inactive area suddenly becomes active or vice versa. As a result, we end up with inadequate space for to-be-active areas and wasted space for areas not active any more, which could quickly trigger another round of rebalancing. We will consider knowledge of past editing activities and uncertainty of future activities for the design of new labeling schemes.

Some applications require fully persistent form of sequences. In software configuration systems, a source code is branched or forked because it may be necessary to develop two versions of the file concurrently. PPSs are partially persistent form of

sequences. To extend PPSs to its fully persistent form, we need to maintain a tree structure to handle branching versions. The tree has a main trunk, which contains the major revision history of a document. A branch in the tree represents the revision history of a new copy of the document from the branching point forward. Maintaining the tree structure does not introduce new technical challenges because we could use a PPS to maintain the revision history for each copy of a document. However, a challenge arises when we want to merge a branch into the trunk. Using a PPS for each branch means that items in a branch will be assigned position stamps independent of the items on the main truck. As a result, during the merge, we could have items that have the same position stamps, but actually represent different objects. In this case, we have to use the position stamps in the first version of the branch to determine the right merging points. However, this only solves half of the problem because the PPS for the trunk may get re-balanced and reassigned its position stamps completely. To handle this problem, we need to maintain a mapping that records the before-rebalancing and after-rebalancing position stamps of items at each rebalancing stage, which could introduce lots of maintenance work and computation overhead for the merge operation. We will use the above idea as a starting point for designing fully persistent form of sequences and explore different techniques to address the efficiency of the merge operation.

7.2.2 Data Consistency Control in Collaborative Text Document Editing Systems

Our work on data consistency control opens the door of using transactional techniques for collaborative text document editing systems. To push this direction forward, we look forward to delve into the following two topics. The first topic is to continue our work on designing new type of collaborations that are more suitable for fine-grained shared access on large documents. At the current stage, collaborative editing systems either postpone the detection of conflicts at the time of merging different copies of

documents or leave the detection of conflicts to end users. The former could result in divergence of different copies too big to be resolved easily, while the latter requires that users be aware of on-going activities at all time, which is unrealistic due to the limited focus zone of users. A more suitable type of collaboration is to allow users to establish dependencies between different parts of documents or set the scope of exclusive editing at fine granularities. In this way, users would receive alarms when dependent data get modified or have exclusive write access to a portion of document without being worried about being blind to simultaneous edits. Our work on using PPSs to track data dependencies and transactions to detect editing conflicts are concrete steps towards this direction. The second topic is mobile collaborative editing systems. Mobile applications face the conditions of network intermittence by default. Our approach can use recovery techniques from the database technologies to design robust solutions for this issue. On the other hand, transactions techniques may be too heavy weight for mobile applications where computation resources are limited. How to balance the tradeoff between performance and robustness of system for mobile applications will be one of our future directions.

7.2.3 Fine-grained Document Provenance Queries

Document provenance management will become increasingly important as collaborations becomes more open-ended and large-scale. This dissertation is the first step towards the goal of providing a general solution for indexing, storing, and querying document provenance at fine granularities. To achieve this goal, the following two problems need to be addressed. First, we need a high-level provenance query language for unstructured text documents. A desirable query language should be in many ways similar to SQL that provides a suitable level of abstraction to allow users to query provenance data without the knowledge of underlying data structures. As

the amount of provenance information increases, we need suitable programming interfaces for users to manage various dimension of provenance and their relationships. Besides suitable abstraction, the query language should take into consideration the spatial dimension corresponding to the sequential structure of text documents. Along with the temporal dimension which has been considered in this dissertation, the spatial dimension should be considered as another basic dimension in the query language such that users are able to zoom into provenance at selected areas. Second, we need an efficient approach to manage large amount of document provenance data. Based on our experiences with Wikipedia articles, document provenance can create a large number of records as we manage provenance at fine granularities. A single document can easily create tens of thousands of entries. Managing all these entries for millions of documents can impose non-trivial performance challenges for any relational or key/value based databases. More practical solutions are needed to manage multi-dimensional provenance data efficiently.

