The VLDB Journal (1997) 6: 296–311

The VLDB Journal

Structured document storage and refined declarative and navigational access mechanisms in HyperStorM*

Klemens Böhm, Karl Aberer, Erich J. Neuhold, Xiaoya Yang

GMD-IPSI, Dolivostraße 15, D-64293 Darmstadt, Germany; {kboehm, aberer, neuhold, yangx}@darmstadt.gmd.de

Edited by Y.C. Tay. Received April 22, 1996 / Accepted March 16, 1997

Abstract. The combination of SGML and database technology allows to refine both declarative and navigational access mechanisms for structured document collection: with regard to declarative access, the user can formulate complex information needs without knowing a query language, the respective document type definition (DTD) or the underlying modelling. Navigational access is eased by hyperlink-rendition mechanisms going beyond plain link-integrity checking. With our approach, the database-internal representation of documents is configurable. It allows for an efficient implementation of operations, because DTD knowledge is not needed for document structure recognition. We show how the number of method invocations and the cost of parsing can be significantly reduced.

Key words: OODBMSs – SGML – Document query languages – Navigation

1 Introduction

1.1 Objective of this work

With open systems, such as the World-Wide Web (WWW), and document exchange formats where markup can be placed within the document at ease, notably HTML [HTM], there neither is control over the structure of individual documents nor over consistency of the document collection. With the combination of SGML (Standard Generalized Markup Language) [ISO86] and database technology, on the other hand, one can specify the logical structure of documents, make assumptions about their structure and ensure the consistency of the document collection. This allows to refine both declarative and navigational access mechanisms.

Declarative access. With regard to declarative access to a document collection, the prevailing question in literature in the past few years has been: which is the most expressive query language? However, the more expressive the query language, the more complex it is. In addition to having an interface allowing the formulation of complex queries, ease of use of the search mechanisms has a high priority for a lot of users. In this article, we describe mechanisms for declarative access, so-called *query templates*. They allow the user to search with a fairly high expressive power without having to know a query language, the respective document type definition (DTD), or the underlying modelling. Different query templates can be made available for individual user groups.

A query template is a document-type-specific query form that is generated automatically from the corresponding DTD and an additional query template specification. With a query template, only a limited set of element and attribute types is made available for querying. Hence, the query template can be seen as a view mechanism. The query template specification is administered by the database application. This ensures consistency of the query template with the respective DTD. In consequence, only meaningful queries can be generated from the user input.

With regard to query templates, we exploit the fact that there is a DTD, i.e., a scheme that documents of a certain type must conform to. On the contrary, if HTML documents were to be administered, the following problems would be in the way of having query templates and making proper use of them.

- *No support for individual document types.* While trying to provide coverage for all kinds of documents, HTML is highly generic. SGML, however, has been developed to reflect the particularities of different application scenarios by using different document types. Queries can be much more specific if the document type may be taken into account.
- *Meta information is not available or may not be consistent.* With appropriate DTDs, any kind of metadata[1] might be seamlessly included into documents to cover users' information needs [KSS95]. With HTML 3.2, consistency of metadata in different documents is not en-

* HyperStorM is an acronym for Hypermedia Document Storage and Modelling

[1] Throughout this article, the terms 'metadata' and 'metainformation' are used interchangeably.

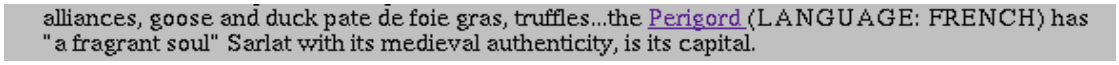


Fig. 1. Example for hyperlink rendition

sured. The HTML DTD does not contain any guidelines on how metainformation should be modelled.
– *Difficult to consider documents' internal structure.* As opposed to SGML DTDs specified for a particular application scenario, HTML allows to model documents' logical structure only in a generic way. In consequence, search mechanisms are only a subset of the ones for full-fledged SGML documents.

Navigational access. Another aspect of applying both SGML and database technology is support for navigation. The following problems are addressed:

– *Navigation failure.* Navigation to a document may fail for a variety of reasons: links to other documents may be outdated, i.e., the document referenced may have been removed or moved to another location in the meantime. Navigation to a document referenced may not be possible because the reader is not entitled to access it, or because the client is unable to present the (multimedia) document, e.g., a viewer for the respective format may not be available at the client site.
– *Expectations vs. actual content.* Frequently, the reader has no meaningful information on the document referenced by a hyperlink within the current document or on the set of documents to be reached by following a hyperlink anchor. Experience shows that quite often expectations awakened by the context of a link anchor are not matched when actually viewing the document. In other words, in order to find out more about the target of the link, there seems to be no alternative to bringing the document to the client site.
– *Lost in hyperspace.* Traversing several hyperlinks may have the effect that the reader looses orientation in the document collection. Being "lost in hyperspace" [Con87] stems from the fact that hypermedia objects, e.g., the WWW pages, are arranged in a non-linear way. The reader may have to inspect several such objects to satisfy his information need. He may not be able to find all the relevant objects by navigation or to avoid the irrelevant ones.

These problems are alleviated by our approach. With WWW-based access to the document base, a document is converted to HTML according to a conversion specification before presentation. This specification is contained in a style sheet. Conversion takes place within the database. Namely, rendition of a hyperlink anchor pointing to another document may depend on that document's characteristics.[2] In particular, metadata on the document referenced, as well as documents indirectly referenced, may be used for link anchor rendition. As a simple example, consider a document collection where the language of the document is made explicit in the document header, i.e., a document with title 'Perigord' contains '<LANGUAGE>FRENCH</LANGUAGE>'. If document 'Perigord' is referenced by another document, the respective hyperlink anchor in the layout version of the document may look as in Fig. 1. This alleviates the expectations vs. actual content problem. In this example, if the reader does not speak French, he knows that he does not want to access the document.

With hyperlink rendition, information on the document directly referenced, but also on documents indirectly referenced, may be taken into account. Information to be displayed within the hyperlink is specified in the database query language. To speed up the process of retrieving a document from the database, HTML conversion results may be materialized in the database.

If we again compare the administration of HTML documents to our approach, we face similar problems as with declarative access. Metainformation may not be available for document rendition, or one cannot tell which metainformation has been made explicit within documents. Inconsistent representation of metadata likewise is a problem.

1.2 Database-internal representation of documents

An efficient database-internal representation of structured documents is a prerequisite to realize the functionality that has been outlined so far. We advocate a hybrid database-internal representation of documents. Only "big" elements are represented by individual database objects. Different "small" elements, so-called *flat elements*, are mapped to the same database object. The structured representation of documents is advantageous to allow fine-grained modifications of documents in the database and to reflect the semantics of hypermedia document components, whereas performance of certain basic operations, such as insertion of a document into the database, is better with an unstructured representation [BAK95]. With regard to the mapping of flat elements to database objects, there are design alternatives that are described in this article.

Flat elements are completely marked up in the database. This allows a more efficient implementation of methods operating on the document structure, as compared to parsing the document fragment (see Sect. 4). Another aspect is that elements can be unambiguously identified by means of a logical object identifier. If, finally, methods are part of the database, as is the case with an OODBMS, method invocations are relatively expensive.[3] It is advantageous to take this into account when implementing the methods.

[2] Throughout this article, a document containing the anchor of a link referencing another document will be called *referencing document*, the other document will be referred to as the *document referenced*.

[3] Our terminology is different from the one used in [Sto96]. Object-oriented DBMSs in our terminology essentially correspond to object-relational ones in [Sto96]. There is no counterpart in our terminology for object-oriented DBMSs in that terminology, which basically are persistent object-oriented programming languages without sophisticated declarative access mechanisms.

The database-internal representation of documents is configurable. The initial configuration is described by means of an SGML document, an instance of the so-called *super-DTD*. The configuration may be partially altered at runtime. Using SGML to describe the startup configuration has the following advantages:

- The consistency of this specification is checked by means of an SGML parser.
- To specify the initial configuration, the specification document is inserted into the database in the same way normal documents are. No differentiation is necessary from the user's perspective.
- The DTD designer can specify the startup configuration using a mechanism he is familiar with.
- The DTD designer may use SGML tools to fulfill his task.
- The mechanisms described in this article for querying the document collection, notably query templates, are also available for document types as documents. In other words, the document type is explicitly available, in analogy to database scheme information contained in a data dictionary.

The remainder of this article has the following structure: in the following section, related work is reviewed. Section 3 contains a brief review of SGML concepts and introduces some notions that are relevant in this context. In Sect. 4, we describe the database-internal representation of documents in our framework. Section 5 describes how declarative access functionality can be extended by using SGML and database technology. Section 6 focuses on navigational access. Section 7 concludes the paper.

