Multimedia Tools and Applications, 5, 275–300 (1997)

Building a Hybrid Database Application for Structured Documents

KLEMENS BÖHM AND KARL ABERER {kboehm, aberer}@darmstadt.gmd.de
GMD-IPSI, Dolivostraße 15, 64293 Darmstadt, Germany

WOLFGANG KLAS {klas@dasy.informatik.uni-ulm.de
Universität Ulm, Fakultät für Informatik, 89069 Ulm, Germany

Received November 25, 1995; Revised March 30, 1996

Editor: ??

Abstract. In this article, we propose a database-internal representation for SGML-/HyTime-documents based on object-oriented database technology with the following features: documents of arbitrary type can be administered. The semantics of architectural forms is reflected by means of methods that are part of the database schema and by the database-internal representation of HyTime-specific characteristics. The framework includes mechanisms to ensure conformance of documents to the HyTime standard. Measures for improved performance of HyTime operations are also described. The database-internal representation of documents is a hybrid between a completely structured and a flat representation. Namely, the structured representation is better to support the HyTime semantics, and modifications of document components. On the other hand, most operations are faster for the flat representation, as will be shown.

Keywords: SGML HyTime object-orientation database

1. Introduction

Administering large document collections and large documents has become a more and more important issue. Not only conventional documents, but also hypermedia documents must be handled. With SGML [10] and HyTime [11], that are standards for document exchange, documents are not in layout format, but in a format where their logical structure has been made explicit. Another advantage of formalisms such as SGML is that metainformation of arbitrary kind may seamlessly be integrated into the document. The structure of such a document is specified by means of so-called *document-type definitions (DTDs)*, i.e., a grammar that the instances of the respective document type must conform to. HyTime is an extension of SGML in that it provides a list of type definitions for document components with a fixed semantics for hyperlinking, referencing etc. that can be included in such DTDs in a very flexible way. HyTime documents can then be exchanged between different platforms, as content or attribute values of HyTime document components are interpreted in a uniform way.

The objective of our work is to design and implement a framework for structured-document storage. The semantics of document components shall be supported by the storage system. One reason is that homogeneous declarative access mechanisms including both primitives for querying SGML structures and primitives reflecting the specific semantics of HyTime constructs can be provided. Another reason is that considerable per-

formance improvements can be achieved. Furthermore, we wish to facilitate modifications of documents that are already contained in the database. With regard to the database-internal representation of documents, we see two alternatives. The first approach is to have a completely structured database-internal representation, i.e., each logical document component corresponds to a database object [1, 4, 13, 8]. The advantage is that document modifications with arbitrary granularity are facilitated, and that the HyTime-semantics can be exploited to refine the modeling in order to support efficient implementation of certain HyTime-specific operations. The disadvantage is that most elementary operations are expensive, in particular, inserting entire documents into the database and retrieving them. The alternative is to store documents as BLOBs in the database. While achieving a better performance for elementary operations, this approach falls short with regard to the other issues where a completely structured representation is suitable. To avoid the disadvantages of the two extremes, we pursue a hybrid approach where, casually spoken, only "big" elements, so-called *non-flat elements*, are represented by individual database objects, while other database objects comprise more than one element, so-called *flat elements*. The operational SGML- and HyTime-semantics can be modeled in both cases. - The most appropriate database-internal representation depends on the access pattern. A generic system, i.e., one for administering documents of arbitrary type, should provide some flexibility to take this into account. Consequently, the database-internal representation of documents has to be specified. To this end, information on the semantics of documents and document components is communicated to the database application. In other words, our system is configurable. Configuration is on the type level.

The contributions of this article are the following ones:

- **Describing documents' database-internal representation.** The database-internal representation of elements and element types has to be transparent, i.e., it must not be visible from outside whether they are flat or not. We point out how modeling primitives can be used in this context, and why extensible or even freely definable modeling primitives are of advantage. Finally, index structures are mandatory to allow for an efficient evaluation of queries over large document collections. It is not obvious what index structures for structured documents should look like.

- **Modifying documents.** When modifying components of documents that are already contained in the database with arbitrary granularity, there are the following problems: it has to be ensured that the new version of the document component conforms to the document-type definition, and the respective database-internal structures have to be generated. When a new element is inserted or deleted, it has to be checked if insertion or deletion at the respective position is feasible, according to the DTD. For our solution to be deployed in real application scenarios, it is a natural requirement that an off-the-shelf SGML editing tool can be used to carry out the modifications. The SGML editors we are aware of allow to load a separate copy of the entire document only. We describe how this apparent contradiction is solved with our approach. - In [2], a database interface for file update is described. These techniques cannot be applied here, because it is not an API for document modification that is desired, but editing functionality.

- **Reflecting the HyTime semantics.** With a system allowing to administer HyTime documents of arbitrary type, it is a complex task to ensure documents' conformance to the HyTime standard, which is a prerequisite for carrying out operations reflecting the HyTime semantics. Next to explaining how the HyTime semantics is reflected in the database, we describe measures to improve performance of HyTime-specific functionality. In [12, 6], work regarding a HyTime engine based on OODBMS technology is described. While in [12] each element is represented by an individual database object, the need for "compression" is acknowledged in [6]. In contrast to ours, that work is concerned with one individual document type. They do not explain how their compression technique can be applied to other HyTime document types, if functionality has been given up for compressed document components, and how compression affects performance of their system. We for our part argue that the HyTime-specific representation of documents must be configurable to achieve a good performance of HyTime-specific operations in many cases. Another OODBMS application for structured-document storage is described in [14]. The system is tailored for a news-on-demand application scenario. Again, only one specific document type is supported, and modifying documents in the database is not described.

The remainder of this article has the following structure: In the following section, the underlying standards are briefly reviewed. In Section 3, the database-internal representation of structured documents is described. Section 4 contains a description of the mechanisms for document modification. Section 5 describes how our modeling has been extended in order to reflect the HyTime semantics. Some experiments described in Section 6 indicate that our approach is useful. Section 7 concludes the paper.

2. Standards for Document Modeling

2.1. SGML

In the context of SGML, the differentiation between documents' logical structure and their layout structure is fundamental. With SGML, logical document components, so-called *elements*, are made explicit by means of markup, as in Figure 1. Documents have a hierarchical structure, the *primary structure*, as depicted in Figure 2.

The arrangement of elements within an SGML document of a certain type is specified by a so-called *document-type definition (DTD)*, as in Figure 3. In essence, a DTD is a grammar. Lines starting with `` `<!ELEMENT' `` are element-type definitions. `` `<!ELEMENT BODY ...>' `` is the definition of element type `BODY`. The regular expression

```
(STATEMNT, (RECOMMND|DECISION|STATEMNT)*)
```

the so-called *content model*, specifies that an element of type `BODY` consists of one of type `STATEMNT`, followed by an arbitrary number of recommendations, decisions or statements. In a nutshell, we say that an element type is directly contained in another one if it occurs in the content model of that element type. The transitive closure of the directly-contained-in relationship is referred to as *contained-in relationship between element types*. `` `+(KEYWORD)' `` is an inclusion model. It specifies that a `KEYWORD`-element may occur anywhere in a subtree whose root is a `BODY`-element. Exclusion models to forbid

```
<PROTOCOL>
  <PARTICIP><PERSON id=jh>Jörg</PERSON>
  <PERSON id=ae>Angelika</PERSON>
  <PERSON id=kb>Klemens</PERSON></PARTICIP>
  <BODY><STATEMNT persons="jh kb"><PARA>The different versions of
  the database schema have to be integrated asap.</PARA>
  <PARA>Jörg cannot carry out the <KEYWORD>integration</KEYWORD>
  due to other obligations</PARA></STATEMNT>
  <RECOMMND persons ="ae"><PARA>Angelika can take over some of
  Jörg's assignments.</PARA> </RECOMMND> ...</BODY>
</PROTOCOL>
```

Figure 1. Sample SGML document of type `Protocol'