Besides addressing the above two problems, we have started the work on deep document knowledge discovery based on document provenance information. We have made concrete progress towards this direction by designing new techniques for vandalism detection in Wikipedia [126]. In the future, we are going to study editing behaviors of users based on their editing histories. We foresee that our progress on this topic will provide valuable insights into several applications including design of incentive strategies to improve sustainability of wikis and the quality of information based on degrees of persistence in users' contributions.

VITA

Qinyi Wu was born and brought up in Guiyang, the capital city of Guizhou. She received her bachelor's degree in Computer Science with outstanding honor from Wuhan University, Hubei, China in 1999. She continued her master's degree in Computer Science from Beijing Institute of Technology, Hebei, China in 2002. After that, she moved to Atlanta to pursue a Ph.D. in Computer Science at the College of Computing at Georgia Institute of Technology. Qinyi pursued her dissertation research in the area of data management in collaborative text document editing and processing systems under the guidance of Prof. Calton Pu in the Distributed Data Intensive Systems Lab. She contributed to a multitude of projects including data consistency control in realtime collaborative document editing systems, fine-grained metadata management for text documents, vandalism detection in wikis, and dependency management in workflows.

REFERENCES

[1] “Activemq: A JMS server.” http://activemq.apache.org/jms-to-jms-bridge.html.

[2] “Gobby: a collaborative text editor.” http://gobby.0x539.de/trac/.

[3] “Google code: google-diff-match-patch.” http://code.google.com/p/google-diff-match-patch/.

[4] “Package javax.swing.” http://java.sun.com/javase/6/docs/api/javax/swing/package-summary.html.

[5] “Wikipedia data dump download.” http://download.wikimedia.org/enwiki/.

[6] “7 things you should know about collaborative editing.” http://connect.educause.edu/Library/ELI/7ThingsYouShouldKnowAbout/39386?time=1202527547, 2005. Educause Connect.

[7] “Lorem Ipsum.” http://www.lipsum.com/, 2009.

[8] “SubEthaEdit.” http://www.codingmonkeys.de/subethaedit/, 2009.

[9] “Alexa traffic rankings.” http://www.alexa.com/siteinfo/wikipedia.org+sina.com.cn+orkut.com+live.com+youtube.com, 2010.

[10] “Bpm and social software.” http://www.bpm2010.org/conference-program/workshops/bpms210/, 2010.

[11] “Comparing and merging files.” http://www.gnu.org/software/diffutils/manual/html_mono/diff.html, 2010.

[12] “Cvs - concurrent versions systems.” http://www.nongnu.org/cvs/, 2010.

[13] “Data structures for text sequences.” http://www.cs.unm.edu/~crowley/papers/sds/sds.html, 2010. Charles Crowley.

[14] “Differential synchronization overview.” http://neil.fraser.name/writing/sync/, 2010.

[15] “Google docs basics.” http://docs.google.com/support/bin/static.py?hl=en&page=guide.cs&guide=20322, 2010.

[16] “Googlewave white paper.” http://www.waveprotocol.org/whitepapers/operational-transform, 2010.

[17] "Mediawiki 1.15.1." http://www.mediawiki.org/wiki/MediaWiki, 2010.

[18] "Oracle berkeley db java edition high availability." http://www.oracle.com/technology/products/berkeley-db/pdf/berkeleydb-je-ha-whitepaper.pdf, 2010.

[19] "Twiki system requirements." http://twiki.org/cgi-bin/view/TWiki/TWikiSystemRequirements, 2010.

[20] "vandalism." http://en.wikipedia.org/wiki/Wikipedia:Vandalism, 2010.

[21] "Wikipedia: a free, web-based, collaborative, multilingual encyclopedia." http://en.wikipedia.org/wiki/Wikipedia, 2010.

[22] "Wikitrust." http://wikitrust.soe.ucsc.edu/, 2010.

[23] "Postgresql 7.2 user's guide." www.postgresql.org/files/documentation/pdf/7.2/user-7.2-US.pdf, 2011.