2 Related work

We have classified related work as follows: related work with regard to document modelling, related work with regard to declarative access, and related work with regard to navigation and HTML conversion.

Related work with regard to document modelling. In order to represent the documents' logical structure using an OODBMS, it seems feasible to carry out a 1:1 mapping from elements to database objects. Furthermore, there would be database classes, so-called *element-type classes*, corresponding to an element type from the DTD. However, if this logical view is identical with the physical representation, the following problem will arise: the duration of certain basic operations such as inserting documents into the database or retrieving documents from the database is almost directly proportional to the number of database objects that are created or retrieved, respectively. This may not be acceptable, as others have observed, too [NBY95]. An alternative seems to be the approach described in [ACM93, ACM95]. They consider structured data whose physical representation is flat, in particular data within files. If the structure is needed for, e.g., query evaluation or updates, the document is parsed, and objects in main memory are generated. Our work differs from theirs in the following respects:

- With our approach, a document is not necessarily represented by one file. Rather, the document may be physically fragmented in the database, and the fragments' logical structure can be recognized using, e.g., the techniques described in [ACM93, ACM95]. The database-internal representation is configurable. One advantage is that a finer granularity is possible with regard to concurrent write access to the document, using concurrency control mechanisms provided by the DBMS.
- With our database-internal representation, the DTD is not needed to recognize documents' logical structure. More specific techniques than parsing the document are applied. The advantages will be pointed out in Sect. 4.
- An element has an OID whose lifetime is independent from the existence of a corresponding object in main memory.

Related work with regard to declarative access. The following issues are of interest with regard to work on declarative access to document collections having originated in the database community [Mac91, C+94, MMM96, QRS+95, B+94, ST94, O+95].

- With our approach, expressiveness of the query language is achieved by using methods of the database scheme, together with OQL query mechanisms [Cat94]. Compared to other approaches, the expressive power is higher, while, on the other hand, it is not necessary to extend the query algebra. Our approach allows for full-fledged information retrieval functionality (IR functionality) [VAB96], which is different from search on a syntactic level [SM93], as well as search on documents' physical characteristics.
- With many of the above references, information with regard to documents' database-internal representation is incomplete or missing.
- To the best of our knowledge, work cited above does not contain any counterparts to query templates or the hyperlink rendition mechanisms described in this article.

In more detail, work described in [C+94] is based on OODBMS technology. They have extended the underlying query algebra to reflect notions such as the lengths of paths. By using OQL that allows inclusion of methods into query statements, and having an adequate set of methods as part of the database scheme, the same expressiveness can be obtained without extending the query algebra. In [B+94], a data type 'structured text' is introduced to be integrated into relational database systems and an extension of SQL is defined. To facilitate updates, the approach is to map SGML structures to tables, but conformance to the DTD remains to be ensured. The PAT query algebra [ST94] lacks certain features, such as the notion of position, querying according to documents' secondary structure, and aggregation. Further, only elements can be retrieved. It is, however, independent of the data model and will be dealt with in Sect. 5 again. While in [O+95] a user interface for an SGML/HyTime document database has been realized, work seems to have been centered around one particular document type.

In [YA94], a coupling of a DBMS and a text search engine is described. There, documents' internal structure is

not modelled within the database. The text engine used there does not support the notion of vagueness. The need for IR functionality, e.g., ranking, is acknowledged in [SDAMK95]. Their objective is to build an integrated system providing both database and IR functionality. Details about documents' internal representation are not revealed. We, for our part, have realized a loose coupling between the OODBMS VODAK [VML95] and the IR system INQUERY [CCH92] to make IR functionality available for database content. With a loose coupling, we will be able to rather easily incorporate improved IR functionality whenever it becomes available.

An objective of others, e.g., [QRS+95, MMM96], is to provide declarative access mechanisms for open-ended systems where assumptions about the data's structure cannot be made, notably the WWW. Even though WWW-related issues currently draw a lot of attention, the question how to exploit consistency of the document collection in controlled environments remains relevant.

Related work with regard to navigation and HTML conversion. With Hyper-G [AKM95], a principal objective is to ensure hyperlink consistency. The idea is that there is a link database. It contains the information which hyperlinks exist between documents. The advantage, as compared to the current status of our work, is that there is no confinement to the content of one database. This is reached by giving up some of the individual information servers' autonomy. Hyper-G is not modular, but, rather, can be seen as "another web", as, for example, proprietary browsers have to be used. In our context, a mere link database would not be sufficient, as arbitrary information on documents referenced can be requested. An unanswered question is whether people are willing to take into account the additional overhead of "a WWW without dangling references". On the other hand, the need to ensure consistency of local document collections clearly exists [S+94]. Conversion of SGML documents to HTML is the topic of [Fre]. The notion of *location grammar* is introduced as a means to specify context-sensitive transformation of element types. It seems that, there, context sensitivity refers to documents' hierarchical structure, but not to other documents. The topic of [TEI94], similarly, is structured document handling in the Internet. They argue that it is the SGML document that should be delivered to the client to facilitate so-called *document post-processing*. Trivially, our database server can also return the original SGML documents. If the DTD allows for it, hyperlink rendition, as outlined above, is still feasible. Annotation servers contain information on WWW documents that may be provided by others. Instead of directly bringing the document to the client, the document goes through the annotation server, and relevant information is added [RMW94].

DSSSL [ISO96] is an expressive language to specify document transformation. The standard specifies a structured representation of documents; conversion is based on that representation. The standard does not deal with the question how to efficiently carry out such a conversion if documents are within a storage system, and if characteristics of documents referenced are taken into account. With our approach, characteristics of the documents referenced can be reflected. Rendition mechanisms for hyperlink anchors likewise are expressive, and are identical with our declarative access mechanisms.

```
<agenda author=Aberer>
  <header><language>English</language>
    <subject>future research topics for the
          department</subject>
    <location>...<date>...
    <invited>
      <name>Fischer</name><name>Chen</name>...
    </invited></header>
  <programme>
    <item>brief review of present funding
          situation: in 1996/97 ...</item>
    <item>problems with diploma thesis students:
          due to the fact ...</item>...
  </programme>
<agenda>
```

Fig. 2. Sample SGML document of type 'Agenda'

3 Modelling metainformation with SGML

The practical relevance of SGML has considerably increased in the recent past. This is possibly due to the close connection between SGML and HTML, the format of WWW documents. Within SGML documents, the logical document components, so-called *elements*, are made explicit by means of markup. The document fragment from Fig. 2 is an example of a marked-up SGML document. '`<item>`', '`</item>`' identify (the start position/the end position of) an element of type `item`. '`<`' is the *start tag open (STAGO)*, using SGML terminology, '`</`' is the *end tag open (ETAGO)*, and '`>`' is the *tag close (TAGC)*. It is an important aspect of SGML that markup may not be arbitrarily chosen and placed within documents. Rather, for each document type, a DTD has to be provided. It specifies which element types may occur in a document, and how elements may be arranged within a document. A DTD is a grammar. The `Agenda` DTD is contained in Fig. 3. Examples of element types from this DTD are `header`, `programme`, and `item`. The regular expression specifying the admissible content of an element of the respective type is referred to as *content model*. For instance, '`(header, programme)`' is the content model of `agenda`: an `agenda` element contains a `header` element, followed by a `programme` element. The expression '`+(keyword)`' is an example of an *inclusion model*. It specifies that the structure within the brackets, in this case an element of type `keyword`, may occur arbitrarily within an element of type `programme`. For instance, the element '`<programme><item>brief review of present <keyword>funding</keyword> situation ...</item> ...</programme>`' conforms to the DTD from Fig. 3. *Exclusion models* are also available to forbid such inclusions in a subtree of the subdocument. If the element type definition of `item` was `<!ELEMENT item (#PCDATA) -(keyword)>`, the above sample element of type `programme` would not conform to the sample DTD any more. `CDATA` and `(#PCDATA)` are terminal element types comparable to the data type `STRING`. Elements may be furnished with attributes. Again,

```
<!DOCTYPE agenda [
<!ELEMENT agenda     (header, programme)>
<!ATTLIST agenda     author  CDATA  #IMPLIED>
<!ELEMENT header     (language?, (location|roomno),
                     date, invited)>
<!ELEMENT programme  (item)+     +(keyword)>
<!ELEMENT (language|location|roomno|date|name
            |keyword)   (#PCDATA)>
<!ELEMENT invited    (name)+>
<!ELEMENT item       (#PCDATA)>
]>
```

Fig. 3. Sample DTD (document type 'Agenda')

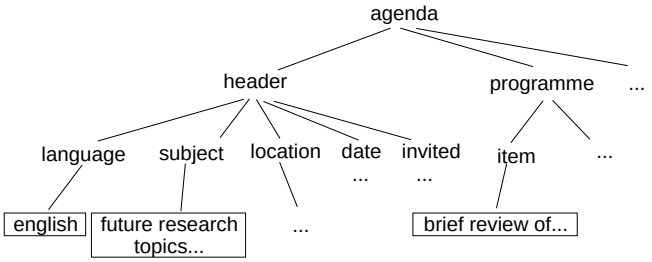


Fig. 4. Sample document's hierarchical structure

the attributes cannot be chosen freely, but must be contained in the DTD. `#IMPLIED` means that a value does not have to be assigned to the respective attribute.

