Design and Implementation of a Document Database Extension

Stefania Leone[1], Ela Hunt[2], Thomas B. Hodel[2],
Michael Boehlen[3], and Klaus R. Dittrich[1]

[1] University of Zurich, Department of Informatics, Winterthurerstrasse 190,
8057 Zurich, Switzerland {leone,dittrich}@ifi.unizh.ch
[2] Swiss Federal Institute of Technology (ETH Zurich), 8092 Zurich, Switzerland,
hunt@inf.ethz.ch, hodel@sipo.gess.ethz.ch
[3] Free University of Bolzano-Bozen, Piazza Domenicani 3,
39100 Bolzano, Italy, boehlen@inf.unibz.it

Abstract. Integration of text and documents into database management systems has been the subject of much research. However, most of the approaches are limited to data retrieval. Collaborative text editing, i.e. the ability for multiple users to work on a document instance simultaneously, is rarely supported. Also, documents mostly consist of plain text only, and support very limited meta data storage or search. We address the problem by proposing an extended definition of document data type which comprises not only the text itself but also structural information such as layout, template and semantics, as well as document creation meta data. We implemented a new collaborative data type Document which supports document manipulation via a text editing API and extended SQL syntax (TX SQL), as detailed in this work. We report also on the search capabilities of our document management system and present some of the future challenges for collaborative document management.

1 Introduction

Text documents are produced by the million in companies, government offices and universities. Papers, reports and business documents contain a large part of an organization's knowledge. These documents are mostly stored in a hierarchical folder structure on file servers, and are often difficult to find, even if a document management system is used.

The need to store, retrieve and edit documents in an efficient manner is obvious. However, in most companies and government offices documents can be manipulated by only one user at a time. Tools for collaborative text editing are rarely deployed in the business community. In addition, the mechanisms for access control, security and consistency of documents provided by common word-processing applications do not satisfactorily meet user demands.

On the other hand, structured data such as customer address or account data in banks is most commonly stored in a database. The data can be accessed, retrieved and manipulated through languages like SQL, and mechanisms for integrity, consistency, access control and security are provided by the DBMS by default. Since document data is of particular importance, and given that with relational DBMS and SQL as an abstract interface, the problem of storing, retrieving and manipulating structured data is optimally supported, it is obvious that document data should be treated in a similar way. Their appropriate integration into the database would solve many management issues such as data organization and querying, recovery, integrity and security enforcement, multiuser operation, distribution management and uniform tool access.

Although there are many different approaches within this area of research, the storage of weakly structured data (documents) within a database, and query and manipulation operations on these data, are not satisfactorily supported. The Lowell Database Research Self-Assessment [1] recommends an integration of text into DBMS. The authors propose to store text within the database in such a way that it can be accessed, queried and manipulated like a first-class data type (e.g. an integer or a character string). We therefore strive for an operational document database which enables the storage of, access to and manipulation of documents in databases in an integrated manner. TX SQL, the TeXt SQL extension presented in this paper, describes in detail a new *transactional* data type whose instances can be manipulated simultaneously by many users. Our data type DOCUMENT extends SQL in order to support access, retrieval and manipulation of data instances within a collaborative text editing and searching environment.

Our contributions are as follows. We define the DOCUMENT data type, and propose a set of new SQL primitives which support document creation, update and delete. We propose extensions to the SELECT statement to support structure and meta data retrieval. We provide details of our implementation which uses an object-relational database system and demonstrate how the multidimensionality of the new data types is supported in an existing prototype.

The paper is structured as follows. In Section 2 we describe the requirements, related research and TeNDaX, the underlying concept and architecture of our approach. In Section 3 we specify the collaborative data type DOCUMENT and the data type specific TX SQL statements, as well as the text search functions. In Section 4 we discuss our work and compare it to previous approaches to text management, and in Section 5 we conclude.

2 Preliminaries

We discuss the requirements for storing documents in the database, related research and the TeNDaX approach for document handling.

REQUIREMENTS

We define a document as a set of objects (characters, tables, images), structure and meta data. Structure includes layout, subdivision into chapters, sections and paragraphs, templates used, business process information, security details, semantic annotations and comments, i.e. all information relating to a document and required to display and interpret it. Additionally, the document holds meta information about the document itself, such as the title, the author(s) and the date of creation.