[24] "Fsfs: Subversion filesystem implementation note." http://svn.apache.org/repos/asf/subversion/trunk/notes/fsfs, March 25, 2011.

[25] "Wd caviar green / gp product specifications." http://support.wdc.com/product/install.asp?wdc_lang=en&fid=wdsfCaviar_Green, May, 2011. Educause Connect.

[26] Abowd, G. D. and Dix, A. J., "Giving undo attention," *Interact. Comput.*, vol. 4, no. 3, pp. 317–342, 1992.

[27] Adler, B. T. and de Alfaro, L., "A content-driven reputation system for the wikipedia," in *WWW '07: Proceedings of the 16th international conference on World Wide Web*, (New York, NY, USA), pp. 261–270, ACM, 2007.

[28] Adya, A., Liskov, B., and O'Neil, P., "Generalized isolation level definitions," pp. 67 –78, 2000.

[29] Alevizou, P. and Forte, A., "Engaging with open education," in *WikiSym '10: Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, (New York, NY, USA), pp. 1–2, ACM, 2010.

[30] Amagasa, T., Yoshikawa, M., and Uemura, S., "Qrs: A robust numbering scheme for xml documents," *Data Engineering, International Conference on*, vol. 0, p. 705, 2003.

[31] Bancilhon, F., Kim, W., and Korth, H. F., "A model of cad transactions," in *VLDB '1985: Proceedings of the 11th international conference on Very Large Data Bases*, pp. 25–33, VLDB Endowment, 1985.

[32] Barga, R. S. and Pu, C., "A practical and modular implementation of extended transaction models," in *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 206–217, Morgan Kaufmann Publishers Inc., 1995.

[33] Bentley, J. L. and Saxe, J. B., "Decomposable searching problems i. static-to-dynamic transformation," *Journal of Algorithms*, vol. 1, no. 4, pp. 301 – 358, 1980.

[34] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P., "A critique of ansi sql isolation levels," in *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 1–10, ACM, 1995.

[35] Bose, R. and Frew, J., "Lineage retrieval for scientific data processing: a survey," *ACM Comput. Surv.*, vol. 37, pp. 1–28, March 2005.

[36] Chazelle, B. and Guibas, L., "Fractional cascading: A data structuring technique with geometric applications," in *Automata, Languages and Programming* (Brauer, W., ed.), vol. 194 of *Lecture Notes in Computer Science*, pp. 90–100, Springer Berlin / Heidelberg, 1985. 10.1007/BFb0015734.

[37] Chen, D. and Sun, C., "Undoing any operation in collaborative graphics editing systems," in *GROUP '01: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, (New York, NY, USA), pp. 197–206, ACM, 2001.

[38] Chrysanthis, P. K. and Ramamritham, K., "Acta: a framework for specifying and reasoning about transaction structure and behavior," *SIGMOD Rec.*, vol. 19, no. 2, pp. 194–203, 1990.

[39] Clifton, C. and Garcie-Molina, H., "The design of a document database," in *DOCPROCS '88: Proceedings of the ACM conference on Document processing systems*, (New York, NY, USA), pp. 125–134, ACM, 1988.

[40] Cohen, E., Kaplan, H., and Milo, T., "Labeling dynamic xml trees," in *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 271–281, ACM, 2002.

[41] Cohen, W. W. and Singer, Y., "Context-sensitive learning methods for text categorization," in *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, (New York, NY, USA), pp. 307–315, ACM, 1996.

[42] Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M., *Version Control with Subversion*. San Francisco, CA, USA: O'Reilly Media, 2004.

[43] Conradi, R. and Westfechtel, B., "Version models for software configuration management," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 232–282, 1998.

[44] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R., "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[45] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E., *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd ed., 2001.

[46] Dale, N. and Walker, H. M., *Abstract data types: specifications, implementations, and applications.* Lexington, MA, USA: D. C. Heath and Company, 1996.

[47] Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P., "A fine-grained access control system for xml documents," *ACM Trans. Inf. Syst. Secur.*, vol. 5, no. 2, pp. 169–202, 2002.

