The design and implementation of an infrastructure for multimedia digital libraries

Arjen P. de Vries
Centre for Telematics and Information Technology
University of Twente, The Netherlands
arjen@cs.utwente.nl

Brian Eberman
Cambridge Research Lab
Digital Equipment Coorporation
bse@crl.dec.com

David E. Kovalcin
UNIX Engineering
Digital Equipment Coorporation
kovalcin@unx.dec.com

Abstract

We develop an infrastructure for managing, indexing and serving multimedia content in digital libraries. This infrastructure follows the model of the web, and thereby is distributed in nature. We discuss the design of the Librarian, the component that manages meta data about the content. The management of meta data has been separated from the media servers that manage the content itself. Also, the extraction of the meta data is largely independent of the Librarian. We introduce our extensible data model and the daemon paradigm that are the core pieces of this architecture. We evaluate our initial implementation using a relational database. We conclude with a discussion of the lessons we learned in building this system, and proposals for improving the flexibility, reliability, and performance of the system.

Keywords: *digital libraries, multimedia databases, multimedia modeling, content-based retrieval, distributed object model.*

1. Introduction

Vast digital multimedia libraries and databases are being created now by businesses and government agencies. These libraries will change the way we interact with multimedia information both personally and professionally, and the business structures we build to deliver, store and index the information.

In order to really use multimedia information, technologies for easily browsing, summarizing and indexing the information must be built. We build an infrastructure for delivering indexed videos using the network, and fostering research in multimedia indexing. We focus on building an indexing engine, and investigate user-interface issues for delivering video to distributed clients.

In order to realize this vision, a structure for managing, indexing and serving content must be developed. We believe, that this structure must follow the model of the web and thereby be distributed in nature. This has implications on the design of the overall system and the structure of the database which is used for indexing and retrieval. In this paper, we discuss our initial prototype design of such a structure and our implementation of a database for managing the information. We conclude with a discussion of the lessons we learned in building this system, and proposals for improving the flexibility, reliability, and performance of the system.

2. Digital libraries and the Librarian

In this paper, we concentrate on the design and implementation of the Librarian. The Librarian is the digital library's component system that manages information about the raw data. Meta information in the Librarian consists of both manual annotations and automatically extracted data.

We first developed the simple demo system shown in figure 1, to enable early evaluation of user-interface and video delivery issues. The user interface of the prototype system consists of HTML pages, CGI scripts and Java programs. Users can connect with the digital library through the web. The user contacts the Librarian and issues a query. The Librarian looks up the location of video fragments matching the query, and presents some information about these fragments to the user. The user selects the video fragments of

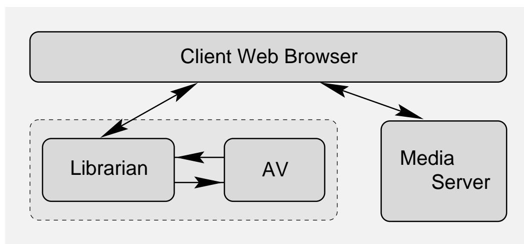

Figure 1. The demo system.

interest. Finally, the media server plays selected fragments to the client.

In the demo system, the only *meta data* we take into account are the words occurring in the closed captions of the videos. This decision makes it easy to prototype a retrieval system using the AltaVista[1] search engine. The Librarian encodes this meta data in an HTML document called the *annotation snapshot*. This annotation snapshot is then indexed by the AltaVista search engine, as if it were a normal web page.

We encode the object identifier of the video, together with timestamps to address the fragment of the video that is relevant to the meta data under consideration, in special HTML tags in the snapshot. Finding appropriate video data for a query is delegated to the AltaVista indexing engine. The Librarian parses the returned HTML documents for the encoding tag, and presents the subset of the results that refers to the videos as *media hits*. After selection by the user, these media hits start a process in the media server to deliver the video stream to the user, starting at the decoded timestamp.

AltaVista	⟨ `term`, `oid` & `timestamps` ⟩
the Librarian	⟨ `oid`, `location` ⟩

Important design decisions of a web-based digital library are clearly visible in the demo system's architecture. We do not believe that the Librarian, the software to manage the meta data, should also manage the video data. Conversely, we believe it is crucial to separate the storage of meta data from the storage of the videos, for two reasons.