As suggested in [1], text should be integrated into the database as a first-class data type. To achieve this, SQL needs to be extended in order to provide operations for accessing, querying and manipulating this new data type (select, insert, update and delete). Compared to ordinary data types, a document is far more complex. A character string, for instance, simply consists of a sequence of characters of fixed or variable length, containing information about a text fragment. Thus, the string contains information which has one dimension, a word which has a certain meaning in the appropriate context. A document, however, consists of content, structure and meta data, and is therefore a multidimensional object. Implementing document as a data type is not only a question of storing the content within the database; it is also necessary to consider the document use, including the storage and querying of its structure and meta data. An SQL extension has to incorporate data type specific operations such as editing and deleting parts of documents (e.g. characters, words, sentences) and applying structure (e.g. layout, semantics) to the document. All operations on documents carried out regularly by word processing users should also be supported. In addition, it is desirable that all these operations are carried out within a collaborative environment which is supported by database transactions. Furthermore, search operations should be supported on whole documents, on document parts, on structure and on meta data.

To summarize, the Document data type should have the following properties.

- It should be stored entirely and comprehensively within the database. This includes content, structure and meta data.
- It should be accessed and manipulated by the use of SQL statements.
- It should be accessed and manipulated simultaneously by more than one user, i.e. the new data type should be collaborative, with full transactional support.

Previous work

Text storage and retrieval in databases has been the subject of much research, and various commercial database products supporting text storage and retrieval are available on the market. `SQL:1999` [15] introduced object-oriented capabilities. It defined new data types which enable the storage of large objects, such as documents and images (BLOBs and CLOBs). Additionally, it introduced user-defined data types that behave in a similar way to objects in an object oriented paradigm. `SQL/MM` [19] introduced several new data types, including Full-Text. Full-Text comes with a number of methods that enable full text indexing as well as methods that define the searches. `SQL/XML` [14] introduces an XML data type.

DB2, Oracle and SQL Server 2005 implement XML data types and provide XML access by both XQuery and SQL/XML [22] [24] [18]. XML is either stored natively, in BLOBs, or shredded to relational tables. DB2 Text Extender [12] and Oracle Text [23] are database extensions for text. DB2 Text Extender enables search and linguistic operations. Columns of different data types can be defined as TEXT columns. Different index types (linguistic, precise and ngram) can be created. Textual data is either stored in a TEXT enabled column or outside the database. Text is not by itself a data type and standard SQL functions cannot be applied to TEXT enabled columns. Transact-SQL (or T-SQL) [13] is an extension of SQL that offers many additional features such as the TEXT and NTEXT data types for storing documents and data type specific statements. The statements enable text editing functionality within a document, but text editing is reduced to content. T-SQL offers functionality for inserting into and removing characters from the TEXT or NTEXT.

SuperSQL [26] supports publishing and presenting the same text data stored in a database in different ways, depending on the query. The target media can be specified within the query. Consequently, the document structure is not persistently stored in the database but is dynamically created when executing the query. The TRDBMS project [3] handles the storage of structured text such as SGML in a database. An extension to SQL including two new data types (TEXT and GRAMMAR) is introduced, as well as specific operators for these data types. Those are stored as columns of relational tables. The new operations mainly provide access and search functionality on the document data. The presented extension is reduced to the storage of structured text data. Navarro and Baeza-Yates [20, 21] present ProximalNodes, a model that combines searching on structure and content of documents in a query language. Structure is represented in a hierarchical way as known from markup languages. Text search is enriched by conditions relating to document structure. Structure and text are kept entirely separate. The language only supports search functionality, and data manipulation operations are not provided.

TeNDaX

Neither the presented research approaches nor the SQL built-in data types for large object storage have been designed with full document manipulation functionality in mind. Our vision is to provide a transactional system for document management. This will combine full database support for all the features currently encountered in text-processing and document management applications. With this requirement in mind, we proceed to describe the existing TeNDaX system prototype, to which this paper adds several new dimensions. TeNDaX is a **T**e**X**t **N**ative **D**atabase e**X**tension which enables the storage of text in databases so that text editing is ultimately represented as an interactive transaction. By text editing we understand the following: writing and deleting characters, copying and pasting, defining layout and structure, inserting comments, setting access rights, defining business processes, inserting tables, pictures and links, i.e. all the actions regularly carried out by word processing users. By interactive transactions we mean that editing text invokes one or more database transactions so that everything which is typed appears within the editor as soon as these objects are stored persistently. Instead of creating files and storing them in a file system, the content, structure and all of the meta data belonging to the documents is stored in the database which enables very fast interactive transactions for all editing tasks. A DBMS does not usually offer very fast interactive access.