[48] Danis, C. and Singer, D., "A wiki instance in the enterprise: opportunities, concerns and reality," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, CSCW '08, (New York, NY, USA), pp. 495–504, ACM, 2008.

[49] De la Calzada, G. and Dekhtyar, A., "On measuring the quality of wikipedia articles," in *Proceedings of the 4th workshop on Information credibility*, WICOW '10, (New York, NY, USA), pp. 11–18, ACM, 2010.

[50] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[51] Demaine, E. D., Langerman, S., and Price, E., "Confluently persistent tries for efficient version control," in *SWAT '08: Proceedings of the 11th Scandinavian workshop on Algorithm Theory*, (Berlin, Heidelberg), pp. 160–172, Springer-Verlag, 2008.

[52] Demartini, G., "Finding experts using wikipedia," *FEWS 2007: Finding Experts on the Web with Semantics Workshop at ISWC 2007 + ASWC 2007*, November 2007.

[53] Dourish, P., "Consistency guarantees: exploiting application semantics for consistency management in a collaboration toolkit," in *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, (New York, NY, USA), pp. 268–277, ACM, 1996.

[54] Dourish, P. and Bellotti, V., "Awareness and coordination in shared workspaces," in *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, (New York, NY, USA), pp. 107–114, ACM, 1992.

[55] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E., "Making data structures persistent," in *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 109–121, ACM, 1986.

[56] Du, F., Amer-Yahia, S., and Freire, J., "Shrex: managing xml documents in relational databases," in *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pp. 1297–1300, VLDB Endowment, 2004.

[57] Ellis, C. A. and Gibbs, S. J., "Concurrency control in groupware systems," *SIGMOD Rec.*, vol. 18, no. 2, pp. 399–407, 1989.

[58] Ellis, C. A., Gibbs, S. J., and Rein, G., "Groupware: some issues and experiences," *Commun. ACM*, vol. 34, no. 1, pp. 39–58, 1991.

[59] Elmagarmid, A. K., ed., *Database transaction models for advanced applications.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[60] Elmasri, R., Wuu, G. T. J., and Kouramajian, V., "The time index and the monotonic b+-tree," 1993.

[61] Flor, N. V., "Globally distributed software development and pair programming," *Commun. ACM*, vol. 49, no. 10, pp. 57–58, 2006.

[62] Fong, P. K.-F. and Biuk-Aghai, R. P., "What did they do? deriving high-level edit histories in wikis," in *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, WikiSym '10, (New York, NY, USA), pp. 2:1–2:10, ACM, 2010.

[63] Garcia-Molina, H. and Salem, K., "Sagas," in *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 249–259, ACM, 1987.

[64] Gray, J., Helland, P., O'Neil, P., and Shasha, D., "The dangers of replication and a solution," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 173–182, ACM, 1996.

[65] Gropengieβer, F., Hose, K., and Sattler, K.-U., "An extended transaction model for cooperative authoring of xml data," *Computer Science - Research and Development*, vol. 24, no. 1, pp. 85–100, 2009.

[66] Grudin, J. and Poole, E. S., “Wikis at work: success factors and challenges for sustainability of enterprise wikis,” in *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, WikiSym ’10, (New York, NY, USA), pp. 5:1–5:8, ACM, 2010.

[67] Halfaker, A., Kittur, A., Kraut, R., and Riedl, J., “A jury of your peers: quality, experience and ownership in wikipedia,” in *WikiSym ’09: Proceedings of the 5th International Symposium on Wikis and Open Collaboration*, (New York, NY, USA), pp. 1–10, ACM, 2009.

[68] Heineman, G. T., “A transaction manager component for cooperative transaction models,” in *CASCON ’93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pp. 910–918, IBM Press, 1993.

[69] Hodel-Widmer, T. B. and Dittrich, K. R., “Concept and prototype of a collaborative business process environment for document processing,” *Data Knowl. Eng.*, vol. 52, pp. 61–120, January 2005.

[70] Holtzblatt, L. J., Damianos, L. E., and Weiss, D., “Factors impeding wiki use in the enterprise: a case study,” in *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems*, CHI EA ’10, (New York, NY, USA), pp. 4661–4676, ACM, 2010.