Most importantly, the owners of the video data and the management of meta-information may involve different partners. The Librarian can inform users about the availability of matches at a media company selling video stock footage, or at the archive of a public broadcaster. Each company most likely has its own policies to serve and maybe sell the data (for example, we copied the following line from a footage company's website: 'We offer footage licensing on a per-second basis for use in video productions or multimedia'). The users decide whether they want to pay for quality information, or use the maybe less valuable but free public video fragments.

[1] AltaVista is a trademark of the Digital Equipment Corporation.

The second reason concerns the fast developments in multimedia networking. It is a much better idea to delegate the responsibility of video delivery to the operating system. Otherwise, we have to keep up with the pace of these developments in our database software. However, query processing and management of the meta data is a task orthogonal to the presentation of the retrieved video data. This orthogonality should be reflected in the design of a digital library system.

In the demo system, the Librarian is implemented as a set of Perl scripts [10], loosely connected to an AltaVista indexing engine. The following problems urge the adoption of database technology for the Librarian. First, the collection of types of meta data that we want to support varies over time. For example, we recently extended the demo system with feature based image retrieval. Also, as new compressed file formats for the digitized video data become available, these should be easily incorporated in the system. Second, relations between meta data and original objects are better managed by a database system. In the extended demo system, we do not only relate the videos to the words from the subtitles, but also to the automatically extracted keyframes. Also, these keyframes are related to their feature vectors. Supporting more modes of retrieval makes it more complex to manage the relations between the objects in the library. Finally, multiple users want to edit and add annotations. Database technology readily provides the functionality needed for scaling up to many users.

The rest of this paper deals with our attempt to use relational database technology for the implementation of a meta database we term the Librarian. We restricted ourselves to the relational model for reasons of software availability on the platforms we must support.

3. Architecture

Creation and maintenance of a digital library involves several more or less independent parties. These parties include human annotators, software to extract meta data automatically, and owners of multimedia footage. A digital library can only be succesful if its infrastructure can connect these parties flexibly. Like Silberschatz, Zdonik et al. [6], we try to avoid creating a monolithic database system that controls the whole environment. Cooperation between the parties participating in a digital library demands an open distributed architecture.

The main theme of our web-based digital library design is *extensibility*. It is very important that we can extend the system with new data types. These data types are new kinds of annotations, or new file formats for the bare data. If footage becomes available in MPEG2 format, we add the data types to handle MPEG2 files. And if we decide to label the speech data of the video by speaker, we have to extend the available

annotation types with a type to capture information about the speaker.

We also need extensibility on operations. We want to add and delete components for automatic meta data extraction. When new tools become available, it should be straightforward to integrate them in the environment. The new kind of meta data may be very useful for retrieval.

Recall the arguments for separating the media server from the Librarian. Similarly, a good design separates meta data extraction from meta data management. We may not own the source code of the software that provides a new method for meta data extraction, or the implementation of a new low-bitrate video encoding standard. Also, meta data extraction may take much processing, needs special knowledge bases, or uses special hardware. A good example of the last case is the extraction of closed captions from a video tape.

The architecture we developed provides extensibility in two ways. First, the Librarian provides an *extensible data model*. Second, we support a set of loosely coupled *daemons* to perform operations on the data. As we illustrate in this paper, the combination of the data model with the daemon paradigm builds an extensible collection of search methods for annotated multimedia data.

3.1. The data model

The basic representation of the database managed by the Librarian is a *semantic network* [5]. A semantic network is a graph where the nodes represent concepts. The arcs in the graph represent relationships between these concepts. The three modelling entities in the Librarian's semantic network are:

- *Semantic object*:
  The basic concept in the network. It models media objects, as well as things, persons, places, and actions that occur in the real world. In the figures in this paper, we draw semantic objects as circles.

- *Representation object*:
  Terminal node in the network, that stores a *value* for the semantic object it is connected to. A representation object can only be attached to one semantic object. It also has a type field, so that we can represent different kinds of representation objects, eg. MPEG and Motion-JPEG. Drawn as hexagons.

- *Semantic relationship*:
  A semantic relationship links two semantic objects. It is used to model annotation, the fundamental process supported in the digital library. Semantic relationships are drawn as diamonds.