The concept and first performance measurement of the implemented prototype are described in [9]. In [8] the concurrency control approach of such a collaborative database-based text editing system is presented. The concept and implementation of collaborative layouting, i.e. the support for different users layouting a document simultaneously is described in detail in [7]. [10] introduces the TeNDaX workflow component which supports ad-hoc creation of workflows within a document assigning (parallel and serial) tasks to different users. [17] describes the real-time propagation of updates within a system where multiple clients are working on the same document.

In [11] the TeNDaX Meta Data System is presented. As all data is stored in the database, various types of information are associated with each character or document part and can later be used for the retrieval of documents. Meta data collected at character level includes the author, timestamps, copy-paste references and user-defined properties. At a higher level of abstraction (called ZONE, see Section 3.2) we store information about structure, template, layout, business process (workflow), security, comments, semantics and versions. At document level we record the creator, roles, date and time, document names, structure, comments, security, size, authors, readers, associations within static folders and user-defined properties. This meta data is of crucial importance for document retrieval functions presented in Section 3.3.

This paper extends TeNDaX with new functionality. We present three new aspects which support richer document manipulation. The first one is the design and implementation of the DOCUMENT data type, and of new types of structural information which can be superimposed onto a text document. The second is extended SQL functionality which supports document creation and update, and third are text specific query functions, implemented as extended SELECT statements.

3 Collaborative Data Type DOCUMENT

This section presents our contribution. We first specify the text editing API, see Section 3.1. We then define newly added interfaces which support the multidimensionality of the DOCUMENT data type, see Section 3.2. In Section 3.3 we define our search functions, in Section 3.4 we focus on implementation details and system architecture, and in Section 3.5 we detail the SQL extensions implemented in TeNDaX. In Section 3.6 we present some associated data types and processing functions.

3.1 Text editing API

The DOCUMENT data type corresponds to the specification presented in Section 2 and consists of content, structure and meta data. The implementation uses the data model shown in Figure 1. The API is defined using Java-like signatures and data types such as `String, int, char` and the array symbol `[]` representing an array of objects of a particular type. Three main types are used to represent

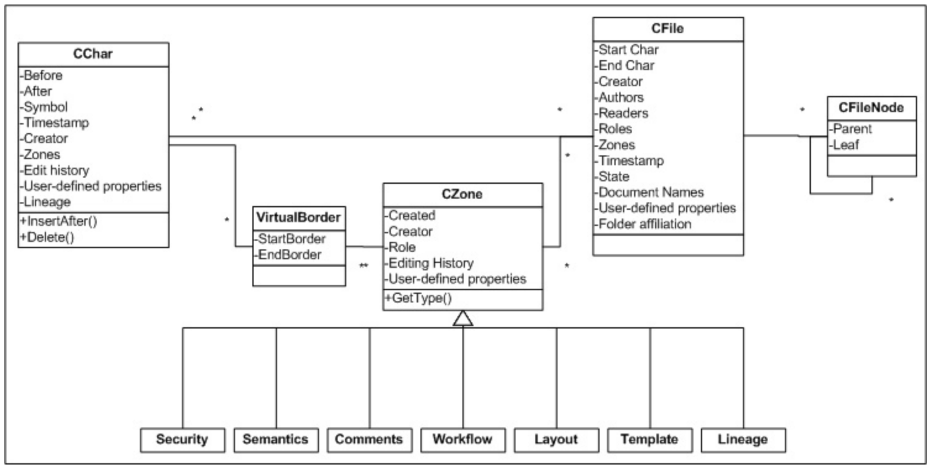


Fig. 1. A simplified view of the relationships between the character object `CChar`, the document `CFile`, the directory `CFileNode`, the zone `CZone` and the border enclosing a part of a document, `VirtualBorder`

the data.
`CChar (CChar before, CChar after, char symbol)` is a single character. It refers to its predecessor `before` and to its follower `after`.
`CFile (CChar startChar, CChar endChar)` is a document. It holds a reference to the first document character `startChar` and to the last `endChar`.
`CFileNode (CFile leaf, CFileNode parent)` is a document folder. It is a tree with a possibly empty leaf of type `CFile` and a parent folder.
Document creation correspond to the creation of a new instance of `CFile`. Text editing uses a number of methods defined on the class `CChar`. Following operations are supported.
`void insertAfter (String text)` is an atomic operation. A string is inserted, to follow the current character.
`void deleteChars(CChar [] forDeletion)` is an atomic operation which removes the characters in an atomic manner. This is a soft delete, and involves marking characters as deleted. As shown in [9], these operations are supported by transactions which guarantee database consistency in a multi-user collaborative editing environment.