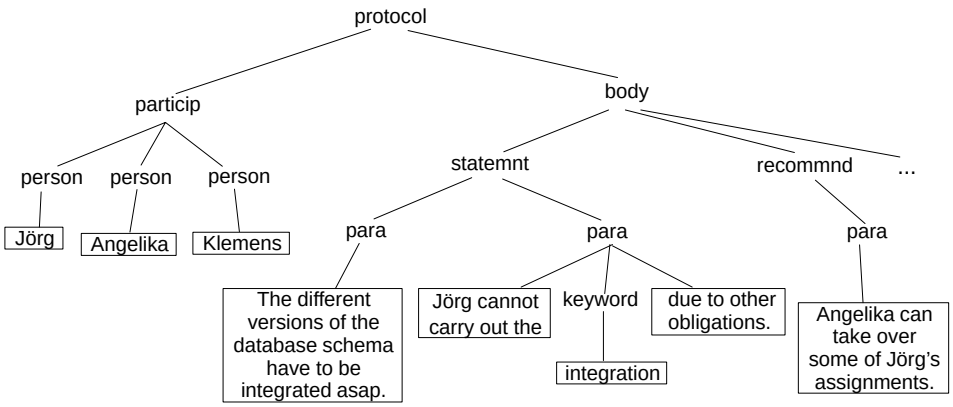

Figure 2. Structure of Sample SGML Document

```
<!DOCTYPE PROTOCOL [
<!ELEMENT PROTOCOL  (PARTICIP, BODY)>
<!ELEMENT PARTICIP  (PERSON+)>
<!ELEMENT PERSON    (#PCDATA)>
<!ATTLIST PERSON    id  ID    #REQUIRED>
<!ELEMENT BODY      (STATEMNT, (RECOMMND|DECISION|STATEMNT)*)
                        +(KEYWORD)>
<!ELEMENT (RECOMMND|DECISION|STATEMNT)  (#PCDATA)>
<-- `RECOMMND' is abbreviation for `recommendation' -->
<!ATTLIST (RECOMMND|STATEMNT)  persons  IDREFS   #IMPLIED>
...]>
```

Figure 3. Sample DTD (Document Type `Protocol')

inclusion models in smaller subtrees are also available. #PCDATA is a terminal element type comparable to data type STRING. Furthermore, elements may be furnished with attributes. `<!ATTLIST PERSON id ID #REQUIRED>' defines that there is an attribute id of type ID for elements of type PERSON whose category is `required', i.e., each element of type PERSON must have that attribute. Attributes of type ID and IDREF, IDREFS allow for arbitrary references between elements, the so-called *secondary structure*. An ID-attribute is a unique identifier of an element, while attributes of types IDREF and IDREFS are references to such elements. The fist one is a reference to exactly one ID, whereas the number of references with IDREFS is arbitrary.

Using SGML, metainformation can be included in documents in a natural way at the corresponding position within the document, be it by introducing new element types, be it by means of attributes. Declarative access based on the logical structure of documents and on metainformation is feasible.

2.2. HyTime

The semantics of element types and attribute types is not part of an SGML DTD. If elements of a certain type shall be processed in a particular way, this must be implemented within an application, i.e., a system on top of the storage system. In consequence, hypermedia documents, whose structure has been made explicit by SGML, cannot be handled in an adequate way by an SGML database application only. The rationale behind HyTime [11, 15] is to provide standardized mechanisms to describe the processing of hypermedia documents. The HyTime standard contains a set of element-type definitions together with a description of their semantics, so-called *architectural forms* or *abstract element types*. As an example, consider the HyTime architectural form dataloc in Figure 4, together with a definition of a concrete element type, i.e., one derived from that abstract one, in Figure 5 and a portion of a sample document in Figure 6. With dataloc, arbitrary pieces of data can be identified, independent of the logical structure of documents that has been made explicit using SGML markup. The content of the referenc-element in Figure 6, whose element type is derived from dataloc, has to be interpreted as follows: the target data

```
<!element dataloc    ...     (dimlist*)> //dimlist is another ar-
                                        //chitectural form that is not
                                        //explained here
<!attlist dataloc    HyTime  NAME  dataloc
                     id      ID
                     quantum (str|norm|word|name|sint|date|time|
                                 utc)   str
                     locsrc  IDREFS    ...>
```

Figure 4. Architectural Form `dataloc`

```
<!ELEMENT    referenc     (targets*)>
<!ATTLIST    referenc     HyTime     NAME     #FIXED     "dataloc"
                          id         ID
                          quantum    (str|word|name|sint|date|
                                         time|utc)     str
                          locsrc     IDREF
                          author     #PCDATA>
```

Figure 5. Element-type definition derived from HyTime architectural form `dataloc`

item starts with the 11th unit and has an extension of two units. Value `word` of attribute `quantum` specifies that these units are words.

An element-type definition may differ from the architectural form it is derived from, i.e., the concrete type may differ from the abstract one in the following ways:

- The concrete element type may have a different name than the abstract one. The connection between concrete and abstract element type is established by means of attribute `HyTime`. Its value is the name of the respective architectural form.
- The content models of concrete and abstract element types may be different. However, the content of an instance of these types must conform to both content models.
- The range of an attribute in the concrete type definition may be a subset of the one it is derived from, e.g., `IDREFS` may be replaced with `IDREF`.
- The concrete element-type definition may contain additional attributes, as compared to the original DTD.

The advantage of using HyTime architectural forms is that relevant aspects of the corresponding elements' semantics are standardized. If HyTime documents are administered by a database system, the HyTime semantics can be exploited for efficient physical data management. With `dataloc`-elements, for example, a materialized view on the pieces of data referenced is conceivable. However, it depends on the application semantics whether such a materialized view is advantageous

```
<annotation linkend=a1 nr=1>
  <content id=c1> We assume that these objects have been chosen
     because there have been complaints ...</content>
  <referenc id=r1 quantum=word locsrc=d7>11 2</referenc>
</annotation>
```

Figure 6. Instances of HyTime architectural forms

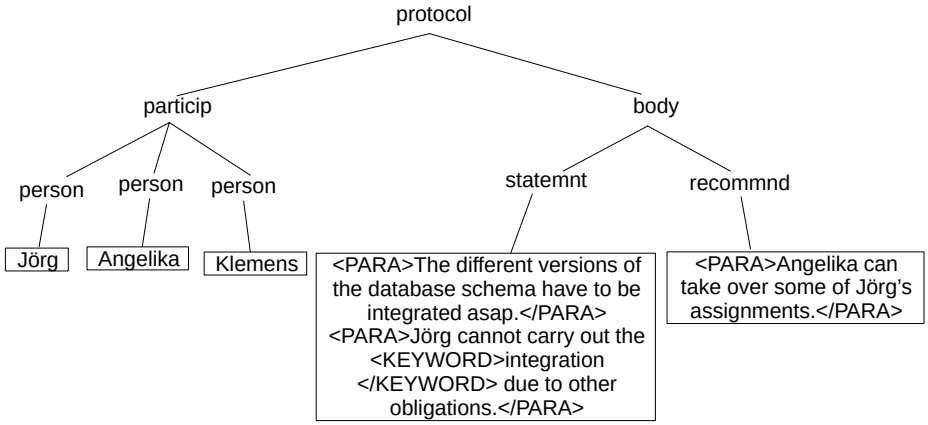

Figure 7. Sample Database Structure

3. An Efficient Database-internal Representation of Structured Documents

With our approach described in [1, 4], every element is represented by an individual database object, and there is a database class, a so-called *element-type class*, corresponding to each element type. However, the duration of read-access operations as well as of document insertion into the database depends to a high degree on the number of database objects that are accessed or generated. This is an argument in favor of a completely unstructured database-internal representation of documents. On the other hand, with an object-oriented DBMS, database objects are the physical units of modification. It depends on the element type's semantics whether its instances are good logical units of modification within a document. Our conclusion is that not all elements should be represented by individual database objects. With regard to elements that are not represented by individual objects, we say that *such elements are flat*, or *the element types are flat*, as elements of the same type shall be represented in the same way within the database. Taking the examples from Figures 1 and 2, suppose that `PARA` is a flat element type. Then, the structure depicted in Figure 7 is generated in the database.