Classifying element types. Element types can be categorized by the role of their instances within the documents [HHM94]. This classification is important, as access patterns, notably with regard to declarative access, are different for the individual categories.

- *Structural element types.* Markup of such elements is used to identify documents' logical structure. Examples from the sample DTD are `programme` or `item`.
- *Non-structural element types.* Non-structural elements are individual words or short sequences of words within structural elements' content having a particular role, e.g., element type `keyword` in the Agenda DTD. In other words, markup of non-structural element types is used to make explicit the meaning of words within text. In most cases, non-structural elements are not bound to structural element types, but may occur rather freely within the text.
- *Informational element types.* Informational elements are metainformation. While non-structural elements occur within actual document text, informational elements do not occur within structural elements' textual content. Rather, they tend to be contained in a document header. Typically, informational element types that do not have an internal structure could also be modeled as SGML attributes, while non-structural element types cannot. An example of an informational the element type is the element type `language` from the DTD in Fig. 3.

Using this categorization of element types according to their roles, we are now in the position to describe how metainformation can be modelled with SGML.

- Informational elements are metadata.
- The markup of structural elements and non-structural elements is metainformation. This is different from informational element types, where the elements themselves are metainformation.
- Elements, normally structural elements, can be furnished with attributes, as described above. The attribute values are metainformation.

Furthermore, the DTD itself can be seen as metadata. Namely, the different ways to represent metadata, as described above, must be complemented with the type definition for meaningful interpretation.

Further SGML mechanisms. The SGML concepts that have been described in this section are merely a subset of SGML. It is the subset for which support is described in the following. In our approach, SGML entities and marked sections are resolved by the parser and do not occur any more within the document in the database. Hence, the mapping of a document to the corresponding database content is not loss-free. Furthermore, notations and the SGML link mechanism are not supported.

4 The HyperStorM database application framework to administer structured documents

The structure of this section is as follows: the database-internal representation of documents is described in the following subsection; configurability mechanisms are described in Subsect. 4.2. In the last subsection, the transformation algorithm from documents to their database-internal representation is presented.

4.1 Reflecting the SGML information model

This subsection covers design decisions and issues with regard to the database-internal representation of structured documents within the database.

1. Hybrid database-internal representation for documents: some elements are represented by individual database objects, while others, the flat ones, are not. This representation is subject to configuration for the particular document type, and the respective configuration mechanisms will be described in the sequel.
2. Flat elements are completely marked up within the database.
3. Elements have a logical OID whose life cycle is independent of the existence of corresponding (C++)objects in main memory.
4. The query language of our system is OQL, together with methods from the database scheme. With methods as part of the query language, expressiveness of the declarative access mechanisms is naturally higher than in other approaches. Method invocations are costly with methods being part of the database. This must be reflected with their implementation.

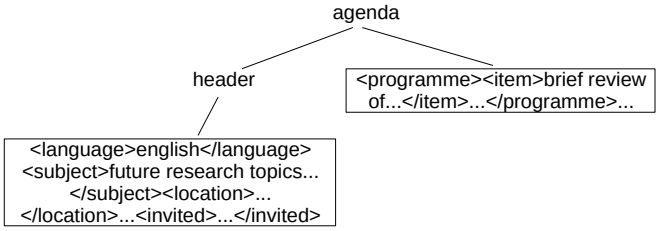


Fig. 5. Possible physical representation of 'Agenda' document

To reflect documents' internal structure, not only documents, but also document components are explicit within the database. A differentiation between flat and non-flat elements is made (cf. Sect. 1.2). Database objects corresponding to flat elements are *flat objects*. The string representation of a flat object's elements is the *flat string of the database object*. As an example of documents' database-internal representation, consider the document from Fig. 2. One out of many representations that are possible within the database is given in Fig. 5. With that particular configuration, `language` and `subject` are examples of flat element types. '<language>English...</invited>' and '<programme> <item>brief...</programme>' are examples of flat strings. '<language>English... </invited>' is the flat string of the language and the subject element. The hybrid database-internal representation facilitates modifications of document fragments and better reflects the semantics of hypermedia document components. It reduces the negative impact of a structured physical representation with regard to performance [BAK95].

Structure recognition of flat elements. As just explained, elements in the database can either be flat or non-flat. While, in SGML, it is allowed to omit markup if the document structure can be unambiguously recognized by means of the DTD, document fragments within the database are completely marked up. Markup that may have been omitted from the original documents is added. Consequently, the document structure can be recognized without the DTD (see the top right fragment in Fig. 6 as an example, as opposed to the top left one). Simple linear access operations are sufficient. The advantages of not using the DTD are the following.

- If the DTD was used for structure recognition, it might seem feasible to construct a fragment DTD on-the-fly. However, DTDs are not context-free due to inclusions and exclusions. Hence, to construct the fragment DTD, one would have to inspect the inclusion and exclusion models of the elements the current flat elements are contained in. This requires a number of access operations to database objects that are unnecessary in our approach. As an example, consider the following clipping from a DTD.

  ```
  <!ELEMENT A (C)* +(G)>
  <!ELEMENT B (C)*>
  <!ELEMENT C (D?, E+)>
  ```

  In order to construct a fragment DTD for an element c of type `C`, in particular, the inclusion model of type `C`, one must check if c is contained in an element of type `A` or `B`. In the first case, the fragment DTD must reflect that c may contain an element of type `G`, as opposed to the second case.
- Structure recognition is more efficient without the DTD: if documents of different types are in the database, it is not necessary to look up the respective type first.
- In a DTD-based approach, a flexible fragmentation of documents in the database is not possible in practice. As an example, consider the bottom left database object in Fig. 5. The corresponding document fragment '<language>...</invited>' does not have a root element. In consequence, either a document fragment suitable for parsing would have to be constructed first. '<header> <language>...</invited> </header>' would be such a parseable fragment. However, this requires access to at least one more database object, namely the header object. If this object contained non-flat elements, further database-access operations would be necessary. Alternatively, concatenation of flat elements to build a flat object would have to be forbidden, i.e., the `language` element, the `subject` element, etc. would be separate database objects. But this may lead to a large increase in the number of database objects.

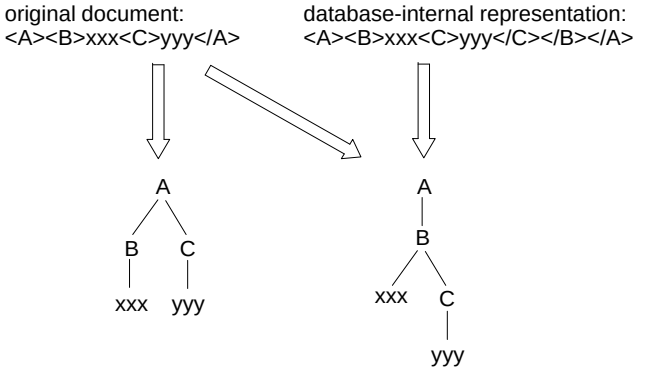


Fig. 6. Inferring the document structure from complete and incomplete markup

Object identifier. Object identity is an important notion in object-oriented modelings. The necessity of OIDs for both flat and non-flat elements introduces a logical and a physical object level. The logical view remains that there be an object corresponding to each element. On the physical level, however, this is not the case. A logical OID consists of a physical OID and the STAGO position within the corresponding flat string, i.e., the byte offset. If the respective element is a non-flat one, the offset is -1. The DBMS has been extended so that it can transparently support method invocation on objects identified by logical OIDs. With message calls, the DBMS resolves logical OIDs and dispatches them to the physical objects. In the parsing approach described in [ACM93, ACM95], object identifiers are available only as long as the corresponding structure in main memory exists.

Classes and methods of the database scheme. The following classes are part of the database scheme.

ELEMENT - The physical database objects representing the document structure are instances of this class, e.g., the nodes in Fig. 4.

ElementType - For each element type from a DTD, there is an instance of the class.

DTD - An instance of this class corresponds to each DTD currently supported.

Document - For each document, there is a corresponding object.