3.2 Zones

Our contribution is the addition of new functions involving layout, semantics and other layers which can provide a richer document management environment. Zones are a way of handling new types of functionality and are implemented as additional object types managing pointers to the existing `CChar` objects. We previously described the mechanisms supporting layout [7] and workflow [10]

integration within a document system. In this paper we focus on new zones: Template, Security, Comment and Semantics. In summary, Layout zone captured the page layout, including font size, type, colour and type face, and the Workflow zone allowed for the invocation of workflows from within the document. Our new zone types add new functionality to the system, as described below.
Template zone corresponds to both XML Schema and style markup. XML Schema is used to define document parts and a *style sheet* can then be superimposed to add formatting to the document. If both layout and template are defined, layout will take precedence, as the zones it marks are normally smaller than the zones defined by the template. The template is document specific, so that users sharing a document will adhere to one document schema and style. On the other hand, layout is applied at the user level, and each user can see text displayed in a different format.
Security zone supports the database model of role-based access to documents and their parts. It is possible to mark documents or their parts as read- or write-protected, with regard to users and roles. This may be useful in the context of financially sensitive data or where data protection needs to be enforced.
Comment zone is analogous to a comment which can be added to the document or a part of the document, similarly to current facilities seen in word processing tools.
Semantics zone can support ontological annotations of the document or document parts. We see this as a mechanism for either user-centred or automated text annotation with ontological terms. This zone can then support a variety of search mechanisms, including visualisation of document semantics, document clustering, or automated annotation with synonyms and ontological terms, to enable document retrieval within an organisation.

Zones are created by users by highlighting a piece of text and adding the appropriate zone type with other relevant data, as currently done in word processing systems. To enable this, we first define a zone object `CZone` as an instance of one of the six zone classes `Workflow, Layout, Template, Security, Comments, Semantics`. Each of those classes refers to a `VirtualBorder(zoneStart CChar, zoneEnd CChar)` class (see Figure 1) which specifies the zone start and end and is annotated with relevant zone attributes. This is accomplished via a Create Zone SQL statement shown in Section 3.4. Each zone has one main access function `String getType()` returning the zone type. Other available functions depend on the zone type. For instance, the template zone can be created by invoking CreateZone on an empty document or importing a DTD and either adding styles manually or importing a style sheet. Alternatively, the user can select the entire document to create a template and then add XML markup specifying the DTD and the styles. If a new zone of the same type and referring to the same two character objects is added, we do not remove the old zone specification but add a new one with a fresher timestamp. In that way we store document lineage in the database and can later perform queries on document history. Zones are superimposed on the document and we have implemented functions which support the analysis of the annotations present in various zone types. The system can list all zones for a given document or document part (see Section 3.5). The implementation of the zone object is described in Section 3.4.

3.3 Searching and Ranking

We define and implement a number of methods supporting document search functionality. The method
`CFile[] getDocuments(String term, String[] searchConstraints, String[] rankingOption)`
returns an array of CFile instances (documents) containing the search term, corresponding to the search constraint(s) and ordered by the ranking option.

The searching and ranking algorithms are implemented in the classes `CIndex` and `CTerm` and make use of the TeNDaX meta data. Search is supported by an index. For the index, inverted lists are used [2]. Since documents not only consist of content but also of structure and meta data, we can benefit from the structural information and from the meta data stored in the database and extend the index by adding this information in order to provide sophisticated searching and ranking options. The two classes are defined as follows. The class
`CTerm(int frequency, String term)`
represents the vocabulary that is all the terms and their frequencies. Index entries are represented by instances of the class
`CIndex(CFile document, CChar termStartChar, CChar termEndChar, CTerm term, CUser[] user, int copied, Boolean original, CZone[] zones, Date lastChanged, int paragraphNumber, int sentenceNumber, int wordNumber).`
`TermStartChar` and `termEndChar` mark the start and the end character of the `term` in the `document`. `user` is a list of users who contributed the term. `copied` indicates how many times a term has been copied to other documents and `original` marks a term if it was originally written in that document. The `zones` array represents the zones that are superimposed on that `term`. `ParagraphNumber`, `sentenceNumber` and `wordNumber` indicate the position of the word within the `document`.

