IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4887

# A Collaborative, Real-Time Insert Transaction for a Native Text Database System

Thomas B. Hodel-Widmer
University of Zurich, Department of Information Technology, Winterthurerstr. 190, CH-8057 Zurich, Switzerland
hodel@ifi.unizh.ch

Klaus R. Dittrich
University of Zurich, Department of Information Technology, Winterthurerstr. 190, CH-8057 Zurich, Switzerland
dittrich@ifi.unizh.ch

## ABSTRACT

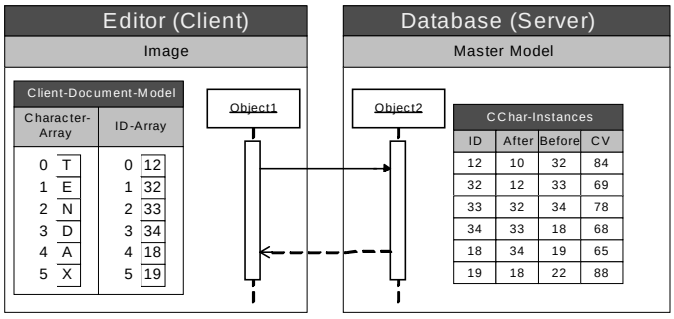
*For a large-scale document management environment, we often make local copies of remote data sources. However, it is often difficult to monitor the sources in order to check for changes and to download changed data items to the copies. In this paper, a insert transaction for a collaborative editor that stores its data natively in a database in order to make all changes immediately available to all users, will be presented. One of the most significant challenges we meet in building real-time cooperative editing systems, is distributed concurrency control. Therefore we focus on a concurrency control schema for a database based collaborative editor.*

## INTRODUCTION

Today, business (customer, product, finance, etc.) and text data (documents) are treated rather differently in computerized systems. Very often, word processing documents are stored somewhere within a confusing file structure with an inscrutable hierarchy and low security. On the other hand, operational and decision supporting data are stored in databases. The infrastructure and the data are highly secure, tailored towards multi-user operation and available to several other tools to build reports, content and knowledge.

Our idea is to use a similar philosophy for texts. We constructed a word processing concept with collaboration functionalities such as simultaneous writing in a shared document, automatic versioning, a document centered undo function, an access control concept based on users and roles and a copy-and-paste function with memory features for automatic updating. Therefore, we are striving for the storage of texts in a database in a native way that enables the above mentioned functionalities. By native, we mean that text is stored in a structured way in the database, so that database transactions can be applied to it.

*Figure 1: Data representation in the editor and database*




Shortcomings of document processing, state of the art in document processing and advantages for our database approach as well as a concept, prototype and performance evaluation of the mentioned transaction in this article are described in [Hodel 2003].

## NATIVE TEXT DATABASE

Historically, text has been perceived as requiring a different set of technologies for retrieval and management than structured data. This perception has not only burdened organizations with multiple storage systems and development environments but has also stood in the way of effectively integrating all organization information assets into a database.

We are convinced that word processing applications should store data in a 'native' way in a database and then benefit from the advantages of a database management system like querying the content, restricting access, persistent storage, inference and rule-based actions, multiple user interfaces, representing complex relationships among data, integrity constraints, backup and recovery, and much more.

Based on the ideas mentioned above and resulting constraints we developed a character based storage system. Such a data schema has big advantages in fulfilling these requests. The main question was how to represent the order of the characters.

The best model, based on our demands, is to implement a text document as a '*ring of double linked fields*' (see Figure 1). Based on these reflections, a 'native text database schema' describes a text as a set of double linked fields. Each character is stored as Unicode and identified by the object ID that is indexed by the database. Each object contains a reference to the next character, the Before property and a reference to the previous character, the After property.