The specification which element types are flat and which ones are not takes place when the document-type definition is inserted into the database, before any documents of that type are inserted into the database [1, 5]. - In the remainder of this section, we will address how the

physical representation of elements and element types is made transparent, which further measures are necessary to insert documents into the database in an efficient way, and which (non-existing) OODBMS features would have been useful in this particular context. We will also point out the role of extensible modeling primitives for this application scenario.

3.1. Making Document Elements' Physical Representation Transparent

In this context, it is important that the logical structure of flat document portions shall be visible. E.g., it should be possible to navigate within flat objects and to formulate queries referring to the logical structure made up by such elements. Thus, `flat' is not the same as `unstructured' or `not interpretable'. Further, the requirement that the interface of operations such as navigation operations be independent of the representation in the database is natural. With a 1:1 mapping of element types to database objects, as practiced previously, elements in the database are identified by means of the database object identifier (database-OID). Navigating within the document is easy, e.g., identifying the element that follows a particular one.

On the contrary, to identify elements within a flat document portion, the database-OID alone is insufficient. Therefore the identification scheme for document components is extended. An element is identified by an instance of type

```
[obj: OID, offset: INT]
```

The OID identifies the database object containing the document element. The offset identifies the position in the flat representation, i.e., within the bytestring, where the document element starts. The default value -1 is used if the element corresponds to the whole database object, i.e., if the element is not flat. For example, with the structure from Figure 7, the first `PARA`-element within the `STATEMENT`-element would be identified by [obj: o1, offset: 0], the second `PARA`-element by [obj: o1, offset: 86], assuming that o1 is the respective OID.

3.2. Making Element Types' Physical Representation Transparent

An important objective is to have both flat elements and element types available for navigation and search when using that extended identification scheme. At the same time, the physical representation shall not be visible to the user. To this end, an extension of the model where element types are mapped 1:1 to database classes is needed. Element types can be categorized in flat ones and non-flat ones. An element type has characteristics that are identical for both flat and non-flat element types, such as an element-type name, while other features are specific for the two categories. For instance, the implementation of a method that identifies all instances of the type is different for flat and non-flat types. Using data-modeling terminology, element types have a so-called *generalization aspect* and a so-called *specialization aspect*. With the OODBMS VODAK, modeling primitives for semantic relationships can freely be defined [17]. Category specialization is modeled so that the generalization aspect is represented by a database object, the *generalization instance*, the specialization object is represented by another database object, the *specialization instance*. All generalization instances are contained in the same class, the so-called

generalization class, but there are several *specialization classes* according to the different categories (see [17] for further details). Figure 8 contains an overview of the modeling. We assume that the database is populated with one `protocol`-document. Classes are represented by ellipses. *Metaclasses* are classes whose instances are themselves classes, such as TERMINAL or NONTERMINAL. Both dots and rectangles with textual content correspond to physical database objects. Arrows and lines have the following meaning. Straight-lined arrows indicate that an object is an instance of the respective class. Straight lines reflect the hierarchical structure of documents, i.e., the structure that is explicit within the database. The instances of FLAT are the set of all flat document portions. ElementType is the generalization class. Each element type is an instance of ElementType. We say that the instances of ElementType are *virtual element-type classes*. The crucial point with this particular modeling is that the specialization instances are not plain database objects in all cases. Non-flat element types are represented by database classes, so-called *element-type classes*, that are an instance of class NONTERMINAL. A flat element type is represented by an instance of class FlatClass. FLAT being a special specialization instance is an instance of TERMINAL. The rectangles that do not correspond to database objects illustrate where properties and methods of the respective objects are defined. CS and CATSPEC are different modeling primitives for category specialization. For example, `ET_INSTTYPE' in the box corresponding to instances of ElementType indicates that these instances have properties and methods that are defined by class ElementType. `CS_INSTINSTTYPE' reflects that they have properties and methods provided by the modeling primitive CS. In [4] it has been described that element types also take part in another specialization relationship that is modeled using CATSPEC. Hence, NONTERMINAL and its instances have properties and methods provided by two modeling primitives.

The situation gives rise to two arguments why modeling primitives should be extensible or even freely definable.

- The instances of the modeling do not have to be plain database objects, they can also be other entities. With CS, the specialization instances may be classes.

- Objects may take part in more than one semantic relationship. The extensibility of modeling primitives is necessary to take this into account.

With regard to search in the document base, one can distinguish between *search on the type level* and *search on the instance level*. In search expressions of the second kind, concrete elements may occur, as opposed to the first one. Search operations on the type level are realized as methods of the virtual element-type class. The objective of the modeling of element types described above is to encapsulate the database-internal representation of element types. When invoking such an operation, it is not necessary to know whether the element type is flat or not. By means of the modeling primitives, a method invocation to the generalization object is forwarded to the specialization object, i.e., either to the element-type class, when it exists, or the instance of FlatClass. In Figure 8 getElements and getElementsByAttribute are search operations on the type level. This is expressed by means of the grey arrows directed towards virtual element type classes. getFirst and getAll, however, are search operations on the instance level. Accordingly, the grey arrows are directed towards virtual database objects that represent individual elements.

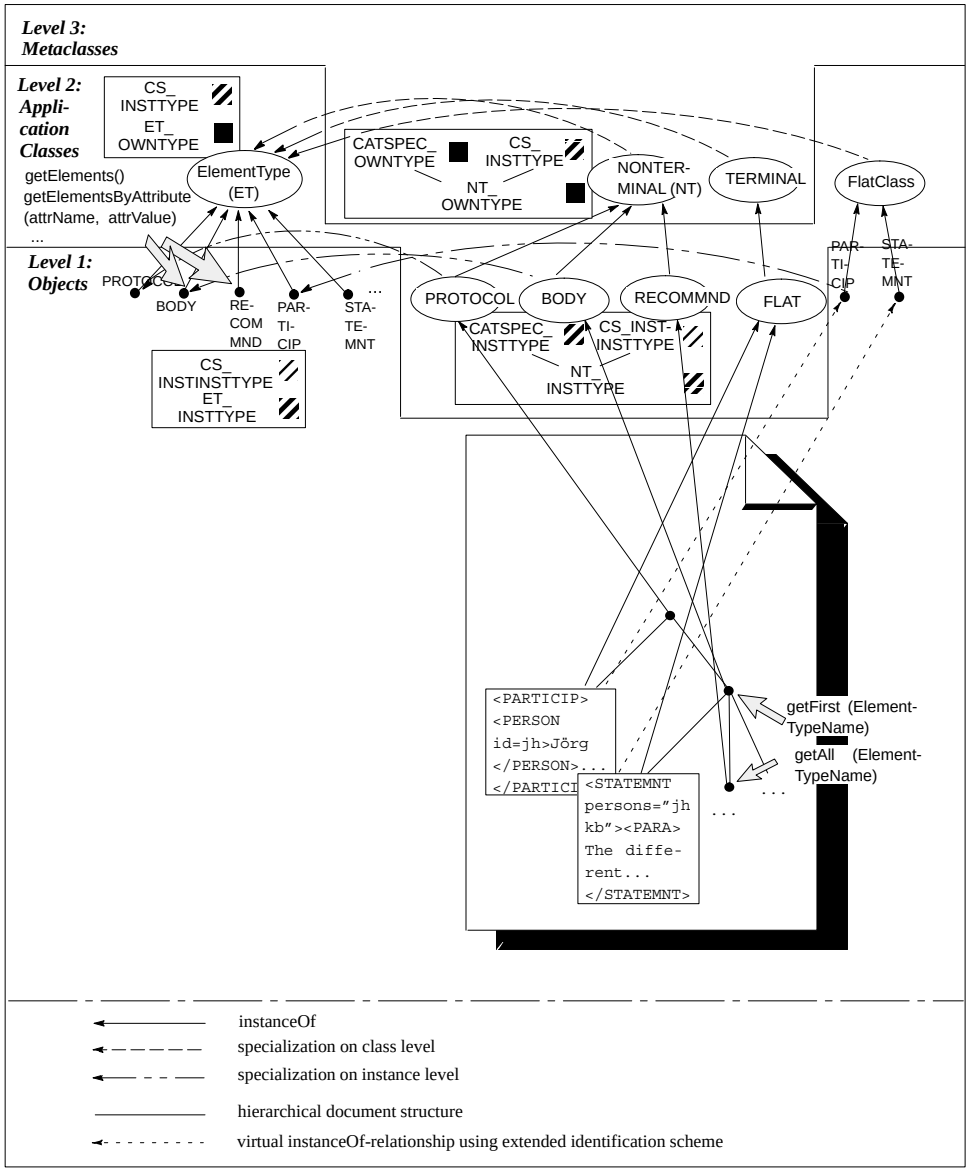

Figure 8. Modeling Overview

An instance of FlatClass representing a flat element type may contain physical references to the instances of FLAT containing elements of the respective type. These references, however, are not identical with the extended identifiers of these elements. Offsets of extended identifiers within references to flat elements are not stored explicitly, but always computed on-the-fly. The advantage is that updates are relatively easy. If an element within a FLAT-object was modified, a possibly large number of references to other elements within the same FLAT-object would have to be updated with explicitly stored references.