Based on these classes we can offer several searching and ranking options. The following search options are supported. Those correspond to the select statement to be shown in Section 3.5.

- `Exact` and `substring` search, both based on content.
- `Structure`: supports the definition of a structural constraint for the search term(s)
- `Role`: finds text written by a user associated with a role
- `Original`: finds text originally written in a document and not copied from somewhere else
- `Mixed`: combines search options

The search result can be ranked according to one of the following options.

- `Frequently read` : Lists the documents on top which were most frequently read.
- `Newest`: Lists the documents on top which were most recently edited.
- `Most copied`: Lists the documents according to the number of times the term has been copied within a document and to other documents.
- `In-text distance (distance)`: Lists the document on top in which the search terms have a minimal distance within the document (Can only be used for searches with two or more search terms.)

3.4 Implementation

Our system uses the object-relational system Caché. The choice of the experimental platform was dictated by the extensibility features which Caché offers and by the fact that it supports SQL, object-orientation and access to multidimensional data storage.

The implementation reported in this paper uses both relational and object-oriented concepts. The document is implemented as a list of double-linked objects of type `CChar`. Character insertion is an insert into a standard double-linked list of characters [25]. To support efficient updates, we extend `CChar` by adding an attribute marking a character as `active`, if it is part of a document, or set this field to false, if the character has been deleted.

In the object-relational implementation, `CChar` is a tuple of `(ID, before, after, char, active, zoneBorder)` and an insert is equivalent to the addition of new tuples, one for each new character, and updates involving before and after character IDs, to reflect the double-linked list semantics. The attribute `zoneBorder` marks a character as start- and/or end border of one or multiple zones. `CZone` is a tuple of `(ID, startBorder, endBorder, zoneType, zoneAttribute)`.
`startBorder` and `endBorder` point to `zoneBorder` of the start and end characters and reflect the VirtualBorder concept discussed in Section 3.2. `zoneType` is one of the zone types (discussed in Section 3.2) and `zoneAttribute` holds the corresponding attribute(s). Zone creation is the addition of a new `CZone` tuple, and an update of the `zoneBorder` attribute. If the character is already a `zoneBorder`, the attribute does not get changed.

The system architecture is shown in Figure 2. While users carry out text editing, editor applications perform data manipulation operations by calling an appropriate extended SQL statement, e.g. INSERT CHARACTERS. The statement is mapped to the corresponding Caché ObjectScript procedure, e.g. `insertAfter` and the insertion is performed as a transaction.

3.5 Extended SQL Statements

Document management functionality was implemented via extended SQL statements corresponding to methods invoked on `CChar`, `CZone`, `CFile` and `CFileNode` objects. Document creation and deletion can be done via the statements
`CREATE DOCUMENT ‘Progress Report’` and `DELETE ‘Progress Report’`.

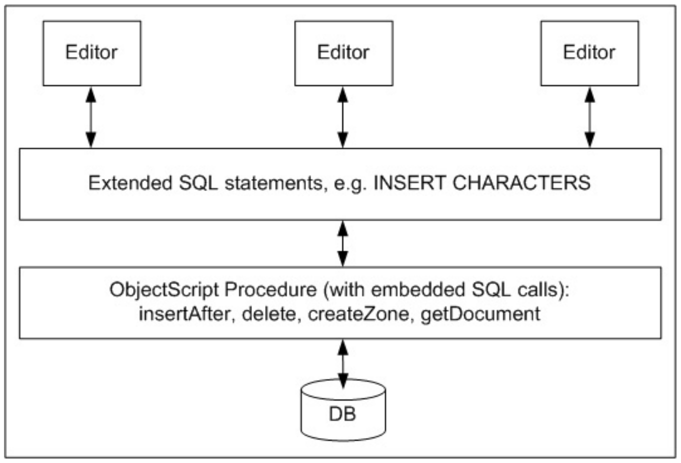

Editor
Editor
Editor
Extended SQL statements, e.g. INSERT CHARACTERS
ObjectScript Procedure (with embedded SQL calls):
insertAfter, delete, createZone, getDocument
DB