[71] Ikeda, R. and Widom, J., “Data lineage: A survey,” technical report, Stanford University, 2009.

[72] Jensen, C. S., Clifford, J., Gadia, S. K., Segev, A., and Snodgrass, R. T., “A glossary of temporal database concepts,” vol. 21, (New York, NY, USA), pp. 35–43, ACM, September 1992.

[73] Jensen, C. S., Mark, L., and Roussopoulos, N., “Incremental implementation model for relational databases with transaction time,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 3, pp. 461–473, December 1991.

[74] Jensen, C. S., “Introduction to temporal database research.” `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.4534&rep=rep1&type=pdf`, 2000.

[75] Jensen, C. S., Mark, L., Roussopoulos, N., and Sellis, T., “Using differential techniques to efficiently support transaction time,” *The VLDB Journal*, vol. 2, pp. 75–116, January 1993.

[76] Jensen, C. S., Mark, L., Roussopoulos, N., and Sellis, T., “Using caching, cache indexing and differential techniques to efficiently support transaction time,” tech. rep., College Park, MD, USA, 1999.

[77] Karat, C.-M., Halverson, C., Horn, D., and Karat, J., “Patterns of entry and correction in large vocabulary continuous speech recognition systems,”

in *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 568–575, ACM, 1999.

[78] Kindler, E., Rubin, V., and Schäfer, W., "Incremental workflow mining for process flexibility," in *Proceedings of the CAISE'06 Workshop on Business Process Modelling, Development, and Support (BPMDS '06)*, (Luxemburg), June 5-9 2006.

[79] Kittur, A., Chi, E. H., Pendleton, B. A., Suh, B., and Mytkowicz, T., "Power of the few vs. wisdom of the crowd: Wikipedia and the rise of the bourgeoisie," 2007.

[80] Krikellas, K., Elnikety, S., Vagena, Z., and Hodson, O., "Strongly consistent replication for a bargain," in *ICDE*, 2010.

[81] Lagogiannis, G., Panagis, Y., Sioutas, S., and Tsakalidis, A., "A survey of persistent data structures," in *ICCOMP'05: Proceedings of the 9th WSEAS International Conference on Computers*, (Stevens Point, Wisconsin, USA), pp. 1–6, World Scientific and Engineering Academy and Society (WSEAS), 2005.

[82] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[83] Laurent, A. M. S., *Understanding Open Source and Free Software Licensing.* O'Reilly Media, Inc., 2004.

[84] Leone, S., Hodel-Widmer, T. B., Böhlen, M. H., and Dittrich, K. R., "Tendax, a collaborative database-based real-time editor system," in *EDBT*, pp. 1135–1138, 2006.

[85] Li, D. and Li, R., "Preserving operation effects relation in group editors," in *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, (New York, NY, USA), pp. 457–466, ACM, 2004.

[86] Li, D. and Li, R., "An operational transformation algorithm and performance evaluation," *Comput. Supported Coop. Work*, vol. 17, no. 5-6, pp. 469–508, 2008.

[87] Li, R. and Li, D., "A new operational transformation framework for real-time group editors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 3, pp. 307–319, 2007.

[88] Mancini, L. V., Ray, I., Jajodia, S., and Bertino, E., "Flexible transaction dependencies in database systems," *Distrib. Parallel Databases*, vol. 8, no. 4, pp. 399–446, 2000.

[89] Mehrotra, S., Rastogi, R., Silberschatz, A., and Korth, H., "A transaction model for multidatabase systems," pp. 56 –63, jun 1992.

[90] Moss, J. E. B., "Nested transactions: An approach to reliable distributed computing," in *International Conference on Management of Data*, 1981.

[91] Myers, E. W., "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[92] Newman-Wolfe, R. E., Webb, M. L., and Montes, M., "Implicit locking in the ensemble concurrent object-oriented graphics editor," in *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, (New York, NY, USA), pp. 265–272, ACM, 1992.