3.3. Configuring the Document Parser

In order to insert a document into the database, it is parsed. The parser generates a sequence of database operations which, in turn, create the database objects representing the document. These operations are part of the database schema. Ideally, the database-internal representation of the instances of an element type should be encapsulated. This, however, has a considerable impact on performance if a document as a whole is inserted into the database. Consider the case that an element is inserted. If it was not known whether the element type was flat or not, an insert operation of the database schema would have to be invoked for the element in any case. However, if the document parser creating the database representation is furnished with such knowledge, inserting documents can be organized in a more efficient way. To continue the example, with several consecutive flat elements, a string-concatenation operation for each element without immediate database access is essentially sufficient. Thus, before a document is inserted into the database, the parser retrieves the configuration specification for the respective document type from the database. The configuration specification is contained in the database in compressed form so that this step is not time-consuming and can be carried out within one database-read operation.

3.4. Database Support for the Extended Identification Scheme

With the existing database mechanisms, formulating queries or sequences of method invocations is difficult if one wants to refer to logical units that are identified not only by means of an OID, but also an additional value, in this case the offset. For illustrative purposes, consider the method getNext (offset: INT): [obj: OID, offset: INT]. The target object and the parameter identify the element. The method identifies the next element within the document. A sequence of method invocations would be as follows:

```
VAR e1, e2, e3: [obj: OID, offset: INT];
...
e2 := e1.obj → getNext (e1.offset);
e3 := e2.obj → getNext (e2.offset);
...
```

Preferably, one would want to do without explicitly identifying the records' components. Furthermore, intermediate variables such as e2 might not have to be introduced. I.e., a sequence of commands of the following kind is preferable:

```
e3 := e1 → getNext() → getNext();
```

In the context of declarative queries, the problem with intermediate variables is more serious. Namely, the algebra-expression corresponding to the query becomes much bigger by introducing additional variables. Furthermore, a base set has to be specified for each variable in a query. There is an arbitrary number of examples that this is not always feasible for such variables that are superfluous on the logical level. - The VODAK query processor has been extended so that methods can be sent to records whose first component is a database OID. If, within a query, a method is sent to such a record, it is actually sent to the database-OID, and the offset being contained in the record becomes the first parameter of the method. E.g., the expressions `e1.obj → getNext (e1.offset)' and `e1 → getNext ()' are identical within a query.

3.5. Indexing Documents

Indexing the data and using materialized views are a usual approach to accelerate query evaluation by orders of magnitude. Our index structures support elementary declarative access patterns that occur in real applications. There are the following index structures and materializations:

- index for content-oriented search,
- index for regular-expression search,
- structure index,
- attribute index,
- ID-/IDREF-tables,
- materializations of HyTime-specific views,
- materializations of layouted versions of the documents.

These index structures are now described in more detail, except for materializations of HyTime-specific views that are discussed in Section 5 and materializations of layouted versions of the documents that are not directly related to query evaluation and are described elsewhere [5].

3.5.1. Content-Based Indexing In [18], a coupling of the OODBMS VODAK with the information-retrieval system INQUERY [7] has been described, and it has been pointed out that content-oriented search or information retrieval (IR) is different from search on the syntactic level, e.g., pattern matching. With the coupling, information-retrieval functionality is made available for database content. The coupling has been integrated into this database application for structured documents. Document components for which IR functionality shall be available are redundantly administered by the information-retrieval system (IRS).

I.e., index structures are within the IRS. - While deploying a loose OODBMS-IRS coupling to provide IR-functionality for structured documents is adequate, it would be out of proportion to use such external technology for the realization of the other kinds of index structures. In the following, we describe which other index structures are meaningful for structured documents, together with some details regarding implementation.

Full expressiveness of object-oriented query languages shall be achieved independent from the fact that document components have a flat or non-flat representation. Index entries do not contain the entire logical OID of the respective element, but only the database-OID. The semantics of index entries is slightly different from the conventional one. Conventionally, an index entry contains an exact reference to the corresponding entity in the database. Here, a database object containing at least one relevant entity is referenced. The positions of the entities must be computed on-the-fly.

3.5.2. Direct Indexing Experience shows that, quite frequently, one wants to find all elements of a particular type with a particular string or regular expression in their content. The so-called *direct index* allows for a more efficient evaluation of such queries. Instead of the elements within the documents, only table entries have to be inspected. Admittedly, this structure rather is a materialized view than an index in the conventional sense of the word. It depends on the access pattern whether direct indexing is advantageous for a particular element type; it does not make sense to index all element types. The direct index of an element type can be turned on or off by means of methods sent to the respective instance of class ElementType. While this index is particularly well-suited for elements whose internal logical structure is not important, the structure index to be described next is useful to evaluate queries that explicitly aim at the logical structure of elements.

3.5.3. Structure Indexing Direct indexing is to identify elements by means of their content. On the other hand, in queries on structured documents, elements can be selected by means of structural characteristics, e.g., “Select all elements of type `SURNAME` that are contained in one of type `AUTHOR`.”. There are two element types occurring in the query: the instance of one type, the *internal element type of the query*, shall be contained in an instance of the second type, the *external element type of the query*. Element type `KEYWORD` is the internal element type of the sample query, while `STATEMNT` is the external one. It may be advantageous to have materialized views for certain containedIn-relationships, the so-called *structure index*. Administering such an index structure is fairly costly if arbitrary modifications of documents are feasible. Namely, not only modifications of the internal and the external elements have to be considered, but also elements in between, i.e., along the path, as well as elements being contained in the internal one.

3.5.4. Indexing Attribute Values Materialized views on attribute values that are kept consistent with updates are advantageous, too. With our system, multi-valued attribute types such as `NAMES` and `IDREFS` are considered: the values are inserted into the index structure one by one, i.e., the search for individual names or ID-references within an

attribute of type NAMES or IDREFS is supported. The disadvantage, however, is that the order of attribute values may have a meaning which is not supported by this index.

3.5.5. Desirable Features of OODBMSs to Realize Index Mechanisms for Structured Documents The indexing structures and mechanisms to ensure consistency described in this subsection are part of the database schema. There are several reasons why the mechanisms provided by the underlying DBMS have not been used:

- Classical index mechanisms are not suited to reflect the special semantics of SGML.
- If no element type was flat, there would be the following problem with regard to structure indexing: the internal element does not need to be directly contained in the external one, the containment-relationship may rather be over an arbitrary number of elements. With conventional path indices, indexing over several objects is essentially possible [3]. However, the number of steps and the relationships between the individual objects must be known in advance. - On the other hand, maintaining materialized views on the result of methods is arbitrarily complex in the general case. Thus, to mitigate the difficulties with the two extremes, it seems that an extended path index allowing to index over an arbitrary number of steps but the same relationship (i.e., the relationship is between objects of the same type) would be an adequate form of database support in this context.
- A problem on the technical level is that index mechanisms for built-in datatypes of the underlying DBMS are not as powerful as one might desire in this particular context. For instance, there is the VML-datatype DICTIONARY, a set of key-value pairs of arbitrary type. Suppose there is a class whose instances have a property of type DICTIONARY. One would like to have indexing mechanisms allowing to index certain keys only, but for all instances of the class. This would be useful to index attributes of non-flat elements.