Fig. 2. System Architecture

Correspondingly, we support insert and delete SQL statements at character and zone level. The user interacts graphically with text and files, and selects objects that class methods are applied to. The graphical user interface delivers the current cursor position. This allows us to refer to the cursor position or current/previous character. As user actions are passed to the database, meta data will also be generated and stored persistently. The user interface has to pass the object and method call to the SQL engine which carries out the appropriate operation. For example, the INSERT operation is defined as `void insertAfter(String text)`. This is implemented as a transaction which performs the insert and adds appropriate meta data. A simplified extended SQL statement to support this operation would be:

```
INSERT CHARACTERS INTO <document> VALUES <String>
AFTER CHAR = <PreviousChar>;
```

We currently generate and display such statements but they are not executed directly. Instead we mimic their execution within a transaction, see Figure 2. We deliver feedback to the database manager on the execution of statements using this syntax, for instance as:

```
INSERT CHARACTERS INTO adbis06call VALUES (‘Call for papers’)
AFTER CHAR = <currentCursorPos>;
```

We proceed in a similar fashion with ZONE CREATION. The user selects an area between *selectionStart* and *selectionEnd* as highlighted by the user and adds zone characteristics by interacting with the editor interface. Our proposed extended SQL statement is

```
CREATE ZONE INTO <document>
ZONETYPE = {TEMPLATE|LAYOUT|WORKFLOW|SECURITY|COMMENTS|SEMANTICS}
ZONEDATATYPE = {<style>|<layout_information>|<workflow_information>|
 <security_information>|<comments_information>|<semantics_information}
WHERE FIRSTCHAR = selectionStart AND LASTCHAR = selectionEnd;
```

and a corresponding message sent to the DBA is

```
 CREATE ZONE INTO adbis06call
 ZONETYPE = LAYOUT
 ZONEDATATYPE = ‘font size, 14’
 WHERE FIRSTCHAR = <selectionStart> AND LASTCHAR = <selectionEnd>;
```

Another SQL extension we propose deals with DELETIONS and takes the form of a statement which refers to a highlighted portion of a document.

```
 DELETE CHARACTERS FROM <document>
 WHERE  CHARS IN (<selectionStart> .. <selectionEnd>);
```

Because of concurrency issues in collaborative editing, in our implementation the editor tool knows object IDs of the characters displayed on the screen, and the characters correspond to `CChar` objects in the highlighted area. In this statement a list of object IDs is passed down to the DB to be marked as inactive.

The SELECT statement is extended in our work to offer both document opening for reading or editing, as well as more complex search functions. A basic document open statement specifies the document name, and is (implicitly) associated with the user, her role, and a timestamp. A sample open statement is:

```
 SELECT DOCUMENT 'ADBIS draft5';
```

The syntax of the proposed TX SQL SELECT statement is presented below. The statement's return value is a multidimensional array where each array slice consists of the `CFile` ID and the start and end character of the specified document or its part. This is sufficient to rebuild the selected document section, including content and structure. Searching options (presented in 3.3) are part of the WHERE clause. For ranking, the SQL ORDER BY clause is extended. The four new ranking options `frequently read`, `newest`, `most copied` and `distance` are all based on meta data.

```
 SELECT {DOCUMENTS|CHAPTERS|SECTIONS|PARAGRAPHS|SENTENCES|WORDS}
 [WHERE [EXACT|SUBSTRING|ORIGINAL] <searchstring> |
  [STRUCTURE = <literal>|ROLE = <literal>|<metadata> = <literal>|...]
 [ORDER BY {FREQUENTLY READ|NEWEST|MOST COPIED|DISTANCE];
```

The next statement shows a SELECT example. All documents that exactly contain the terms ‘ICDE Proceedings’ are returned. The document which was most frequently read is listed on top.

```
 SELECT DOCUMENT
 WHERE EXACT ‘ICDE Proceedings’
 ORDER BY FREQUENTLY READ;
```