The methods for elements include the following.

```
hasTextualContentRegex (r: REGEX): BOOL
hasAttrValueRegex (attrName: STRING, r: REGEX): BOOL
getIRSValue (q: STRING): REAL
isContainedIn (e: logicalOID): BOOL
getReferencedElements (attrName: STRING): {logicalOID}
getAttrValue (attrName: STRING): STRING
getSize (): REAL

getAll (elementTypeName: STRING): {logicalOID}
getFirst (elementTypeName: STRING): logicalOID
getElementText (): STRING
```

Instances of ElementType have method getElements(): {logicalOID}.

Method hasTextualContentRegex returns TRUE iff r is contained in the target element's textual content. Method hasAttrValueRegex returns TRUE iff the value of attribute attrName contains r. getIRSValue returns the belief value of the element's textual content with regard to IRS query q, as computed by the underlying IRS. isContainedIn returns TRUE iff the target element is contained in the parameter element e. If attribute attrName of the target element of getReferencedElements is of type IDREF(S), the logical OIDs of the elements referenced (within the same document) are returned. Otherwise, the empty set is returned. getAttrValue returns the value of attribute attrName. getAll returns all elements of type elementTypeName that are contained in the target element; getFirst returns the first element of type elementTypeName (in pre-order) that is contained in the target element. getElementText returns the target element's textual content. getElements returns the logical OIDs of all elements of the type.

With regard to methods hasTextualContentRegex and getIRSValue, some comments are appropriate. Data administered by a storage system may be subject to different paradigms. In the case of some element types, one wants to search their instances with exact mechanisms, i.e., by means of pattern matching on the syntactic level such as regular expression search. Such search mechanisms are in place for element types such as SURNAME or PART_NO. In this case, method hasTextualContentRegex should be used. On the other hand, the objective of IR search is to cover the user's information need by going beyond the syntactic level. Results of IR queries are never precise and may differ from system to system, as the content of a piece of text may be seen differently by different systems. In the IR context, it is too undifferentiated to merely say 'The document is relevant.' or 'The document is not relevant.'. Rather, relevance is expressed by means of a belief value b such that $b \in [0;1]$. The belief value is the probability that the document is relevant with regard to the query, as computed by the system. As a rule of thumb, IR mechanisms for text only work well for texts containing more than 20–30 words, they do not work for individual words or short sequences of words. In consequence, search on the syntactic level makes sense for informational element types, i.e., metadata, while IR search mechanisms should only be applied to structural element types, i.e., raw data.

Not only individual concepts, but also complex query terms in the IRS query language may be parameters of method getIRSValue. With INQUERY being the underlying IRS, parameters such as '#and(HyTime, MHEG)', '#not(Java)', or '#uw10(SGML, HTML)' can be processed. (The last expression specifies that 'SGML' and 'HTML' must occur within a window of 10 words.) In combination with other search mechanisms, this gives rise to a powerful search functionality.

The set of methods reflects our practical experience and is now stable. Methods hasTextualContentRegex, hasAttrValueRegex, getIRSValue, isContainedIn, getReferencedElements, and getSize are necessary to formulate queries corresponding to terms in the extended PAT language, and none of these methods can be omitted without lowering expressiveness (cf. Sect. 5.2). Method getAll is an example of a method that is needed for more efficient query evaluation, as compared to isContainedIn. getFirst and getAttrValue go beyond the expressiveness of the extended PAT language.

Example of method implementation. Method next identifies the right sibling of the target element in the logical document structure. In the sample document from Fig. 4, the next element of the subject element is the location element. Method next makes use of method getPositionOfETAGO. getPositionOfETAGO returns the (byte offset) position of the end tag open which corresponds to the start tag whose (byte offset) position is the method parameter. The method illustrates that, with our database-internal representation, operations on documents' logical structure are feasible without directly using knowledge on the document type. Furthermore, method implementation is specific for our database-internal representation of documents. For instance, it is a prerequisite that a flat object must not contain any other object.

```
(1)  next(): logicalOID {
(2)      IF (SELF is a flat element) {
(3)          p := position of SELF within flat string;
(4)          p := SELF -> getPositionOfETAGO (p);
(5)          f := flat string of SELF;
(6)          p := f ->> find (p, STAGO)     //if the next element is
                    contained in the same flat
                  //database object, it begins after the end tag of the
                    target element
                  //'find' starts to search at byte offset identified by
                    the first parameter.
                  //It returns the byte offset where the second param.
                    has been found, otherwise -1.
(7)          IF (p > -1) {  //next element is contained in the same
                    flat
             //database object, as its begin markup has
                    been found
(8)          compute logical OID from p;
(9)          RETURN logical OID just computed; }; };
(10)     convert the (physical) OID of the next database object to
                    logical OID;
               //trivial to identify next database object with
                    structured representation
(11)     RETURN logical OID just computed; };
```

In the database, tag delimiters, e.g., STAGO, TAGC, are represented by special characters so that they cannot be mistaken with symbols '<', '>' within text, and search becomes more efficient.

Improving method performance. With OODBMSs, database method invocation is costly. Knowledge of the physical representation can be used to cut down the number of method invocations, and to reduce the parsing effort. In particular, it is worthwhile to avoid recursiveness. Consider the following implementation of method getAll.

```
(1)  getAll (E: ElementType): {logicalOID} {
(2)      r := {};
(3)      IF (SELF is a flat element) {
(4)          p := position of SELF within flat string;
(5)          p_end := SELF -> getPositionOfETAGO (p);
(6)          f := flat string of SELF;
(7)          WHILE ((p < p_end) AND (p > -1)) {
                     //make sure that only elements within
                     //target element are retrieved
(8)                  p := f ->> find (p, concatenate (STAGO,
                     TypeName (E)))
(9)                  IF (p > -1)
(10)                     IF ((isWithinBeginMarkup (p, f))
                         AND (E == type name of the element
                          whose begin markup includes position p)) {
                             //make sure that, e.g., AUTHORS is
                               not found instead of AUTHOR
(11)                         l := logical OID computed from p;
(12)                         r := r ∪ {l}; }; }; }
(13)     ELSE
(14)         DO (children of SELF, element, e)
                 //iterate over the children of SELF
(15)             r := r ∪ (e -> getAll (E));
(16)     IF (ElementType (SELF) == E)
(17)         r := r ∪ {SELF};
(18)     RETURN r; };
```

On the contrary, a straightforward implementation would be recursive for all elements (as opposed to the one above that is only recursive for non-flat elements). Based on the MMF DTD [S+94], we have conducted experiments to verify that the first version is more efficient. If all elements are flat, and the root element is the target element of the original method invocation, the first version is faster by a factor of approximately 1000. Naturally, the difference becomes smaller with fewer flat elements. If no elements are flat, the performance of the two versions is nearly identical.

4.2 Configurability mechanisms

It is subject to configuration which elements are represented by individual database objects and which ones are flat. The configuration mechanisms are described next.

With our database application, documents of arbitrary type can be administered. Insertion of documents consists of the following steps:

1. The corresponding DTD is parsed. If the DTD is correct, a parser for instances of the DTD is generated. Furthermore, the DTD is (on a syntactical level) transformed to an SGML document that conforms to a specific DTD, the so-called *super-DTD*. The super-DTD is a DTD whose instances are DTDs. In the sequel, we will refer to any DTD different from the super-DTD as *application DTD*. For instance, the DTD from Fig. 3 is an application DTD. It corresponds to the super-DTD-instance in Fig. 7. (A fragment of) the super-DTD itself is contained in Fig. 8.
2. At this point, the super-DTD instance contains exactly the information from the DTD. Attribute `elemName` of element type `ELEM`, to give an example, contains the element type name, attribute `contentModel` contains the content model as a string. Furthermore, the super-DTD instances generated in Step 1 contain additional attributes that, initially, are instantiated with a default value. These attributes essentially contain information on the physical representation of element types or attribute types. For example, type `ELEM` has an attribute `FLAT`: value `NO` signifies that such elements are represented by individual database objects, `YES`, on the other hand, stands for a flat database-internal representation. By means of further attributes, the index structures are specified.[4] Summing up, in this step, i.e., Step 2, the physical representation of documents of a certain type is configured.
3. The document generated in Step 2 is parsed by a super-DTD parser. In addition to checking the document's conformance to the DTD, the parser invokes database commands that generate the database objects that represent the document.
4. A database-internal bootstrap operation is invoked that, given the document inserted in Step 3, generates the corresponding database classes, index tables etc.
5. Now, documents conforming to that application DTD can be inserted into the database. The document parser

```
<DOCTYPE docName=AGENDA ...>
<ELEM elemName=AGENDA ... contentModel='( HEADER ,
    PROGRAMME )' ...>
  <ATTRIBUTE attrName=AUTHOR attrKeyDecl=CDATA
    attrKeyDef=IMPLIED ...>
</ELEM>

<ELEM elemName=HEADER ... contentModel
    ='( LANGUAGE ?,( LOCATION |
ROOMNO ), DATE , INVITED )' ...></ELEM>

<ELEM elemName=PROGRAMME ... contentModel='( ITEM ,
    ITEM *)' ...></ELEM>
```

Fig. 7. Fragment of the super-DTD instance corresponding to 'Agenda' DTD

```
<!ELEMENT ELEM   (ATTRIBUTE*)>
<!ATTLIST ELEM   elemName   NAME  #REQUIRED
                 contentModel  CDATA  #IMPLIED ...>
<!ELEMENT ATTRIBUTE    EMPTY>
<!ATTLIST ATTRIBUTE    attrName  NAME
                       #REQUIRED ...> ...
```

Fig. 8. Fragment of the super-DTD

[4] Index structures can be turned on or off at a later stage by means of method invocations. The flat-/non-flat configuration, however, cannot be modified any more. Such a reorganization of the database would be extremely costly, and the need for such functionality has not yet arisen in our context.

that has been generated in Step 1 not only checks conformance to the DTD, but also invokes database operations generating the corresponding database representation, updates index tables, etc.