These characteristics of the underlying database technology are not VODAK-specific, at least the first two. With other OODBMSs, the index mechanisms provided by the system could not have been used either.

3.6. Supporting Documents' Secondary Structure

Documents do not only have a hierarchical structure. In the context of structured documents, arbitrary relationships can be established by means of the ID-/IDREF(S) mechanism. These mechanisms allow to define *the documents' secondary structure*. Next to differentiating between primary and secondary structure, a differentiation between intra-document relationships and inter-document relationships (not described here) is feasible.

3.6.1. Intra-Document Relationships With SGML, each element can be furnished with an ID, which must be unique within the document. With attributes of type IDREF or IDREFS, elements can be referenced by means of their ID. The value of an attribute

of type IDREF must occur in the document as value of an attribute of type ID. When attribute values are updated, this constraint is checked. To support navigation according to the secondary structure or to process declarative queries based on that structure, it may be advantageous to maintain an ID table, i.e., a set of key-value pairs: An SGML-ID is a key, that must be unique, the value is the OID of the corresponding database object. With our approach, if the element is within a FLAT-object, it is again only the OID that is stored, but not the offset. The motivation is the same as before, namely not to make small modifications cumbersome. - In the previous subsection, the index structures for attributes have been described. One might wonder why there is another mechanism for ID attributes. One reason is that the attribute index is for the whole document collection. The ID mechanism, in turn, is laid out for individual documents. Another reason is that with attributes of type ID the attribute name is not relevant, as opposed to other attributes.

An ID table supports the processing of queries in one direction, such as “Select all elements referenced by particular elements.", but not queries in the opposite direction, i.e., “Select all elements referencing particular elements.". In this case, an IDREF-table is advantageous. An IDREF-table maps ID-references to database objects. - It is not mandatory to maintain such tables. Naturally, the decision depends on the concrete scenario. Some criteria are as follows: a table is of rather little use if the document is not very much fragmented in the database, i.e., most portions are flat. If documents are frequently updated, as opposed to the number of queries on the secondary structure, maintaining the tables may not be worthwhile. Finally, if there are hardly any queries with regard to the secondary structure, such a table will not be necessary.

Supporting the ID-/IDREF-mechanisms is important for an efficient implementation of the HyTime-semantics that will be described in Section 5.

4. Modifying Documents

In this section, mechanisms for modifying documents contained in the database will be described. In the following section, our approach to reflect the semantics of HyTime architectural forms in an OODBMS is discussed. Next to the fact that both modifying documents and modeling of HyTime architectural forms are facilitated by a structured database-internal representation of documents the relationship between the two issues is as follows: a prominent feature of HyTime is to provide standardized mechanisms allowing to model hyperlinks. As an example, annotations of different types can be modeled. These annotations must be part of a HyTime document. Hence, generating an annotation is an insertion of elements into the document.

With regard to document modification, the following distinction can be made: overwriting an attribute value or a (terminal) element's textual content is rather simple, because the database structure representing the document is left unchanged. Index structures have to be updated. The more general case is that elements can be inserted into the document or removed from the document. At first sight, in order to modify a document, it seems feasible to remove the document as a whole from the database, modify and then insert it into the database again. This, however, may be time-consuming with big documents. Furthermore, granularity for updates would be on the document level. In the general case, this is too

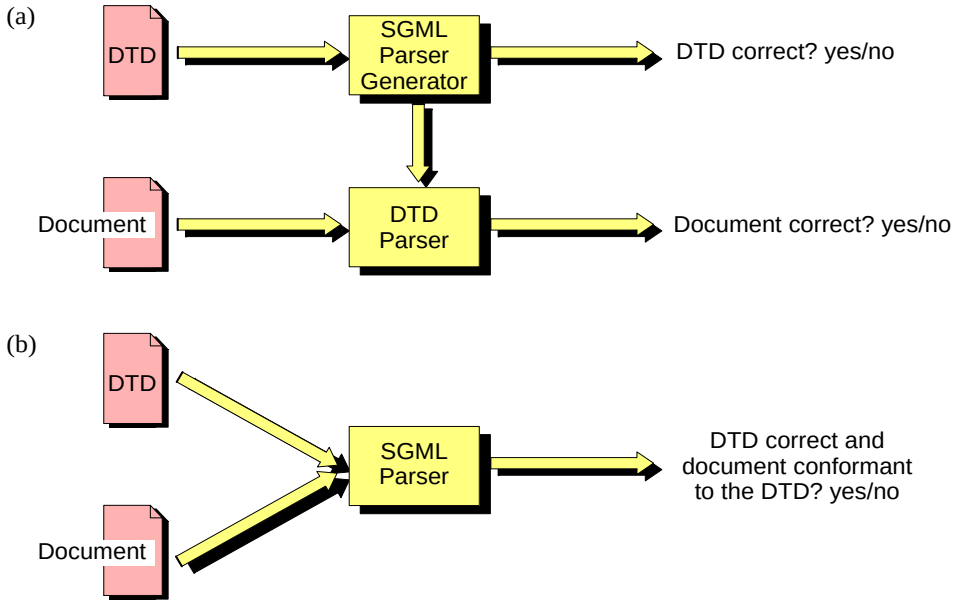

Figure 9. different approaches to parsing SGML documents

coarse. Incremental modifications of documents, particularly, insertion of elements into a document already contained in the database, leads to the following difficulties:

1. The corresponding database structure has to be generated. If a document as a whole is inserted into the database, this is controled by the SGML parser. The parser, however, can only parse entire documents.
2. It has to be ensured that insertion does not lead to a violation of the DTD, and that the restrictions of the (SGML-)standard are met.
3. Index structures must be updated. With structure index and IRS index, elements may be affected by the modification that are not contained in the subtree that has been inserted or deleted, but are above it in the hierarchical structure.

4.1. Generating the Database Structure

4.1.1. Extending the Document-Type Definition An SGML parser that controls document insertion into the database is part of our architecture. In principle, there are two different kinds of parsers for structured documents (see Figure 9). In the first case, the DTD is compiled, i.e., a parser for documents of the type is generated. In the second case, the DTD is interpreted. From another perspective, in the first case, parsing consists of two steps, namely parsing the DTD and parsing the document, in the second case, there is only one step. In the context of our work, parsing is as in Figure 9 (a). I.e., a parser generator for

```
<!DOCTYPE PROTOCOL_EXT [
<!ELEMENT PROTOCOL_EXT  (DUMMY|(PARTICIP, BODY))>
<!ELEMENT PARTICIP      (PERSON+)>
<!ELEMENT PERSON        (#PCDATA)>
<!ATTLIST PERSON        id  ID>
<!ELEMENT BODY          (STATEMNT, (RECOMMND|DECISION|STATEMNT)*)>
<!ELEMENT (RECOMMND|DECISION|STATEMNT)  (#PCDATA)>
<-- `RECOMMND' is abbreviation for `recommendation' -->
<!ATTLIST (RECOMMND|STATEMNT)  persons  IDREFS>
<!ELEMENT DUMMY         (PARTICIP|PERSON|BODY|STATEMNT|RECOMMND|
                             DECISION)>
...]>
```

Figure 10. Modified DTD

a two-step parsing process is used. With this premise, our solution to the problem addressed in Item 1 is as follows: To generate the database-structure corresponding to the element to be inserted by means of an SGML parser, the document-type definition is modified in a schematic way that does not require manual intervention. Individual elements now are admissible documents. E.g., the DTD in Figure 3 becomes the DTD in Figure 10. Thus, an arbitrary element becomes a document conformant to the DTD by enclosing it in

`<PROTOCOL_EXT><DUMMY>', `</DUMMY></PROTOCOL_EXT>'.