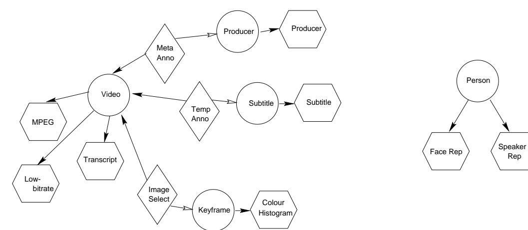

Figure 2. Modelling a video and a person in the Librarian.

Figure 2 gives a first impression of modelling objects in this data model. On the left, we represent some particular video. Several representations are attached to the video. We have the digitized MPEG video, the transcript from the closed caption decoder, and a low-bitrate version of the video for viewing over a low bandwidth connection. We also see a meta annotation for its producer, and a temporal annotation for a subtitle. Note that the semantic object for subtitle and its text representation are different entities in our model. In the physical design, we could of course combine the semantic object subtitle and its text representation in the physical design. However, on the logical level, these are different concepts. We can only specify conditions on the content of representation objects. Finally, we see how a keyframe and its colour histogram are attached to the video through an image select relationship. On the right of figure 2, we show the representation of a specific person. This person has representations attached that are used as models in pattern recognition processes. We see a model for face recognition and a model for speaker identification.

To model specific concepts, like 'video' or 'person', we define an inheritance hierarchy over the semantic objects. Inheritance enables us to inherit *relationship constraints*: constraints that apply to pairs of semantic objects. An example of such a relationship constraint is that 'only video objects can be annotated with closed-caption representation nodes'. The most important type defined under semantic object is *media object*. Media object is specialized for video, audio, text, or image. Other things, such as people or places, are also defined under semantic object.

Representation objects are terminal nodes in the network. Because only representation objects can store values for the semantic objects, conditions that specify constraints on the content of an object always involve representation nodes. Where the semantic objects are used to capture the structure of and relationships among the data managed by the Librarian, the representation objects provide the content for querying and presentation. Physically, this 'content' can be just a reference to a location in a media server.

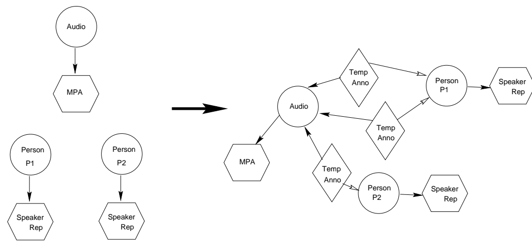

Figure 3. Annotating an audio track using speaker identification.

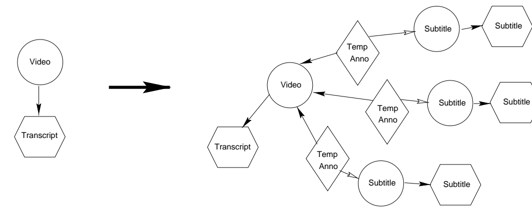

Figure 4. A daemon that annotates a video with its transcript.

Annotations are modelled with semantic relationship nodes between two semantic objects. We refer to the semantic objects linked by the relationship as the *major* and the *minor* participant in the semantic relationship. The major participant is the object that the annotation is about, the minor participant is the object that models the annotation itself.

We introduce two base types of annotations in the Librarian: *meta annotation* and *temporal annotation*. A meta annotation always applies to the major participant as a whole. Examples are the title, the producer, or a contract covering the use of footage for a movie. A temporal annotation applies only to a particular fraction of the major participant, defined by a time interval. Examples of such annotations are closed captions, color histograms of keyframes, and the output of speaker or face identification algorithms. A third base annotation type could be *vague-temporal annotation*, that would also store a measure representing the confidence that the minor participant is really about the major participant.

We also use semantic relationship nodes to model part-of relationships between different semantic objects. A video can be segmented in several keyframes, that can be used for image querying [9]. We model this dependence as an 'image select' relationship between the video's semantic object and an image semantic object. Extracted information for the keyframe image, eg. its colour histogram, is attached to the image's semantic object. Similarly, we use an 'audio select' relationship between the video and its audio track.

An audio track can be annotated with references to its speakers. With a speaker identification algorithm, this annotation process can be performed automatically if we have trained speaker models. The Librarian manages the information needed to perform this pattern recognition process. Annotating the audio track of a video by speaker is shown in figure 3. A user may later correct the automatic classification by editing the temporal annotations added by the algorithm.