In summary, the physical representation of documents is configurable, with element or attribute types being the granules of configurability. The dimensions of configurability are orthogonal to each other and transparent to the application programmer.

4.3 The transformation algorithm from documents' logical structure to their physical representation

In the sequel, we give the transformation algorithm that generates a document's database-internal representation from its logical structure. We will prove that the output of the transformation algorithm has certain important characteristics.

By definition, *Element type B is directly contained in element type A with regard to DTD D* if B occurs in the content model of A in D.

Definition 1. *Element type B is contained in element type A with regard to DTD D if*

1. *B is directly contained in A with regard to D, or*
2. *there is an element type C such that B is contained in C with regard to D, and C is contained in A with regard to D, or,*
3. *in D, A has an inclusion model that contains B, or*
4. *there is an element type C in D such that C has an inclusion model containing B, and A is contained in C with regard to D.*

The following lemma allows to derive information on the document type from a document that conforms to the underlying DTD. Due to the complex definition of containment on the type level, the lemma is not trivial. For instance, if an element a is directly contained in an element b, one cannot infer that the element type of a occurs in the content model of the element type of b (because of inclusions). From another perspective, the lemma shows that Definition 1 is meaningful.

Lemma 1. *If an element a of type A is directly contained in an element b of type B in some document of type D, then A is contained in B with regard to D.*

Proof. The proof is by induction on the depth of the document tree.

- a is directly contained in b, and b is the root of the document.
  In this case, for a to be directly contained in b, either
  1. a occurs in the content model of B, or
  2. a occurs in the inclusion model of B.

  In both cases, it follows directly from the definition that A is contained in B.
- a is directly contained in b. Furthermore, 'b contained in c. $\Rightarrow$ B contained in C.'.
  For a to be directly contained in b,
  1. a occurs in the content model of B, or
  2. a occurs in the inclusion model of B, or
  3. a occurs in the inclusion model of an element type C, and there is an element c of type C such that a is indirectly contained in c.

  It follows from items 1, 3, 4 from the definition (corresponding to items 1, 2, 3, respectively) that A is contained in B.

Lemma 2. *If an element a of type A is contained in an element b of type B in some document of type D, then A is contained in B with regard to D.*

Proof. The lemma immediately follows from Lemma 1 and item 2 in the definition of 'contains'.

In the transformation algorithm, the function isFlat with signature isFlat (E: Element Type): BOOL is used. It returns TRUE if E is contained in an element type that has been marked as flat in the corresponding super-DTD instance. In the algorithm, the document is traversed recursively in a depth-first-like manner. If the type of the current element is not flat, a new database object is created and inserted into the tree structure that is already there (lines 14-23). Otherwise, the current element's string representation is just appended to the current database object, which is flat (lines 6-12).

The transformation algorithm is as follows:

```
(1)  transform (e: Element, lastElementWasFlat: BOOL, currentObj:
       OID, parentObj: OID,
       root: BOOL): OID {
(2)  IF (Type (e) -> isFlat()) {
(3)       IF (NOT (lastElementWasFlat)) {
(4)            currentObj := ELEMENT -> new();
(5)            insert currentObj as rightmost child of parentObj; };
(6)       IF (Type (e) is terminal element type)
               //e.g., CDATA, (#PCDATA)
(7)            currentObj -> append (textualContent (e))
(8)       ELSE {
(9)            { currentObj -> append (BeginMarkup (e));
(10)           DO (children of e, element, e')
               //iterate over the children of e, e' is loop var.
(11)                c := transform (e', TRUE, currentObj, parentObj,
                    FALSE);
(12)           currentObj -> append (EndMarkup (e)); }; }
(13) ELSE {    //current element type is not flat
(14)      currentObj := ELEMENT -> new();
(15)      IF (NOT (root))
(16)           insert currentObj as rightmost child of parentObj;
(17)      store ElementTypeName (e) with currentObj;
(18)      store Attributes (e) with currentObj;
(19)      currentElementIsFlat := FALSE;
(20)      c := NULL;
(21)      DO (children of e, element, e')
               //iterate over the children of e
(22)           { c := transform (e', currentElementIsFlat, c,
                    currentObj, FALSE);
(23)           currentElementIsFlat := isFlat (Type (e')); }; }; };
```

ELEMENT is the database class described before, while Element is the type of SGML elements. The initial invocation of transform is transform (root, TRUE, NULL, NULL, TRUE). The actual implementation of the algorithm is non-recursive. Namely, an SGML parser has been extended to control the transformation that does not work recursively. Note that the database objects generated are untyped, i.e., they may either contain flat element types or represent non-flat elements. We

say that *the database object is flat* or *the database object is non-flat*, respectively. By definition, an object becomes a flat one or a non-flat one by means of the assignments in lines (7), (9), (12) or in lines (17), (18), respectively. Thus, the definition of flat and non-flat database objects is an algorithmic one. From now on, this definition of flat database objects replaces the previous one.

The implementation of methods reflecting the SGML semantics such as getAll is based on the following lemmas.

Lemma 3. *After a type (i.e., either flat or non-flat) has been assigned to a database object, the type does not change any more in the course of the transformation algorithm.*

Proof. "⇒": Consider a flat database object. The assignments making this object a non-flat one occur in lines (17), (18). The object is generated immediately before (line (14)). In consequence, it cannot happen that a flat object is subject to the assignments making it a non-flat one.

"⇐": An object that is already non-flat cannot become a flat one later. Namely, non-flat objects are generated in line (14) only. It can easily be seen that such an object does not become currentObj any more in the course of transformation after having specified that it is non-flat.

The following lemma shows that transformation by means of the algorithm is sound (cf. our remark on the implementation of next, and such knowledge has also been used for the implementation of getAll (lines (3)-(12))).

Lemma 4. *A non-flat object is never contained in a flat one.*

Proof. Suppose a non-flat object was contained in a flat one. Then, there is a non-flat element e_1 that is directly contained in flat element e_2. This requires that either

1. ElementType (e_1) occurs in the content model of ElementType (e_2), or
2. there is an element type F s.t. ElementType (e_1) occurs in the inclusion model of F, and there is an element e s.t. ElementType (e) = F and e_1 is contained in e.

Case 1 cannot happen because ElementType (e_1) would have to be a flat one. With regard to Case 2, it follows from Lemma 2 that ElementType (e_2) is contained in F, and ElementType (e_1) is contained in F. The last item from Definition 1 implies that ElementType (e_1) is contained in ElementType (e_2). This, however, is a contradiction to the definition of isFlat, because, in that case, ElementType (e_1) would have to be flat.

A variant of the transformation algorithm is used in the context of document modification, i.e., in order to insert elements into documents that are already in the database.

5 DTD-specific and generic declarative access mechanisms

By using SGML and database technology, we have come up with query mechanisms for a document collection characterized by the following features: (1) Formulating expressive queries is possible without knowing a query language, the DTD, the underlying data model. (2) For different user groups, different mechanisms can be generated, closely matching the user group's needs. The description of these mechanisms and how to configure them covers a large part of this section. This query mechanism, though expressive, provides for a lower degree of expressiveness than others, as we will show. Our conclusion is to let the user choose between various, in our case three, query mechanisms differing with regard to expressiveness, but also with regard to intuitiveness and user-friendliness. In addition to query templates, there are extensions of the PAT algebra and OQL, together with methods from the scheme. We will show that the extended PAT algebra is more expressive than query templates, and that OQL together with a relevant set of methods is more expressive than the extended PAT algebra. On the other hand, however, in order to formulate queries with the individual mechanisms, the user must have different levels of knowledge, as indicated in Table 1. More precisely, 'n' in the second column does not include the language of regular expressions and the underlying IRS, and 'n' in the third column does not reflect that the user has to understand the semantics of element and attribute type names.