[93] Nichols, D. A., Curtis, P., Dixon, M., and Lamping, J., "High-latency, low-bandwidth windowing in the jupiter collaboration system," in *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, (New York, NY, USA), pp. 111–120, ACM, 1995.

[94] Noël, S. and Robert, J.-M., "Empirical study on collaborative writing: What do co-authors do, use, and like?," *Comput. Supported Coop. Work*, vol. 13, no. 1, pp. 63–89, 2004.

[95] Olson, M. A., Bostic, K., and Seltzer, M., "Berkeley db," in *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 43–43, USENIX Association, 1999.

[96] Ortega, F. and Gonzalez Barahona, J. M., "Quantitative analysis of thewikipedia community of users," in *Proceedings of the 2007 international symposium on Wikis*, WikiSym '07, (New York, NY, USA), pp. 75–86, ACM, 2007.

[97] Oster, G., Urso, P., Molli, P., and Imine, A., "Data consistency for p2p collaborative editing," in *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, (New York, NY, USA), pp. 259–268, ACM, 2006.

[98] Oster, G., Molli, P., Urso, P., and Imine, A., "Tombstone transformation functions for ensuring consistency in collaborative editing systems," pp. 1 –10, nov. 2006.

[99] Oster, G., Urso, P., Molli, P., and Imine, A., "Data Consistency for P2P Collaborative Editing," in *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, (Banff, Alberta, Canada), pp. 259–267, ACM Press, nov 2006.

[100] Preguica, N., Marques, J. M., Shapiro, M., and Letia, M., "A commutative replicated data type for cooperative editing," *Distributed Computing Systems, International Conference on*, vol. 0, pp. 395–403, 2009.

[101] Priedhorsky, R., Chen, J., Lam, S. T. K., Panciera, K., Terveen, L., and Riedl, J., "Creating, destroying, and restoring value in wikipedia," in *GROUP '07: Proceedings of the 2007 international ACM conference on Supporting group work*, (New York, NY, USA), pp. 259–268, ACM, 2007.

[102] Pu, C., Kaiser, G. E., and Hutchinson, N. C., "Split-transactions for open-ended activities," in *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 26–37, Morgan Kaufmann Publishers Inc., 1988.

[103] Ressel, M., Nitsche-Ruhland, D., and Gunzenhäuser, R., "An integrating, transformation-oriented approach to concurrency control and undo in group editors," in *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, (New York, NY, USA), pp. 288–297, ACM, 1996.

[104] Roddick, J. F. and Patrick, J. D., "Temporal semantics in information systems: a survey," *Inf. Syst.*, vol. 17, pp. 249–267, May 1992.

[105] Saad, Y., *SPARSKIT: a basic tool kit for sparse matrix computations - Version 2.* 1994.

[106] Schwartz, A., "Who writes wikipedia?." `http://www.aaronsw.com/weblog/whowriteswikipedia`, 2006.

[107] Segev, A. and Gunadhi, H., "Efficient indexing methods for temporal relations," *IEEE Trans. on Knowl. and Data Eng.*, vol. 5, pp. 496–509, June 1993.

[108] Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., and Naughton, J. F., "Relational databases for querying xml documents: Limitations and opportunities," in *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 302–314, Morgan Kaufmann Publishers Inc., 1999.

[109] Silberstein, A., He, H., Yi, K., and Yang, J., "Boxes: Efficient maintenance of order-based labeling for dynamic xml data," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, (Washington, DC, USA), pp. 285–296, IEEE Computer Society, 2005.

[110] Simmhan, Y. L., Plale, B., and Gannon, D., "A survey of data provenance in e-science," *SIGMOD Rec.*, vol. 34, pp. 31–36, September 2005.

[111] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P., "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pp. 1150–1160, VLDB Endowment, 2007.

[112] Stonebraker, M., Stettner, H., Lynn, N., Kalash, J., and Guttman, A., "Document processing in a relational database system," *ACM Trans. Inf. Syst.*, vol. 1, no. 2, pp. 143–158, 1983.

[113] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D., "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, 1998.