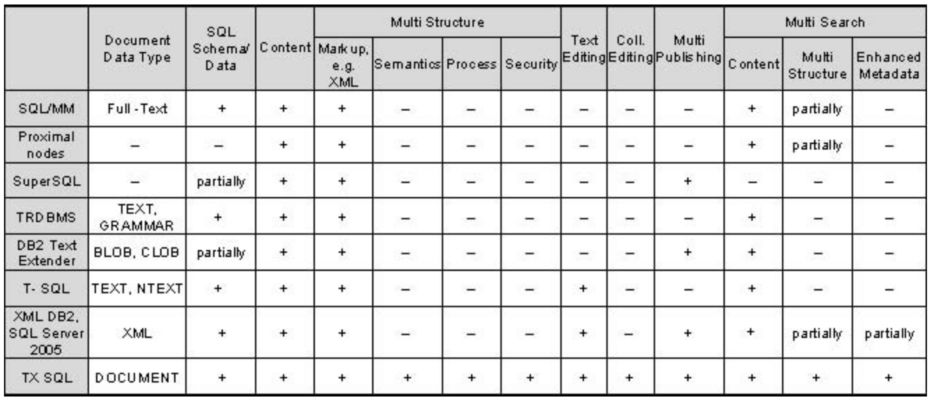
3.6 Associated data types

Rich document functionality requires additional support for data types which are parts of documents or serve as exchange formats. With this aim in mind we provide additional SQL support to handle tables, images and video. INSERT and DELETE statements are provided for table, image, and video objects. Data exchange is supported by IMPORT DOC and EXPORT DOC SQL statements. Import and export statements support document conversion to and from XML, PDF, HTML, and DOC. They also facilitate conversion between ASCII and Unicode.

4 Discussion and Future Work

We will discuss the following two issues: system functionality and the relationship between our work and the issue of data provenance.

Figure 3 gives an overview of various requirements for the integration of document data into the database, and compares our work with previous approaches, discussed in Section 2.



	Document Data Type	SQL Schema/ Data	Content	Multi Structure				Text Editing	Coll. Editing	Multi Publishing	Multi Search		
				Markup, e.g. XML	Semantics	Process	Security				Content	Multi Structure	Enhanced Metadata
SQL/MM	Full-Text	+	+	+	–	–	–	–	–	–	+	partially	–
Proximal nodes	–	–	+	+	–	–	–	–	–	–	+	partially	–
SuperSQL	–	partially	+	+	–	–	–	–	–	+	–	–	–
TRDBMS	TEXT, GRAMMAR	+	+	+	–	–	–	–	–	–	+	–	–
DB2 Text Extender	BLOB, CLOB	partially	+	+	–	–	–	–	–	+	+	–	–
T-SQL	TEXT, NTEXT	+	+	+	–	–	–	+	–	–	+	–	–
XML DB2, SQL Server 2005	XML	+	+	+	–	–	–	+	–	+	+	partially	partially
TX SQL	DOCUMENT	+	+	+	+	+	+	+	+	+	+	+	+

Fig. 3. Overview of the presented approaches

In the following we refer to table columns. DOCUMENT DATA TYPE: Several approaches provide a data type for textual data but only our approach adopts the definition of a document as consisting of content, structure and meta data. SQL SCHEMA/DATA SUPPORT, CONTENT: Most of the approaches support standard schema and data statements and all approaches support the storage of text content. MULTI STRUCTURE refers to the support and storage of document structure such as markup, semantics, business process and security. Markup is most commonly supported. With the different zone types (semantics, layout, template, process (workflow), security and comments), TX SQL is the only approach offering multi-structure support at an arbitrary level of granularity. TEXT EDITING is a basic requirement which is supported by some of the approaches, whereas

Collaborative editing is only supported by TX SQL (via text editing API, and extended SQL including statements such as insert characters). Multi publishing refers to document export in different formats and encodings, which is a key requirement for interoperability. However, only a limited number of approaches support such functionality. TX SQL fulfils that requirement with the previously presented export doc statement.
Multi-search refers to searching and ranking based on content, structure and meta data. Whereas most of the approaches support content search, search on structure and meta data is only supported by our TX SQL.

Our work is related to current research into data provenance [27], [5]. TENDaX records and manages document provenance metadata, and supports provenance queries. Current implementation results in a storage overhead of a factor of eight, as compared to MS Word. This storage overhead can be reduced significantly using well-proven storage reduction methods. For instance, Buneman et al. [4] show that their transactional-hierarchical storage approach reduces the storage overhead by a factor of five and also reduces query processing costs in comparison with a naive storage approach.