Table 1. Knowledge necessary to use different query mechanisms

	Modelling	(Syntax of the) query language	DTD
OQL	y	y	y
PAT	n	y	y
Templates	n	n	n

5.1 Query templates

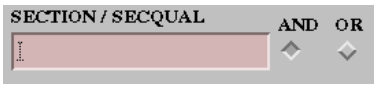
Query templates are automatically generated document-type-specific query forms. They may contain widgets of different types. It is subject to configuration which widgets are part of a query template. The following widgets are part of the framework.

– Entry field for element content search. The figure is an example of such a widget, as seen in a WWW browser. The user has to type in a list of regular expressions, each of them separated by a blank space. The operation corresponding to the widget takes all elements of the respective type, in this case SURNAME. If AND is selected, it returns all documents containing elements of the type that contain all of the regular expressions. If OR is selected, it returns all documents containing elements of the type that contain one of the regular expressions.

– Entry field for IR search (information retrieval search). An entry field for IR search actually consists of two fields, as can be seen in the figure. The user must type a concept to be searched for in the first entry field and a threshold value t in the second one. It must hold that t $\in$ [0;1). The corresponding operation takes all elements of the respective type. It returns all documents containing those elements that match the concept with a likelihood greater than the threshold value, as computed by

the underlying IRS. Instead of a concept, a query in the language of the underlying IRS can also be typed in.

– Entry field for attribute search. The corresponding operation takes all elements of the corresponding type, in this case SECTION. If AND is selected, it returns all documents containing elements whose value for attribute SECQUAL contains all of the regular expressions that have been typed in. If OR is selected, it returns all documents containing elements whose value for attribute SECQUAL contains one of the regular expressions that have been typed in.



– Entry field for structure search. The corresponding operation takes all elements of the first type, in this case SURNAME. For all such elements that are contained in one of the second type, in this case AUTHOR, and contains all of the regular expression that have been typed in, the corresponding document is returned, if AND has been selected. Analogously, with OR, only one regular expression must be contained.

– Entry field for search for physical characteristics. As opposed to the other atomic entry fields, these entry fields are hardcoded. However, they can be turned on or off by means of the configuration mechanisms. At this point, there is an entry field for document size allowing specification of a lower and upper bound.

The overall structure of a query template is depicted in Fig. 9. The left column of widgets is for the document to be retrieved, the right column will be explained below. Results corresponding to individual entry fields in a column are combined using logical AND. Only those entry fields are considered where something has been entered.

More complex queries can be formulated using the widget for secondary structure search. Furthermore, one wants to specify documents by means of the (link) relationships that exist with other documents. In addition to those two columns of widgets, there is a pulldown menu with the following options: [NO LINKS], an element-type name/attribute-type name pair followed by a right arrow, and an element-type name/attribute-type name pair followed by a left arrow. The semantics of the menu items is as follows:

– If [NO LINKS] is selected, the documents matching the entries in the left column are retrieved. Entities in the right column are ignored.

– If '-> <E>/<A>', e.g., '-> HYPLINK/REFERENC', is selected, selection is based on all pairs of documents (d_1, d_2) such that d_1 matches the template entries in the left column, and d_2 matches the entries in the right column. The query returns all documents d_1 that contain an element of type E; this element has attribute A with value n, and n is the name of d_2.

– If '<- <E>/<A>', e.g., '<- HYPLINK/REFERENC', is selected, selection is based on all pairs of documents (d_1, d_2) such that d_1 matches the template entries in the left column, and d_2 matches the entries in the right column. The query returns all documents d_1 such that d_2 contains an element of type E; this element has attribute A with value n, and n is the name of d_1.

The menu for secondary structure search is also subject to configuration.

The distinction between regular expression search and IR search has been reflected by means of methods hasTextualContentRegex and getIRSValue in Sect. 4. Analogously, query templates may contain both fields for element content search and for IR search. As pointed out before, not only individual concepts, but also complex query terms in the IRS query language may be typed into entry fields for IR search. Consequently, it is not necessary to provide an AND/OR toggle for this widget type.

Specifying query templates. The DTD alone is not sufficient as a basis for automatic generation of query templates. Frequently, one wants to make available only a restricted set of types for declarative access. This corresponds to the notion of 'view' in the context of conventional database systems. The motivation why views should be part of the framework is as manifold as it is with view mechanisms in conventional systems. In principle, we see two alternative ways of specifying query templates.

1. The super-DTD is extended so that its instances contain the query template specification. Different ways of modelling the specification are conceivable. For example, there may be an additional element type QUERYFORM with attributes of type IDREFS. These references point to the different element and attribute types to be included in the template. The type definition of QUERYFORM may be as follows:

```
...
<!ELEMENT QUERYFORM   EMPTY> ...
<!ATTLIST QUERYFORM   ...
                      CONTENTSEARCH   IDREFS
                      ATTRSEARCH      IDREFS
                      IR_SEARCH       IDREFS
...>
```

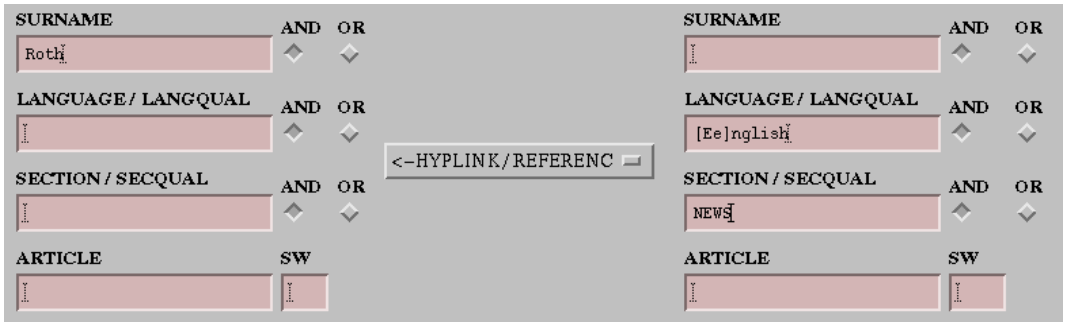


Fig. 9. Query template generated from the MMF-DTD

2. Each query template has a specification contained in a file.

With regard to item 2, as one may need different templates for one document type, one may also want to freely add new views over time. However, it is important to ensure the consistency of the query template specification with the DTD. Otherwise, queries could be generated, for which a solution cannot exist, and the user would not even notice it. But an operation which directly reads the specification from a file and checks for its consistency would be too time-consuming with large DTDs. Hence, query template generation must consist of two steps: First, the specification is read from a file, its consistency to the DTD is checked, and it is inserted into the database. Then, a database-internal, consistent version of the specification can be accessed. We have realized the first alternative and are now implementing the second one.

5.2 Other declarative access mechanisms and a comparison of their expressive power

An extension of the PAT algebra. The PAT algebra, originally described in [ST94], is a query language independent of the underlying data model. In our extension of the PAT algebra, query terms are generated by the grammar

```
e ->  <Element-type name> |
      e UNION e|
      e INTERSECT e|
      e DIFF e|
      CONTENT_SELECT (e, r)|
      ATTR_SELECT (e, A, r)|
      IR_SEARCH (e, c, t)|
      e INCLUDS e|
      e INCL-IN e|
      REFERENCES (e, A, e)|
      REF-BY (e, A, e)|
      ID-REFER (e, A, e)|
      ID-REF-BY (e, A, e)|
      LB-SIZE (e, s)|
      UB-SIZE (e, s)|
      (e)
```

The term <Element-type name> stands for the set of all elements of the respective type. UNION, INTERSECT, and DIFF are set operators with the usual semantics. CONTENT_SELECT takes a set of elements and returns those where the content contains regular expression r. ATTR_SELECT takes a set of elements and returns those where attribute A contains regular expression r. IR_SEARCH takes a set of elements and returns those matching concept c (or the IR query c) with a probability greater than t, according to the underlying IRS. INCLUDS and INCL-IN take two sets of elements E_1 and E_2 and return the set of elements

E_1 INCL-IN E_2 = $\{e_1 \in E_1 \mid \exists e_2 \in E_2\ s.t.\ e_1$ *is contained in* $e_2\}$

E_1 INCLUDS E_2 = $\{e_1 \in E_1 \mid \exists e_2 \in E_2\ s.t.\ e_1$ *contains* $e_2\}$

REFERENCES, REF-BY, ID-REFER, and ID-REF-BY take two sets of elements E_1 and E_2 and return the set of elements

REFERENCES(E_1, A, E_2) = $\{e_1 \in E_1 \mid \exists e_2 \in E_2\ s.t.\ e_1$ *has attribute A with value v, and v is name of the document in which* e_2 *is contained in*$\}$

REF-BY(E_1, A, E_2) = $\{e_1 \in E_1 \mid \exists e_2 \in E_2\ s.t.\ e_2$ *has attribute A with value v, and v is name of the document where* e_1 *is contained in*$\}$

ID-REFER(E_1, A, E_2) = $\{e_1 \in E_1 \mid \exists e_2 \in E_2\ s.t.$ e_1, e_2 *are contained in the same document,* e_2 *has an attribute of type ID with value v,* e_1 *has attribute* A *of type* $IDREF(S)$ *containing* $v\}$

ID-REF-BY(E_1, A, E_2) = $\{e_1 \in E_1 \mid \exists e_2 \in E_2\ s.t.$ e_1, e_2 *are contained in the same document,* e_1 *has an attribute of type ID with value v,* e_2 *has attribute* A *of type* $IDREF(S)$ *containing* $v\}$

LB-SIZE takes a set of elements and returns those whose size is greater than s, UB-SIZE returns those elements whose size is smaller than s.