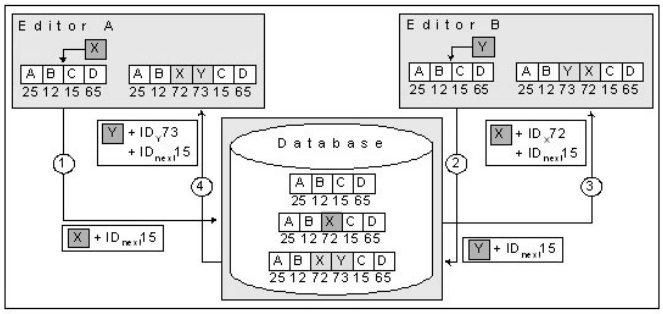
## CONSISTENCY PROBLEMS

In this section we will examine text transactions for our native text database if more than one user is editing a document at the same position and will be explained through four problem situations, based on the possible combinations of insert and delete operations. The problem situations must be examined one by one since they can cause hidden problems.

### Problem 1: Inserting a Character Simultaneously at the Same Position

Editor A (see Figure 2) inserts character X at the third position. The operation of editor A is received and processed correspondingly by the database. Right after that, the update process is activated. Before editor B gets informed about this insert, editor B has inserted character Y at the same position. This operation is also processed correctly and

*Figure 2: Insert at same position*




leads to an update process. Within the context of such a scenario, consistency problems arise. The models of editor A and the database correspond to the order of the operations carried out. The model in editor B, however, is not consistent to all other models. The problem is caused because the order of the operations within editor B is distorted. Character Y is inserted after character X in the database. Within editor B, character Y is inserted before character X. Editor B processes the operations in another order than the database system. This is possible because this approach does not guarantee the total causal order of the operations.

**Problem 2: Deleting the Same Character Simultaneously**

Editor A (compare Figure 2) deletes character C by transferring the ID of this character and informs the database system about this operation. After the character was deleted successfully, the update process is activated. Before editor B could be updated, editor B has deleted the same character C. Thus, the database system no longer knows ID 15 of the character any more. The delete operation of editor B therefore leads to a database error. In the meantime the update process has propagated the operation of editor A, and deleted character C with ID 15 in all other editors, including the editor B. The problem is being solved by itself.

**Problem 3: Simultaneously Inserting and Deleting at the Same Position**

Editor A (compare Figure 2) inserts character X, and the database system gets informed about this operation. Character X was saved in the database, and the update process informs the other editors about this operation immediately.

In the meantime, editor B has deleted the reference character C. The update process for inserting X uses character C with the ID 15 as a reference. Since editor B has just deleted character C, the update report of editor B leads to problems. Editor B does not know at which position the new character has to be inserted. These problems arise as well from the exchange of the operation order. Unlike problem 1 and 2, this situation leads to an error in the editor.

**Problem 4: Simultaneously Deleting and Inserting at the Same Position**

This scenario is nearly identical to the previous situation with the difference being that a character is deleted first and then a character is inserted. But this difference leads to a completely different problem. Character X (compare Figure 2) with the reference character C is no longer in the database. This has the consequence that the database will send B a related error message back to the editor since the operation fails.

**Summary of Consistency Problems**

Problems 2 and 4 are relatively simple to solve since an error message is automatically generated by the database system. Problems 1 and 3 can be explained by the missing total casual order of the operations. Without additional actions, inconsistent documents within editors are created in such situations.

In the next sections various actions which try to solve these problems are discussed. One possibility is to prevent text manipulations in the same position at the same time. A second possibility is to guarantee the total causal order of the operations by such situations. This can be done by implementing certain algorithms in the editor and in the database system.

## APPROACHES FOR FULL CONCURRENCY SUPPORT

Collaborative real-time groupware tools often use the operational transformation [Ellis 1989] [Sun 2002] approach to guarantee consistency of shared data. Algorithms like aDOPTed [Ressel 1996], GOTO [Sun 1998], SOCT 2,3,4 [Suleiman 1998] [Vidot 2000] and TreeOPT [Ignat 2003] were developed but can not be used in our native text database, because we do not have the position of a character within a text in the database (compare Figure 1 and Figure 2). Instead of that, we know the object ID of a character. Based on this situation we search in this section for a full concurrency supported solution.