3.2. The daemon paradigm

In our architecture, *daemons* are autonomous software components that perform operations on the data in the digital library. A daemon needs an input object, and generates a not-yet existing output object. The daemons are not controlled by the Librarian, but operate on their own initiative. They connect to the Librarian, request some work, and promise to perform this work within a certain amount of time. The Librarian registers what work is done, and what work is on the way. If the daemon does not finish in time, the Librarian may give out that work to a another daemon that requests work. A daemon could simply compress a raw video file, or move a captured video from an annotation machine to the media server.

The selection of work is a recursive query, issued by the daemon and processed by the Librarian. We refer to this query as the `get_work` query. A `get_work` query consists of a parent condition, p, and two child conditions, c_1 and c_2. The Librarian has to identify those objects, that fulfill p, that do have a descendent fulfilling the first child condition c_1, but do not have a descendent fulfilling the second child condition c_2. A daemon that creates a low-bitrate version of an MPEG video, specifies the following expressions:

p	semantic object type is video
c_1	representation object type is MPEG video
c_2	representation object type is low-bitrate video

We identify two daemon classes, distinguished by the kind of conditions expressed in p, c_1 and c_2. Like the low-bitrate daemon, most daemons only condition on the types of objects involved. Other daemons also have to condition on the content of the objects though. We refer to these classes as *type-triggered* daemons and *content-triggered* daemons. The distinction is very similar to the classification of multimedia objects in active and passive objects, based on their role in query processing [1].

An example of a type-triggered daemon is a daemon that annotates a video with its transcript, see figure 4. A transcript is a text representation of the information in an audio or video object, that is synchronized with the ob-

ject through timestamps. It can contain hardware-decoded closed-captions, or a text that has been produced manually during human video annotation. This daemon operates on videos that are attached to a transcript, but have not been annotated with subtitles yet. The daemon parses the transcript and adds annotations to the video. Other examples of type-triggered include a daemon that creates a low-bitrate version from an MPEG video, and a daemon that moves a digitized video file from the client machine to the media server.

A daemon that performs facial recognition on video fragments would typically be implemented as a content-triggered daemon. Because facial recognition is an expensive operation, we do not want the daemon to work on videos that are not likely to contain any human faces. Testing if a fragment has skin colour is a relatively inexpensive operation, and thus the daemon may request the subset of video fragments that have a high amount of skin colour. Condition c_1 is not just defined by the video's type, but also by its content.

4. Implementation

We implemented a prototype of the extensible architecture outlined in the previous section. For reasons of availability, we used the Postgres database system [8] as relational backend in the first implementation.

All components of the web-based digital library have been prototyped using the Perl language. Communication between clients, daemons, media servers, and the Librarian, used the HTTP protocol and an Apache web server. We wrote a (trivial) RPC mechanism in Perl, using HTTP POST and simple string-based parameter mashalling, for interaction between the Librarian and its daemons and clients.

4.1. The data model

The data model has been implemented as follows. A Postgres-library for Perl provides basic persistency with the relational model. We implemented the functionality needed to manage the Librarian's data model in several layers on top of this.

In the database, we create the following tables:

semantic object	⟨ `id`, `type` ⟩
representation object	⟨ `id`, `type`, `parent_id`, `value` ⟩
semantic relationship	⟨ `id`, `type`, `major_id`, `minor_id` ⟩

Furthermore, we use three more tables to store the type information, and another table to store the relationship constraints. The `type` field in the three tables for the basic concepts of the semantic network is the identifier of the type in its associated table of types. The value field of the representation objects stores either a string or a URL representing a location in the media server.

The first layer of the Librarian is a wrapper around the underlying relational database. In this wrapper, we provide table inheritance. For convenience, we used the inheritance model readily provided by Postgres. In this bottom layer, we also hide caching of the tables used to maintain our type system. This is not entirely safe, because another user could change the type system interfacing directly to the database. To avoid such inconsistencies, the underlying database can only be accessed through the Librarian.

The second layer provides the base data types needed to represent the semantic network. The basic operations are creation and deletion of a new data type, creation and deletion of new objects, attachement of an object to another object, and the management of relationship constraints. For example, when adding a new semantic object type, like 'Video', we must also specify which representation types it can be linked to, like 'mpg', 'transcript', and 'low-bitrate'.