The extensions, as compared to the original algebra [ST94], are the distinction between search on a syntactic level and IR search, the fact that documents' secondary structure has been taken into account, and the fact that documents' physical characteristics have been considered.

OQL queries. The expressive power of OQL stems from the fact that methods from the database scheme can be used within queries at liberty. The structure of an OQL query is the same as with SQL. The select clause specifies what is to be selected. The from clause specifies which database classes, or, more generally, which sets the query refers to. The where clause contains a condition that must be fulfilled

by the query result. All variables occurring in the query must be bound in the from clause. The reader is referred to [Cat94] for more information on OQL.

Illustrations. For illustration purposes, consider the query template in Fig. 9. The template entries correspond to the query "Select all documents containing an element of type SURNAME whose textual content contains 'Roth', and that are referenced by a document containing an element of type SECTION whose value of attribute SECQUAL contains 'NEWS' and containing an element of type LANGUAGE whose value of attribute LANGQUAL contains 'English' or 'english'.". The corresponding extended PAT expression is

```
REF-BY (MMF INCLUDS CONTENT_SELECT (SURNAME,
          'Roth'),
        REFERENC,
        HYPLINK INCL-IN (MMF INCLUDS ATTR_SELECT
            (SECTION, SECQUAL, 'NEWS'))
          INCLUDS ATTR_SELECT (LANGUAGE, LANGQUAL,
            '[Ee]nglish'))
```

The corresponding OQL expression is

```
select D0.name
from D0 in Document, D1 in Document,
     P0 in D0.root -> getAll ('SURNAME'),
     P1 in D1.root -> getAll ('SECTION'),
     P2 in D1.root -> getAll ('LANGUAGE'),
     P3 in D1.root -> getAll ('HYPLINK')
where P0 -> hasTextualContentRegex ('Roth') and
     P1 -> hasAttrValue ('SECQUAL', 'NEWS') and
     P2 -> hasAttrValue ('LANGQUAL', '[Ee]nglish') and
     P3 -> getAttrValue ('REFERENC') = D0.name
```

The following lemmas reflect the expressive power of the different mechanisms.

Lemma 5. *The extended PAT language is more expressive than query template entries.*

Proof. The proof is by defining a mapping from query template entries to expressions in the extended PAT language. The full mapping is given in [Boeh97]. To illustrate the mapping, consider the widget for attribute search. Let E be the respective element-type name, and A be the attribute-type name. With r_1 ... r_n being the input to the respective field, the corresponding expression is <root-element-type> INCLUDS (ATTR_SELECT (E, A, r_1) π ... π ATTR_SELECT (E, A, r_n)) with $\pi \in \{$UNION, INTERSECT$\}$. In the opposite direction, it is obvious that, e.g., the extended PAT expression A INCLUDS B INCLUDS C cannot be mapped to any query template input.

Lemma 6. *The extended PAT query language is less expressive than OQL, together with the methods given in Sect. 4.*

Proof. The proof is by defining a mapping of extended PAT expressions to OQL statements. The proof is recursive over the structure of query algebra terms. Again, the full mapping is contained in [Boeh97]. As an example, let Q be the OQL query corresponding to the PAT expression e. Then ATTR_SELECT (e, A, r) is mapped to

```
select p from p in Q where (p -> hasAttrValueRegex ('<A>', '<r>'))
```

In the opposite direction, it is obvious that, e.g., the OQL query

```
select p, p -> getFirst ('CHRNAME')
from e in ElementType, p in e -> getElements()
where (e.name == 'AUTHOR')
```

cannot be mapped to any extended PAT expression.

For evaluation, both input to query templates and extended PAT expressions are mapped to OQL expressions. - Declarative access mechanisms are also relevant in the following section.

6 Hyperlink rendition mechanisms in HyperStorM

With WWW-based access to the database application, documents can be converted to HTML. Conversion is specified by means of a stylesheet contained in a file.

Hyperlink rendition. Documents may contain references to other documents. Usually, such references are made explicit within the document with hypertext anchors. With our system, rendition of anchors of links pointing to other documents in the database may depend on characteristics of the documents referenced. As a special case of such rendition, only anchors of sound links are converted to HTML anchors to avoid some cases of navigation failure. The layout specification specifies how link anchors are encoded in documents of the respective type.

In Sect. 4, it has been described how the physical representation of documents and document components can be configured using the super-DTD. These mechanisms, however, are not used to specify document conversion for the following reasons:

- An initial configuration must have been specified before documents are inserted into the database. This does not have to be the case for the conversion specification. Furthermore, a higher degree of flexibility and ease of modification is necessary with the conversion specification, as compared to the configuration specification. It is appropriate if the configuration is altered by means of method invocations, but this is too complicated and inflexible for the conversion specification.
- From an organizational perspective, while the database-internal configuration should be specified by the DTD designer, this is not necessary for document rendition, as readers' individual preferences may be reflected.
- The super-DTD has been designed to represent information on individual (element or attribute) types. But the super-DTD instance would become too big if information from several stylesheets was included. There should be no restrictions to the number of stylesheets.

Incorporating information on documents referenced into hyperlink anchors. With our system, information on documents directly or indirectly referenced can be used to render the corresponding hyperlink anchor in the referencing document in a very flexible way. The core idea is that, for an element type whose instances are hyperlink anchors, the stylesheet contains a database query. The query specifies the information to be included in the hyperlink anchor. For this purpose, the full expressive power of OQL can be exploited.

In this context, there are two problems impeding why OQL queries cannot just be written down and executed during document conversion.

1. Query results must be of a type that can be displayed within an HTML document.
2. Within such queries, one would like to refer to the particular hyperlink anchor. For example, one would like to formulate the query "Select all elements of type AUTHOR within the document that is referenced by the hyperlink whose anchor is currently being rendered.". So far, there is no straightforward way to refer to the hyperlink whose anchor is currently being rendered in a query.

The solution to item 1 is as follows: the query is parsed. Then, the query result type is looked up. If the type is not available for display, e.g., because the query result is a set of database OIDs or a set of instances of bulk types, the query in the stylesheet is ignored. The current version of the system accepts only sets of strings.

With regard to item 2, OQL queries within the stylesheet may contain the symbols ANCHORELEM and LINKATTR. Before query evaluation, the symbol ANCHORELEM is replaced by an expression that can be interpreted by the query processor. In a nutshell, it is the logical OID of the anchor element. Since the replacement is carried out within the database, it is ensured that the new query is correct. Consider the following example:

```
select   p -> getElementText()
from     d in Document, p in d.root -> getAll ('LANGUAGE')
where    (d.name = (ANCHORELEM -> getAttrValue (LINKATTR)))
```

The query selects all LANGUAGE elements in the document referenced (cf. Fig. 1).

Expanding instances of element types is meaningful for informational and non-structural element types. For instance, one may want to display all instances of a non-structural element type KEYWORD in the referencing document. The usefulness of such an expansion for structural element types is limited, except when these element types are at the same time informational, such as, say, TITLE.

Summing up, the drawbacks pointed out in the introduction are alleviated as follows: checking for a link target before activating the anchor prevents navigation failure. Displaying metainformation alleviates the "expectations vs. actual Content" problem and is a measure against "getting lost in hyperspace".

With the functionality described so far, document conversion should be part of the services offered by the database. It would be an unnecessary step to generate SGML from the database content first and then transform the result to HTML. Furthermore, conversion on the client would be inefficient because of unnecessary communication over the network to obtain information on the documents referenced. Besides that, and most importantly, materialized views are used to avoid executing queries during document conversion. Their consistency is ensured by the database system.