**Exclusive Lock Approach**

The locking approach is a pessimistic one because it prevents the described problem situations in advance. Today's database systems offer this locking mechanism. Since the update process runs in the database completely independently of the editor processes, these locking mechanisms simply cannot be used, since in general it is not allowed to create locks over overlapping processes. The advantage of solving the problem within the database alone lies there, that different applications can access this database and profit from this supply.

In principle, the lock can be set at two different times. An operation arises by moving the cursor within the text. The current character (i.e. the one at the cursor) is then locked, to be on the safe side. This indicates a considerable load on the system since this operation occurs relatively often. A better time to invoke this lock is before an insert, delete or change operation is called. If such an operation arises, an exclusive lock must be requested for the corresponding character first.

The situation corresponds to problem 1. Two editor applications insert a character at the same position at the same time. This situation has led, as pointed out, to an inconsistent system. Using locking mechanisms can prevent such situations.

Editor A (compare Figure 2) inserts character X at the position two behind character C. This operation is passed on to the database system. First, it tries to set a lock on character C, the reference character. If this is successful, the character is inserted in the database and afterwards the update process is triggered. At the same time, editor B tries to insert character Y at the same position. Editor B reports this insert operation to the database system. The database system now also tries to lock the reference character C, but this character is already locked. This is the reason for stopping this procedure. The lock on the reference character C is canceled as soon as the update process has updated all editors and these have inserted the new character in its editor document model.

It can be shown that with such a mechanism, problem 3 can also be solved. The system can be designed so that all consistency violation requests can result in an error message on the database side.

**The update process for the exclusive lock approach in detail**

The unlock process was not discussed precisely enough in the previous part. The statement made silently assumes that the update process in the database system knows when the informed editors have updated their client document models. Then afterwards, the update process unlocks the reference character.

Two scenarios are possible. Two threads are active in the editor. The connection thread waits for incoming reports of the update process and passes these on to the update thread which analyzes the messages and updates the client document model correspondingly.

For scenario 1, the update is completed for the database update process after sending the update report successfully to the update process of the editor. But the connection thread within the editor must read the message from the buffer and arrange the update of the client document model. The update is taken care of by its own update thread for the client document model. The update is actually finished at this time.

Scenario 2 is the 'correct' process, at which the update thread informs the update process running in the database system after a successful update of the client document model. Only after the update process has received the confirmation from the editor is the lock canceled.

From this point of view, locking with scenario 1 does not completely solve problem 1, because there is a lack. After the update process has sent the message successfully, it continues to work and cancels the lock on character C (compare Figure 2) in the next step. But this does not mean that the update in the client document model is completed at this time yet. If character Y is inserted before the update of the client document model is done, since the lock was already deleted, problem 1 arises. If the insert process is very fast, it is theoretically possible that character Y is inserted before X in the client document model. A solution would be that the update process waits for a confirmation, according to scenario 2. The same statements could also apply to the problem 3 under the acceptance of scenario 1.

From a theoretical point of view, it is clear that the update process according to scenario 2 must be implemented since this prevents problems 1 and 3 from arising. However, the implementation of the update process scenario 2 turns out to be substantially difficult, as shown below.

*Update process according to scenario 1*

Editor A (compare Figure 2) inserts character X in the shown text document at the second place. Before the client document model of editor A is updated, the database system is informed about the insert of character X, by sending the new character; in this example X, and the ID 15 of the reference character C. The responsible process (called DB-Process for editor A) for the connection in the database system accepts this message and locks character C first. If character C can be locked successfully, character X is inserted in the ring of double linked fields. The new character is saved in the database and all editors, who have this document opened, must be informed about this insert operation now.