If, for example, the element

```
<PERSON id="rb">Ralph</PERSON>
```

is to be inserted into the document, the document

```
<PROTOCOL_EXT><DUMMY><PERSON id="rb">Ralph</PERSON></DUMMY>
</PROTOCOL_EXT>
```

is parsed. In more detail, element-type name DUMMY is predefined. Instances of DUMMY are treated differently: it ignores elements of type DUMMY when constructing the database structure, but remembers that it is within an element of type DUMMY. Instead of generating a new tree in a database, a subtree is inserted into a tree that already exists. The insertion operation is furnished with parameters specifying the position in the document where the element is to be inserted.

4.1.2. Inclusion and Exclusions With this approach, as described so far, one does not have to generate a fragment DTD on the fly. This, however, is necessary to some degree when taking into account inclusions and exclusions. Namely, because of inclusions and exclusions it is context-dependent whether elements are allowed within another one. This is illustrated by means of the following DTD-fragment.

```
<!ELEMENT A      (C)+     +(D)>
<!ELEMENT B      (C)+>
```

If an element c of type C is inserted into a document, an element d of type D is allowed within c if c is inserted into an element of type A, but not into an element of type B. To solve the problem, the inclusion models and exclusion models of all elements must be inspected where c is inserted directly or indirectly. Depending on the position where c is to be inserted the inclusion model and exclusion model for DUMMY has to be generated dynamically. - Another observation is the following one: when entire documents are inserted into the database, communication is in one direction only, namely from the parser to the database (except for fetching the configuration specification). This, however, is not sufficient if a document within the database is modified. For the position where insertion shall take place, the types of the elements containing this position are needed for parsing, as explained above.

4.2. Conformance Checking

Communication between parser and database is necessary to ensure SGML conformance. In the context of IDs it has been pointed out that components of an attribute of type IDREF(S) must occur as ID-values within the document. When an entire document is parsed, it is checked at the end of the process if there is a corresponding ID-value for all IDREF-/IDREFS-values. If an ID-reference is not valid, the parser terminates with an error message. If a fragment of a document containing IDREF-/IDREFS-attributes is inserted, it is not only this fragment that must be searched for corresponding ID-values. The part of the document already contained in the database must be considered, too. I.e., information from the database is needed to decide whether the fragment may be inserted or not.

Furthermore, it must be ensured that, according to the DTD, the element may be inserted at the specified position within the document. The SGML parser with the DUMMY-extension described before does not carry out this check, it merely ensures that the internal logical structure of the element to be inserted conforms to the DTD. With our implementation, that check is carried out by means of the DREAM parser [9], a tool for analyzing document-type definitions that has been integrated into our framework. For each nonterminal element type, our extension of the DREAM parser generates an automaton corresponding to the content model. The automata are used to check if the new sequence of elements that would be the result of the insertion process conforms to the content model of the father element. This check is carried out within the database.

5. Modeling HyTime Architectural Forms and Ensuring Conformance of Documents to the HyTime Standard

Ensuring conformance of HyTime documents to the standard is arduous due to the complexity of the standard. In the following paragraphs, the modeling is described. After that, the mechanisms for conformance checking are described.

Elements derived from HyTime architectural forms have both SGML- and HyTime-specific characteristics. For instance, two numbers in a `dimspec`-element, i.e., an element derived from HyTime architectural form `dimspec`, have to be interpreted as address, while the meaning of the same numbers within the content of elements of other types is not fixed in advance. In other words, an element in its role as HyTime element has additional characteristics. They are modeled by means of an additional database object. In order to use the VODAK modeling primitives for role specialization [17], it is required that both the generalization aspect as well as the specialization aspect are modeled by individual database objects.[1] This is depicted in Figure 11. The generalization objects are in the left half of the figure, while the specialization objects are in the right half.

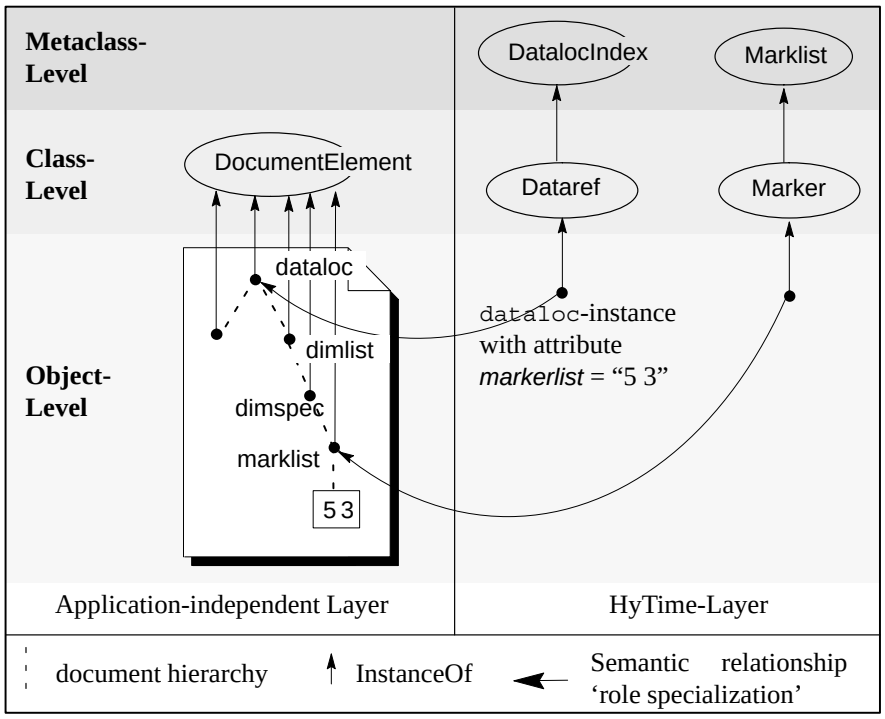

Figure 11. Modeling of a HyTime-Document

Consider an element type in a DTD being derived from a HyTime-architectural form. When the DTD is inserted into the database, a corresponding role-specialization class, a *HyTime-element-type class*, is generated. All HyTime-element-type classes that are derived from the same architectural form are instance of the same metaclass.[2] In Figure 11, Dataref and Marker are HyTime element-type classes. They correspond to element types from the application-DTD that are derived from HyTime architectural forms `dataloc` and `marklist`, respectively. `dimspec` and `dimlist` are other HyTime architectural forms. DatalocIndex and Marklist are metaclasses corresponding to HyTime architectural forms `dataloc` and `marklist`.

All information that is needed for conformance checking or for HyTime-specific processing is contained in properties of the metaclass. This includes the content model of the architectural form, and information about which of the form's attributes have to be recognized. It is not only the attributes that are known to the metaclass, but also the category of the respective attribute, e.g., `#IMPLIED`, `#REQUIRED`. An attribute's category is specific to the architectural form. An attribute `id`, for example, may be required in one architectural form, but be optional in another one.

5.0.1. HyTime Conformance Checks In the sequel, it is described how the requirements stemming from the HyTime standard are reflected in the realization of our system. Both when inserting the DTD and when inserting documents, conformance checks have to be carried out. If requirements are not fulfilled, insertion terminates with an error message. Database objects may have been generated before this point. Generation of such objects is undone by means of database transaction mechanisms. In the sequel, it is described in which phase of document- or DTD-insertion the individual conformance checks are carried out. Not all checks are explicitly mentioned, i.e., checks that are of a technical nature are omitted from the following list. For instance, the root element of a document must have an attribute `HyTime`, and it must have the value `HyDoc`. Our system issues a warning, if this requirement is not fulfilled.

1. Checks before HyTime element-type classes are generated:

   - If an attribute required according to the architectural form is not contained in the element-type definition, insertion is aborted with an error message.
   - The attributes' category is specified in the architectural form. For instance, the architectural form may specify that a certain HyTime attribute must be defined in the DTD. This is the case if the attribute definition in the architectural form has the keyword `#REQUIRED`. As another example, the architectural form may specify that an attribute value must be fixed in the DTD, i.e., be the same for all instances of the respective element: if the attribute definition in the architectural form contains keyword `#FIXED-in-DTD`, the respective attribute definition in the DTD must contain keyword `#FIXED`. The corresponding check takes place in this phase of the insertion process.
   - As pointed out before, the range of an attribute in an element-type definition derived from an architectural form must be a subset of the corresponding attribute range from the architectural form. E.g., `NAMES` may be replaced by `NAME`. - The respective check is carried out during this phase.