Using materialized views. It is advantageous to materialize the conversion result within the database. In principle, an arbitrary number of different views on the same document being the result of different conversion parameters is conceivable. To cope with the requirement of materializing more than one view, but being restricted to a limited number of materializations, we have implemented a simple page replacement strategy (LRU). With our system, documents within the database may be modified. Then, materialized views have to be updated after the corresponding documents have been altered. The problem is aggravated by the fact that updates of individual documents cannot be seen in isolation. Instead, documents that directly or indirectly reference the modified document also have to be taken into account. In this context, we assume that each document can reference any other document. Assertions that would allow restrictions of this assumption would be helpful, but cannot be made in the general case. With our system, a view is updated when a document is accessed, if any document has been modified after the view's last generation date. With frequent updates, a conceivable refinement is to differentiate between modifications having an impact on conversion of referencing documents and modifications without such effects. In this case, however, conversion specifications have to be administered by the database.

7 Conclusions

One objective of this article was to point out how the combination of database technology and SGML can be exploited in order to ease access to the document collection, both declaratively as well as by means of navigation. In order to ease declarative access, a mechanism to express information needs has been designed that is expressive, while, at the same time, neither knowledge of a query language nor the document type or the underlying modeling are needed to use it. This so-called query template mechanism can be configured to match the needs of different user groups. We exploit the fact that SGML documents conform to a DTD. Such a DTD is comparable to a database scheme in that both can be seen as integrity constraints on the data. HTML would not be very useful in this particular context due to its high genericity. The HTML DTD does not impose any real constraints on documents' logical structure. The query template mechanism, though expressive, is less expressive than other query languages. Thus, in order to provide the user with a choice of declarative access mechanisms, queries can also be formulated using two other languages. These languages have also been described, and their expressiveness has been compared.

Navigational access is eased as follows: metainformation on the documents can be exploited for hyperlink rendition. Database technology is advantageous to speed up conversion, and to ensure consistency of conversion outputs that have been materialized. The approach would not work well with HTML because, with previous versions of HTML, no mechanisms are provided to model metainformation, and with HTML 3.2 it cannot be ensured that metainformation is modeled consistently. It is left to the author of the document which metainformation is provided, and how he provides it.

With the description of documents' database-internal representation, we have covered several original features. With our hybrid representation of documents, elements can or cannot be represented by individual database objects. The actual representation is subject to configuration. The initial configuration is specified by means of an SGML document. To recognize the logical structure of physically unstructured document components, it is not necessary to use the DTD at this stage. We have an object-oriented modeling of structured documents, together with methods that are the basis for declarative access. Elements have a (logical) OID, even though they do not have to be represented by an individual database object. With method implementation, it has been reflected that, with OODBMSs, method invocations are expensive.

In the future, we wish to evaluate the behavior of a real user community. We will examine the expressive power of query templates and see whether new primitives are needed. In previous work, we examined how to reflect the semantics of HyTime architectural forms in the database and devised with an implementation [BA94, BAK95]. The HyTime link model is more sophisticated than the one that has been considered so far. Arbitrary document portions can be referenced, independent of the structure that has been made explicit with SGML markup. It may be worthwhile to extend the reflections with regard to link conversion to the HyTime model.

References

[ACM93] Abiteboul S, Cluet S, Milo T (1993) Querying and updating the file. In: Agrawal R, Baker S, Bell D (eds.) Proceedings of the International Conference on Very Large Data Bases, VLDB Endowment, Dublin, Ireland, pp 73–84

[ACM95] Abiteboul S, Cluet S, Milo T (1995) A database interface for file update, In: Michael J. Carey, Donovan A. Schneider (eds.) Proceedings ACM SIGMOD, ACM Press, New York, pp 386–397

[AKM95] Andrews K, Kappe F, Maurer H (1995) The Hyper-G Network Information System, J UCS 1(4):206–220

[B+94] Blake GE, et al (1994) Text / Relational database management systems: harmonizing SQL and SGML, In: Witold Litwin, Tore Risch (eds.) Proceedings of the First International Conference on Applications of Databases, Lecture Notes in Computer Science. Springer, Berlin Heidelberg New York, pp 313–324

[Boeh97] Böhm K (1997) Using object-oriented database technology for structured document storage (in German). PhD thesis, Technical University of Darmstadt

[BA94] Böhm K, Aberer K (1994) Storing HyTime documents in an object-oriented database. In: Adam NR, Bhargava B, Yesha Y (eds) Proceedings of the Third International Conference on Information and Knowledge Management. ACM Press, New York, pp 26–33

[BAK95] Böhm K, Aberer K, Klas W: Building a configurable database application for structured documents. Multimedia - Tools and Applications

[C+94] Christophides V, et al (1994) From structured documents to novel query facilities. In: Richard T. Snodgrass, Marianne Winslett (eds.) Proceedings ACM SIGMOD, ACM Press, New York

[Cat94] Cattell RGG (ed) (1994) The Object Database Standard: ODMG-93. Morgan Kaufmann, san MAteo, Calif.

[CCH92] Callan JP, Croft WB, Hardig SM (1992) The INQUERY Retrieval System, In: A. Min Tjoa, Isidro Ramos (eds.) Proceedings of the Third International Conference on Database and Expert Systems Application. Springer, Berlin New York Heidelberg, pp 78–83

[Con87] Conklin J (1987) Hypertext: an introduction and survey, IEEE Comput Mag:17–41

[Fre] Freese ED, The Transformation of SGML Documents for Presentation on the World Wide Web. http://www.sil.org/sgml/freese.html

[HHM94] Haake A, Hüser C, Möhr W (1994) Milestone M1; BERKOM Project:CLIP-ING; Workpackage 2: System Architecture, GMD-IPSI

[HTM] HyperText Markup Language (HTML). Available under "http://www.w3.org/pub/WWW/MarkUp/"

[ISO86] Information Technology – Text and Office Systems – Standardized Generalized Markup Language (SGML) (1986)

[ISO96] Document Style Semantics and Specification Language (DSSSL) (1996)

[KSS95] Kashyap V, Shah K, Sheth A (1995) Multimedia Database Systems: Issues and Research Directions. (Chapter Metadata for building the MultiMedia Patch Quilt). Springer, Berlin New York Heidelberg

[Mac91] Macleod IA (1991) A query language for retrieving information from hierarchic text structures, Comput J 34(3):254–264

[MMM96] Alberto O, Mendelzon GA, Mihaila G, Milo T (1997) Querying the World Wide Web. International Journal on Digital Libraries 1(1):54–67

[NBY95] Navarro G, Baeza-Yates R (1995) A language for queries on structure and contents of textual databases, In: Edward A. Fox, Peter Ingwersen, Raya Fidel (eds.) Proceedings of 18th ACM Conference on Research and Development in Information Retrieval (SIGIR'95), Seattle, Wa, pp 93–101

[O+95] Özsu MT, et al (1995) An object-oriented multimedia database system for a news-on-demand application, Multimedia Syst 3:182–203

[QRS+95] Quass D, Rajaraman A, Sagiv Y, Ullman JD, Widom J (1995) Querying semistructured heterogeneous information. In: Tok Wang Ling, Alberto O. Mendelzon, Laurent Vieille (eds.) Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases, Singapore, pp 319–344. Springer Verlag Berlin Heidelberg

[RMW94] Röscheisen M, Mogensen C, Winograd T (1994) Shared Web annotations as a platform for third-party value-added information providers: architecture, protocols, and usage examples. Technical Report STAN-CS-TR-97-1582, Stanford University

[S+94] Süllow K, et al (1994) MultiMedia Forum - an Interactive Online Journal. In: Hüser C, Möhr W, Quint V (eds) Proceedings of Conference on Electronic Publishing. Wiley, New York, pp 413–422

[SDAMK95] Sacks-Davis R, Arnold-Moore T, Kent A (1995) A standards based approach to combining information retrieval and database functionality. Int J Inf Techn World Sci 1(1):1–16

[SM93] Salton G, McGill M.J (1983) Introduction to Modern Information Retrieval. First edition. McGraw-Hill, New York

[ST94] Salminen A, Tompa FW (1994) PAT expressions: an algebra for text search. Acta Linguist Hung 41(1):277–306

[Sto96] Stonebraker M (1996) Object-relational DBMSs, Morgan Kaufmann, San Mateo, Calif.

[TEI94] Guidelines for Electronic Text Encoding and Interchange. 1994.
http://etext.lib.virginia.edu/TEI.html

[VAB96] Volz M, Aberer K, Böhm K (1996) Applying a Flexible OODBMS-IRS-Coupling to Structured Document Handling, In: Proceedings of the 12th International Conference on Data Engineering, New Orleans, pp 10–19,

[VML95] VODAK V 4.0 User Manual (1995) Technical Report 910, GMD-IPSI, St. Augustin, Germany

[YA94] Yan TW, Annevelink J (1994) Integrating a structured text retrieval system with an object-oriented database system, In: Jorge B. Bocca, Matthias Jarke, Carlo Zaniolo (eds.) Proceedings of the International Conference on Very Large Data Bases. VLDB Endowment, Santiago, Chile, pp 740–749