4.1 Future work

We plan to carry out extensive performance measurements in the near future as well as a user acceptance test. Additionally, we plan to extend the search facilities by supporting approximate matching using an ngram index [16]. We will consider ideas expressed by Chaudhuri [6] on combining flexible scoring with query optimization techniques.

Finally, we would like to benchmark the system against other implementations.

5 Conclusion

This paper presents a TeXt SQL Extension for an operational document database system. The system stores documents in an object-relational database. It provides full collaborative text editing and multi-search functionality as well as data definition, retrieval, and manipulation for the data type Document. Document data is integrated into the database as a first-class data type, as proposed in [1] and is the first *collaborative and transactional* document data type we know. TX SQL offers an abstract interface to the TeNDaX System and makes the implemented functionality available for other word-processing applications.

References

1. S. Abiteboul et al. The lowell database research self-assessment. *Commun. ACM*, 48(5), 2005, 111-118.
2. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press, 1999.

3. G. Blake et al. Text / relational database management systems: Harmonizing SQL and SGML. In *Applications of Databases*, 1994, 267-280.
4. P. Buneman et al. Provenance management in curated databases In *SIGMOD 2006*.
5. P. Buneman et al. Why and where: a characterization of provenance Management. In *ICDT 2001*, 316-330.
6. S. Chaudhuri et al. Integrating DB and IR technologies: what is the sound of one Hand Clapping In *CIDR 2005*.
7. T.B. Hodel et al. Supporting collaborative layouting in word processing. In *CoopIS/DOA/ODBASE 2004*, 355-372.
8. T.B. Hodel and K.R. Dittrich. A collaborative, real-time insert transaction for a native text database system. In *IRMA 2004*.
9. T.B. Hodel and K.R. Dittrich. Concept and prototype of a collaborative business process environment for document processing. *Data & Knowledge Engineering, Special Issue:Collaborative Business Process Technologies, Elsevier Science Journal*, 2004, 61-120.
10. T.B. Hodel et al. Dynamic collaborative business processes within documents. In *SIGDOC 2004*, 97-103.
11. T.B. Hodel et al. Using text editing creation time meta data for document management. In *CAiSE 2005*, 105-118.
12. IBM DB2 Universal Database. Text extender administration and programming. http://www-306.ibm.com/software/data/db2/extenders/text/, 2000.
13. D. Iseminger. *Microsoft SQL Sever 2000 Reference Library*. Microsoft Press, 2000.
14. ISO/IEC 9075–14:2003. *Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML)*, 2003.
15. ISO/IEC 9075-2:1999. *DatabaseLanguageSQL Part2: Foundation(SQL/Foundation)*, 1999.
16. N. Koudas et al. Flexible String Matching Against Large Databases in Practice. In *VLDB 2004*, 1078-1086.
17. S. Leone et al. Concept and architecture of a pervasive document editing and managing system. In *SIGDOC 2005*, 41-47.
18. Z.H. Liu et al. Native Xquery processing in Oracle XMLDB. In *SIGMOD 2005*, 828-833.
19. J. Melton and A. Eisenberg. SQL Multimedia and Application Packages (SQL/MM). *SIGMOD Rec.*, 30(4), 2001, 97-102.
20. G. Navarro and R. Baeza-Yates. A language for queries on structure and contents of textual databases. In *SIGIR 1995*, 93-101.
21. G. Navarro and R. Baeza-Yates. Proximal nodes: a model to query document databases by content and structure. *ACM Trans. Inf. Syst.*, 15(4), 1997, 400-435.
22. M. Nicola and B. van der Linden. Native XML support in DB2 Universal Database. In *VLDB 2005*, 1164-1174.
23. Oracle Corporation. Oracle Text Application Developer's Guide 10g Release 2 (10.2). http://download-east.oracle.com/docs/cd/B19306_01/text.102/b14217.pdf, 2005.
24. S. Pal et al. XQuery implementation in a relational database system. In *VLDB 2005*, 1175-1186.
25. R. Sedgewick. *Algorithms*. Addison-Wesley, 1983.
26. M. Toyama. SuperSQL: an extended SQL for database publishing and presentation. In *SIGMOD 1998*, 584-586.
27. J. Widom. Trio: a system for integrated management of access, accuracy, and lineage. In *CIDR 2005*, 262-276.