   It is not mandatory to compare the content models of architectural form and derived element-type definition in the DTD. Such a comparison is not part of the current version of the system.

2. Checks before objects corresponding to HyTime elements are generated:

   - The content of an element derived from an architectural form must conform both to the architectural form's content model and the corresponding content model in

the DTD. One approach to realization is that, for each architectural form's content model, there is a corresponding automaton, and the content is then compared with the respective automaton.

- It is checked if attributes that are required according to the HyTime standard are indeed instantiated within the document, and if attributes whose value is fixed in the architectural form have the correct value.

3. Checks after the creation of HyTime objects:

   - Under certain circumstances, attributes and attribute values can be omitted from the document. The database application then assigns a default value to such attributes.
   - The HyTime standard contains conditions on HyTime documents that cannot be modeled using conventional mechanisms, i.e., in particular, cannot be reflected by means of the content model. E.g., the content of instances of architectural form `dimspec` must resolve to a list of numbers. Normally, such conditions do not refer to individual elements in isolation. Rather, several elements may have to be inspected to check the condition. Hence, these conditions can only be checked in the database after having generated the HyTime objects. If an error is identified, generation of the database objects is undone by means of the database transaction mechanisms.

The current version of the database-application framework contains an implementation of those architectural forms that can be used to model annotations in a variety of ways. Considering the remaining architectural forms, with regard to implementation of the HyTime conformance checks, no major difficulties are expected because these checks can directly be taken over from the other forms' implementation. On the other hand, for some of the remaining forms, the implementation of architectural-form-specific semantics is not always obvious. In particular, this holds true if time-dependent media have to be considered.

5.1. Improving the Performance of HyTime-specific Operations

5.1.1. Reducing the Number of Database Objects In order to improve performance in this particular context, there does not exist only one, but two metaclasses for some architectural forms. The second metaclass differs from the first one in that its metainstances bear materializations of HyTime-specific views. The interfaces of their instances and metainstances are identical (with regard to the read operations). As an example, consider the HyTime architectural form `dataloc`. Instances of this architectural form reference arbitrary pieces of data. In some cases, it is advantageous if the `dataloc`-role-specialization object contains a materialized view on the pieces of data referenced. In other cases, there may be no advantages resulting from such a materialized view. The arguments are roughly the same as the ones for ID tables in the previous section. In consequence, this aspect of the elements' physical configuration has been made configurable, just as elements' flat-/non-flat characteristics. - However, it may be too undifferentiated to have exactly one materialized view. The views on the document that should be materialized depend on the application scenario, i.e., the document types and the operations and queries on the

document collection. Furthermore, with the HyTime mechanisms, views over an arbitrary number of elements are feasible. In consequence, enriching the modeling with additional view materializations is of advantage.

Performance is not always improved by means of HyTime-specific view materializations. Clearly, HyTime-specific read operations are accelerated. However, the restriction that HyTime elements must not be flat has the consequence that certain basic operations such as document insertion and retrieval of documents from the database are decelerated. Hence, the above restriction is too rigid. There should be a differentiation between the individual architectural forms, i.e., the HyTime aspect of HyTime-element types should not always be modeled by means of a role-specialization object. There are the following criteria:

- If the materialization of a view is advantageous for a HyTime-element type, it is natural to model the HyTime aspect using a role-specialization object. In the introduction, we have pointed out that representing an element by an individual database object may be advantageous to reflect the HyTime semantics. By that remark, we have referred to the materialization of HyTime-specific views, that are best modeled as a property of the respective database object.

- Some HyTime architectural forms are only used within other forms, as a rule, and, from the user's perspective, do not represent a meaningful document component, e.g., `dimspec`. Methods reflecting the semantics of such architectural forms are not invoked by the application, but within operations of the HyTime-element they are contained in. It seems feasible that those methods operate on the flat database structure. By means of interpretation it is checked that the method may be executed for the target element.

5.1.2. Conformance Checking Checking if a document fulfills the requirements of the HyTime standard consists of various individual checks, as indicated above. These checks may become fairly time-consuming, because not only the individual HyTime-element is subject to such checks. Rather, an arbitrary number of elements may be involved. Furthermore, the corresponding HyTime element-type classes and metaclasses have to be accessed to obtain the information the conformance checks are based on. - The following measures help to cut down the duration of conformance checking:

- Neither knowledge on the architectural forms that is needed for conformance checks nor information about which element types are derived from which architectural form is stored with the corresponding database object any more, but instead is cached in main memory. In analogy to the automata corresponding to the content models from the DTD, data structures may have to be generated anew each time the database server is started.

- Conformance checks are shifted from the instance level to the type level. The content model of the element-type definition is compared with the one of the architectural form it is derived from in order to find out if the extension of the element type's content model is a subset of the one of the architectural form, or if the intersection of these two extensions is empty. Such a comparison is not obligatory according to the HyTime standard, and, in the general case, such statements about two content models cannot

be made. The objective, however, is to come up with such results for a possibly large number of cases. In such a case, no checks on the instance level would be necessary, and they could then be omitted for improved performance.

Furthermore, checks on the instance level may become superfluous by analyzing the document-type definition. For instance, with regard to architectural form `dimspec`, in some cases it can be inferred from the document-type definition that its content always resolves to a list of numbers. Such checks have to be developed for each architectural form individually. Again, it is not feasible to cover all cases. The objective is to avoid checks on the instance level for a possibly large number of cases. On another level, to reduce checking on the instance level, it seems advantageous to derive recommendations to the DTD-designer that correspond to such restrictions in the standard.

- With the current version of the system, HyTime conformance checks take place in the database. The basic idea is to carry out checking outside of the database as much as possible to avoid access to database objects, in particular when inserting documents as a whole. When a document is inserted into the database, its logical structure is first recognized by an SGML parser. At first sight, it seems feasible to extend the SGML-parser so that HyTime conformance checks take place during parsing. However, while certain clean-cut extensions of the parser have been sufficient so that it controls document insertion into the database, this is not the case for HyTime conformance checking. - Alternatively, the architecture depicted in Figure 12 is feasible: the SGML parser checks whether the document conforms to the respective DTD. The database operations generating the database-internal representation of the document are not invoked by the SGML parser. If the document conforms to the SGML-DTD, another version of the document is generated. In this version of the document, no markup is omitted, and the tag delimiters have been made easily identifiable by means of special characters. Based on this version of the document, no knowledge on the grammar is necessary to recognize the logical structure of documents. The motivation is to recognize the logical document structure without knowledge on the DTD. As a second step, the HyTime checker recognizes the documents' logical structure, independent of the DTD, and carries out the HyTime conformance checks. If the document is conformant to the HyTime standard, it invokes the database operations to generate the documents' database-internal representation.

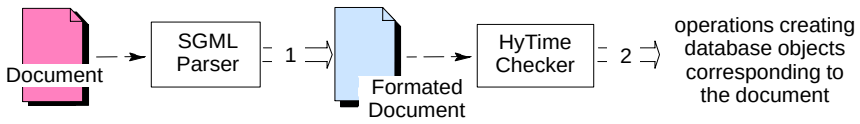

Figure 12. Architecture

6. Experimental Results

We have conducted experiments to investigate the effect of different physical representations of document components. For these experiments, MMF-documents have been used [16]. An MMF-DTD consists of 128 element types. We have run tests with four different configurations: everything flat, nothing flat, 70 out of 128 element types flat, 95 out of 128 element types flat.

Within these experiments, we have had a closer look at four elementary operations:

1. Inserting entire documents into the database.
2. Retrieving and displaying entire documents.
3. Selecting all elements of a certain type, independent of the document.
4. Navigating within the documents, e.g., traversing to the following element, the element the current element is contained in etc.

As expected, the duration of document insertion into the database strongly depends on the number of elements whose physical representation is flat. On an average, there is a factor 12 between the two extremes ("everything flat" vs. "nothing flat"). Further measures reducing the duration with the "everything-flat" configuration by another 50% are conceivable.