The DB process for editor A informs the update process first about this insert operation. The update process represents an independent process and works parallel to the other database processes.

After informing the update process, the DB process for editor A informs the operation triggering editor A and returns the ID of the newly inserted character X as parameter. Then the operation triggering editor A updates its own client document model. The update process informs all editors (except the operation triggering editor A) which have also opened this document at this time, by sending an update message to the corresponding editors. The update message contains the new character as well as the ID of the newly inserted character X, and the ID of the reference character. This information is enough for inserting the new character correctly in the client document model. After informing all editors successfully, the update process unlocks character C without waiting for confirmation by the editors.

*Update process according to scenario 2*

Here the update process does not immediately unlock the reference character C (compare Figure 2) after informing all editors successfully. In this scenario, the update process waits until all editors send a confirmation back that the update has successfully been brought to the client document model. After all confirmation messages arrive in the database system, the reference character is unlocked.

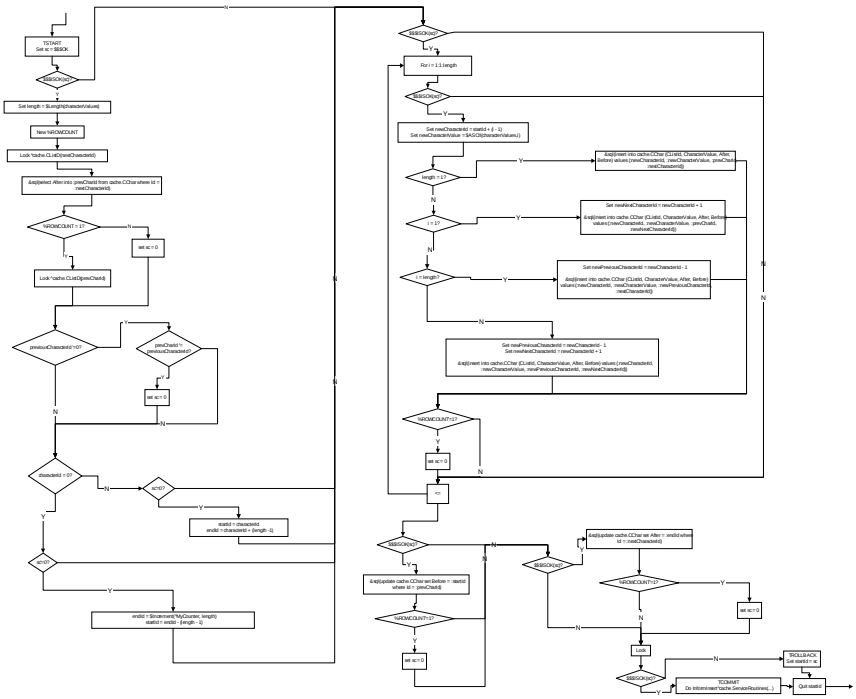
## CONCLUSION OF THE UPDATE PROCESS FOR THE LOCK APPROACH

Problems 1 and 3 can be solved with the lock approach by implementing the update process according scenario 2. Then, full consistency can be guaranteed. The disadvantage is, as illustrated, a considerable loss of performance. In addition, a different situation must be captured, like e.g. an editor is shut down before a confirmation could be sent. If this situation is not recognized, then the update process cannot cancel the lock for lack of confirmations received. If the update process according scenario 1 is used, the performance of the system is much better, but problem 1 and 3 can still occur.

### Validation Approach

The validation approach represents another possibility to solve these problems. It detects and prevents such situations. Before a new character is saved or deleted in the database, the database system validates the operation. An additional parameter within the send message is used for this validation. Validation means that the database system checks whether the client document model and the master model

*Figure 3: Insert Transaction*



are consistent in the modified text area. The demanded text manipulation is carried out only in the case of a consistency. The additional message parameter for the validation is the ID of the character after which a character shall be inserted or deleted.