[114] Sun, D. and Sun, C., "Context-based operational transformation in distributed collaborative editing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 1454–1470, 2009.

[115] Tammaro, S. G., Mosier, J. N., Goodwin, N. C., and Spitz, G., "Collaborative writing is hard to support: A field study of collaborative writing," *Comput. Supported Coop. Work*, vol. 6, no. 1, pp. 19–51, 1997.

[116] Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E., and Zhang, C., "Storing and querying ordered xml using a relational database system," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 204–215, ACM, 2002.

[117] Tichy, W. F., "Rcs—a system for version control," *Softw. Pract. Exper.*, vol. 15, no. 7, pp. 637–654, 1985.

[118] Tolone, W., Ahn, G.-J., Pai, T., and Hong, S.-P., "Access control in collaborative systems," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 29–41, 2005.

[119] Tropashko, V., "Nested intervals tree encoding in sql," *SIGMOD Rec.*, vol. 34, no. 2, pp. 47–52, 2005.

[120] van der Aalst, W. M. P., van Dongen, B. F., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. J. M. M., "Workflow mining: a survey of issues and approaches," *Data Knowlwdge Engineering*, vol. 47, pp. 237–267, November 2003.

[121] Viégas, F. B., Wattenberg, M., and Dave, K., "Studying cooperation and conflict between authors with history flow visualizations," in *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 575–582, ACM, 2004.

[122] Voss, J., "Measuring wikipedia," in *Proceedings of the International Conference of the International Society for Scientometrics and Informetrics (ISSI)*, no. 10, (Stockholm), 2005.

[123] Weikum, G. and Vossen, G., *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[124] Weiss, S., Urso, P., and Molli, P., "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks," in *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, (Washington, DC, USA), pp. 404–412, IEEE Computer Society, 2009.

[125] West, A. G., Kannan, S., and Lee, I., "Detecting wikipedia vandalism via spatio-temporal analysis of revision metadata?," in *Proceedings of the Third European Workshop on System Security*, EUROSEC '10, (New York, NY, USA), pp. 22–28, ACM, 2010.

[126] Wu, Q., Irani, D., Pu, C., and Ramaswamy, L., "Elusive vandalism detection at wikipedia: A text stability-based approach," in *Proceeding of the 19th ACM conference on Information and knowledge management (CIKM '10)*, 2010.

[127] Wu, Q. and Pu, C., "Modeling and implementing collaborative text editing systems with transactional techniques," *International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'10)*, 2010.

[128] Wu, Q., Pu, C., and Ferreira, J. E., "A partial persistent data structure to support consistent shared access in collaborative editing applications," in *26th IEEE International Conference on Data Engineering (ICDE'10)*, 2010.

[129] Wu, Q., Pu, C., and Irani, D., "Cosmos: a wiki data management system," in *Proceedings of the 5th International Symposium on Wikis and Open Collaboration (WikiSym '09)*, (New York, NY, USA), pp. 1–2, ACM, 2009.

[130] Wu, Q., Pu, C., Sahai, A., Barga, R., and Jung, G., "Dscweaver: Synchronization-constraint aspect extension to procedural process specification languages," in *Journal of Web Service Research*, pp. 320–330, 2008.

[131] Wu, Q., Pu, C., Sahai, A., and Barga, R. S., "Categorization and optimization of synchronization dependencies in business processes," in *Proceedings of the 23rd International Conference on Data Engineering (ICDE'07)*, pp. 306–315, 2007.

[132] Wu, Q. and Sahai, A., "Dag synchronization constraint language for business processes," in *Proceedings of the The 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC'06)*, (Washington, DC, USA), IEEE Computer Society, 2006.

[133] Xia, S., Sun, D., Sun, C., Chen, D., and Shen, H., "Leveraging single-user applications for multi-user collaboration: the coword approach," in *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pp. 162–171, 2004.

[134] Xu, L., Ling, T. W., Wu, H., and Bao, Z., "Dde: from dewey to a fully dynamic xml labeling scheme," in *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 719–730, ACM, 2009.

[135] Zobel, J. and Moffat, A., "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, p. 6, 2006.