With the second item, the difference between the two extremes is even larger, namely there is a factor 35.

The third kind of access that has been examined is a little more differentiated. If in a document there are only one or two instances of the element type that is searched for, the "nothing-flat" representation is about 50% faster than the other extreme. In both cases there is about the same number of database objects that are accessed. With the "flat"-configuration, however, the FLAT-string has to be scanned in addition.[3] However, if there are more elements of the particular type contained in the same FLAT-object, the flat representation is advantageous, as there are less database objects to be accessed.

When navigating within documents, we have not noticed considerable differences between the flat and the non-flat representation. Non-flat is slightly faster.

Summing up, with small- to medium-sized documents, the "everything-flat" configuration is superior to the alternatives available if there is a "normal" mix of access operations. The structured representation, however, is more appropriate with regard to concurrent access to a document including modification operations and the modeling of element types' semantics, as has been explained in the previous sections. The conclusion we have drawn is that some flexibility is worthwhile to take into account the application-specific semantics, and we see our approach as a way to achieve this.

7. Conclusions

The objective of our work is to build a storage system for structured documents. Object-oriented database technology gives way to mechanisms for modifying the document collection, and allows for exploiting the SGML-/HyTime semantics for the documents' physical

representation. On the other hand, with a straightforward mapping from document components to database objects, as carried out previously [1], certain access operations are rather slow. It has turned out that, with a hybrid database-internal representation, the weaknesses of the two extremes can be avoided. The question what data structure is the most appropriate one for the documents' physical representation cannot generally be answered. This depends on the access pattern, which in turn differs for different document types and different application scenarios. A generic system should offer flexibility to support the actual access pattern in an adequate way. The basic observation is that element or attribute types should not be treated in a uniform manner. The characteristics of document components should be reflected with the database-internal representation. In other words, the database-application framework has been made configurable. One may choose between different physical representations for elements (i.e. “flat" vs. “non-flat"), several indexing structures for the document collection of elements' and attributes' contents, and diverse mechanisms supporting access according to documents' secondary structure. The effect is that, compared to our previous work, performance of basic operations is considerably improved. Further, the functionality that has been achieved with the previous prototype has not been subtracted from.

With our database-application framework, documents of arbitrary types can be administered. The initial database-internal representation of documents of a certain type is configured when the DTD is inserted into the database, before documents of that type are inserted. For performance reasons, the parser is furnished with information on the current configuration.

Acknowledgments

We thank Peter Fankhauser and Bertin Klein for their help with the DREAM-Parser.

Notes

1. In consequence, with our current implementation there is the restriction that elements with HyTime semantics must not be flat so that the modeling primitives for role specialization can be applied.
2. With VODAK, metaclasses are classes whose instances are classes themselves. It is feasible to generate (application-)classes, i.e., instances of metaclasses, at runtime. In this case, a metaclass comprises both the type definition of its instances and its metainstances.
3. In this context we point out that the sample documents consist of a lot of elements and are highly structured, but, on the other hand, are not very big. With bigger documents, the difference is expected to be larger.

References

1. Karl Aberer, Klemens Böhm, and Christoph Hüser. The Prospects of Publishing Using Advanced Database Concepts. In Christoph Hüser, Wiebke Möhr, and Vincent Quint, editors, *Proceedings of Conference on Electronic Publishing*, pages 469–480. John Wiley & Sons, Ltd., April 1994.
2. S. Abiteboul, S. Cluet, and T. Milo. A Database Interface for File Update. In *Proceedings ACM SIGMOD*, pages 386–397. ACM Press, 1995.
3. E. Bertino and C. Guglielmina. Path-index: An Approach to the Efficient Execution of Object-Oriented Queries. *Data & Knowledge Engineering*, 10(1):1–28, 1993.

4. Klemens Böhm, Karl Aberer, and Erich J. Neuhold. Administering Structured Documents in Digital Libraries. In Nabil R. Adam, Bharat Bhargava, and Yelena Yesha, editors, *Digital Libraries*, Lecture Notes in Computer Science, pages 91–117. Springer Verlag, 1995.
5. Klemens Böhm, Karl Aberer, Erich J. Neuhold, and Xiaoya Yang. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM, 1996. Accepted for publication in VLDB Journal.
6. John F. Buford. Evaluating HyTime: An Examination and Implementation Experience. In *Proceedings of Hypertext'96*, pages 105–115. ACM Press, March 1996.
7. J.P. Callan, W.B. Croft, and S.M. Hardig. The INQUERY Retrieval System. In *Proceedings of the Third International Conference on Database and Expert Systems Application*, pages 78–83. Springer Verlag, 1992.
8. P. Francois, P. Futtersack, and C. Espert. SGML/HyTime Repositories and Object Paradigms. *Electronic Publishing*, 8(2 & 3):63–79, 1995.
9. Tilman Göttke and Peter Fankhauser. DREAM 2.0 User Manual. Technical Report 660, GMD-IPSI, 1992. St. Augustin.
10. Information Technology - Text and Office Systems - Standardized Generalized Markup Language (SGML), 1986.
11. Information Technology - Hypermedia/Time-based Structuring Language (HyTime), 1992.
12. J.F. Koegel et al. HyOctane: a HyTime Engine for an MMIS. In *Proceedings of the ACM Conference on Multimedia*, pages 129–136. ACM Press, 1993.
13. Kyuchul Lee et al. Object-Oriented Modeling, Querying, and Indexing for Multi-structured Hypermedia Document Database. In *Proceedings of the International Workshop on Multimedia Database Management Systems*, August 1996.
14. M.T. Özsu et al. An Object-Oriented Multimedia Database System for a News-on-Demand Application. *Multimedia Systems*, (3):182–203, 1995.
15. S.J. De Rose and D.G. Durand. *Making Hypermedia Work*. Kluwer Academic Publishers, 1994.
16. Klaus Süllow et al. MultiMedia Forum - an Interactive Online Journal. In Christoph Hüser, Wiebke Möhr, and Vincent Quint, editors, *Proceedings of Conference on Electronic Publishing*, pages 413–422. John Wiley & Sons, Ltd., April 1994.
17. VODAK V 4.0 User Manual. Technical Report 910, GMD-IPSI, April 1995. St. Augustin.
18. M. Volz, K. Aberer, and K. Böhm. Applying a Flexible OODBMS-IRS-Coupling to Structured Document Handling. In *Proceedings of the 12th International Conference on Data Engineering*, pages 10–19, 1996. New Orleans.

Multimedia Tools and Applications, 5, 301–300 (1997)

Contributing Authors

Klemens Böhm Klemens Böhm is a researcher in the division for open adaptive information systems at GMD-IPSI in Darmstadt, Germany. His work mainly deals with structured document storage, and he is also interested in query optimization and applications of object-relational database technology. In 1993 he has earned a diploma degree in information science from the Technical University of Darmstadt, Germany.

Karl Aberer Karl Aberer received his Ph.D. in mathematics in 1991 from the ETH Zürich (Eidgenössische Technische Hochschule Zürich). In 1991 and 1992 he was postdoctoral fellow at the International Computer Science Institute (ICSI) of the University of California, Berkeley. In 1992 he joined GMD-IPSI. There he is manager of the database research division. He is lecturing courses on non-conventional and multimedia databases at the Technical University of Darmstadt. His research interests include object-oriented database systems, multimedia database systems, database interoperability and advanced database applications.

Wolfgang Klas Dr. Wolfgang Klas is Professor for Computer Science at the University of Ulm, Germany, working in the area of multimedia information systems. He was head of the Distributed Multimedia Systems Research Division (DIMSYS) at the Integrated Publication and Information Systems Institute (IPSI) of the German National Research Center for Computer Science (GMD). In 1987, Dr. Klas founded the VODAK database research project which resulted in a full-fledged open object-oriented database system which serves as the basis for the multimedia database extensions like continuous object management, multimedia playout management, synchronization management, and multimedia document management. He is lecturing courses on database systems and multimedia information systems. Dr. Klas was a visiting fellow at the International Computer Science Institute (ICSI) at the University of California at Berkeley.