Editor A (compare Figure 2) inserts a new character X and also submits the ID of the previous character in addition to the ID of the next character. In this example, the previous character is B with ID 12 and next character is C with ID 15. Editor B simultaneously inserts character Y at the exact same position and submits the same message parameters. First, an entering operation in the database system is validated. This means the ID of the next character and the ID of the previous character must lie beside each other. This must be the case before inserting the new character, provided that the client document model is consistent to the database model.

Problem situation 3 also can be solved the same way. Editor A inserts a character and editor B tries to delete character C at the same time. Before deleting, the database system checks if the previous ID still corresponds to the character before C to the ID 12. This validation reveals a temporary inconsistency, and the delete process is stopped.

From this point of view, the validation approach is comparable with the lock approach. Both approaches attempt to avoid problem situations in advance. The validation approach has, however, a better performance and is easier to implement than the lock approach.

**Approach with Sequences**

This approach does not try to prevent the problem situations but solves them by guaranteeing the total causal order of all operations with sequence numbers. The order of the operations is carried out more or less identically like in NetEdit [Zafer 2001] project.

The editor holds phase diagrams, with which the system can order the operations correctly. With this approach, problem 1 can be solved, but problem 3 still remains. To be able to solve problem 3, every editor must manage a buffer in addition, in which those operations just performed are noted. If, for example a character is deleted, then the character, the ID, and further necessary information as well are stored in the buffer. By receiving an update message based on an insert operation, the reference character within the client document model is searched. If the reference character is not found, problem 3 can be solved with the help of the buffer.

The approach with sequences is solved mainly in the editor as opposed to the other approaches.

**Conclusion of the Approaches for Full Concurrency Support**

Based on these reflections, the best performance for the synchronous approach with direct update and the asynchronous approach is the exclusive lock approach combined with the validation approach. This combination guarantees full consistency for our ring of double linked fields. In Figure 3, the insert transaction is illustrated. This transaction is not only used for inserting characters in a collaborative environment, but also for setting security, layout, structure, and flows.

## CONCLUSION

In conclusion, we have chosen a hybrid approach, where the central database management system plays a central role. Editing text means invoking database transactions which generate consistency across all editors. The consistency model depends on the update system and uses exclusive locks and validation to guarantee full consistency within a ring of double linked fields, representing documents.

The described transaction is as a scouting expedition for other database-based word processing systems. In this capacity, we have learned a number of valuable lessons, including the value of character-position/objectId duality, the relationship between editor and database as a mechanism to support consistency, and a number of interface design concerns that arise specifically from the use of characters as lowest granularity for text transactions.

## REFERENCES

Ellis, C.A. et al.: Concurrency control in groupware systems. In: Proceedings of ACM, SIGMOD, 1989, Vol. 18, 399-407.

Hodel, T.B., Technical Report University of Zurich 12003101, 2003. (http://www.tendax.net)

Ignat, C.L. et al.: Customizable Collaborative Editor Relying on treeOPT Algorithm. In: Proceedings of ACM, ECSCW, 2003.

Ressel, M. et al.: An integrating, transformation oriented approach to concurrency control and undo in group editors. In: Proceedings of ACM, CSCW, 1996, 288–297.

Suleimann, M. et al.: Concurrent operations in a distributed and mobile collaborative environment. In: Proceedings of IEEE, ICDE, 1998, 36–45.

Sun, C. et al.: Achieving convergence, causality-preservation and intention-preservation in real-time cooperative editing systems. In: Proceedings of ACM, TOUCHI (1), 63-108.

Sun, C. et al.: Consistency maintenance in real-time collaborative graphics editing systems. In: Proceedings of ACM, TOCHI 9(1), 1-41.

Vidot, N. et al.: Copies convergence in a distributed real-time collaborative environment. In: Proceedings of ACM, CSCW, 2000.

Zaffer, A. et al.: NetEdit: A Collaborative Editor. TR-1-13, Computer Science, Virginia Tech, 2001.