The Librarian's top layer is the API. This API specifies how daemons and clients interact with the Librarian. It manages fragmentation of objects over semantic objects and representation objects. It also provides the query interface to clients and daemons. In the current implementation, we can only express conditions over three different views, consisting of the fields that are shared by all instances of semantic objects, representation objects, or semantic relationships. The API call `find_objs` executes an SQL query over one of these views. Extra fields added for subtypes can only be accessed through `get_all_attributes`, and cannot be used for querying.

Retrieving all video's that contain 'all American bear' in the subtitles works as follows. First, we specify a condition on the value of the subtitle. Thus, the where clause of the query we perform over the representation view is:

```
value LIKE '%all American bear%'
      AND type = 'subtitle'
```

Next, we select all annotations that have a minor participant in the set returned by this query, and we join the major participant identifiers of these annotations with the view on semantic objects to return the video objects containing this phrase.

4.2. The daemon paradigm

Daemons connect to the Librarian and request work. Next, they commit to perform the work within a specified interval. If the daemon does not finish within this interval, the work expires, and can be handed to another daemon. The management of work that is 'in progress' is based on keys encoding a timestamp when the work is due. The database generates a key for the object, and also creates a temporary object that stores the key and functions as a placeholder for the result. The key is handed to the daemon, and is required

when the daemon submits its work. Because the key encodes a timestamp, it is easy to collect the identifiers of expired placeholders while processing new requests for work.

The Librarian provides two ways to handle a daemon's request for work. If the daemon calls `get_and_do_work`, it promises to do the work within the amount of time it specifies as a parameter. After finishing its work, the daemon calls `finish_work` to submit its results. Some daemons need information about the object to work on, before they can indicate the time it will take to perform the job. For example, the time necessary to generate a low-bitrate version of an MPEG video depends on the size of the video. In that case, the daemon first calls `get_work`, which returns the object identifiers of a set of objects that need work. Subsequently, the Librarian can provide more information about the object throught the normal query interface. Then, `do_work` is called to commit to the work. If this job has not been handed out in the meantime, the daemon proceeds like before.

The `get_work` query is a recursive query, because we do not know the structure of the database before we access it. We have to ascend the outgoing connections from the objects that satisfy the parent condition, to check that there exists a child for condition c_1, but no child fulfilling condition c_2. Traversing the recursive structure can be an expensive operation. The straightforward implementation iterates over the objects that satisfy p, and traverses the network for each parent object sequentially. By using three temporary tables to manage two generations of objects under consideration, and the parent objects that still satisfy both conditions, it is possible to traverse the network for all parents at once. This approach significantly speeds up processing of the `get_work` query.

5. Evaluation of the prototype implementation

Evaluation of the prototype implementation described in the previous section revealed several performance problems. For example, adding only 700 annotations for an hour of digitized video takes an unacceptable 55 minutes on an Alpha NT box with 256 MB main memory. Some of these performance problems are caused by the low performance of the Postgres database system, and will automatically disappear with a higher performance database. However, the layered implementation of our extensible data model introduces unnecessary query processing that should be avoided.

Because we used the relational data model for the implementation of the Librarian, the following pieces of the data model have to be managed *outside* the database system:

- Use of the type system;
- Rule checking upon insert and delete;
- Fragmentation of objects over separate representation and semantic object tables.

Because this processing takes place outside the database, in the interaction with the relational backend we often perform *tuple-at-a-time* processing instead of *set-at-a-time* processing. Also, the optimizer cannot choose the correct query execution plan, because it does not know the relationships at the higher level. To avoid these problems, the bottom layers that provide our extensible type system should be pushed into the database management system.

Another problem, also related to the implementation of the data model on a relational database, is the lack of power of the associated query language. Since we cannot use new operations on the data in the query language, it is not straightforward to query on content of the objects in the database.

One result of this problem is that including content-triggered daemons in this implementation forces us to separate such a daemon in two subdaemons. For the face-recognition daemon described before, that prefilters the videos using a condition on the colour distribution of the keyframe images, we first have to perform the operation specified in the condition and store a numeric value as output value. Next, we can refer to that value in condition c_1 of the second subdaemon, that really does the facial recognition.

6. Advantages of extensible database systems

An object-relational system extends the relational model with abstract data types (ADTs) [7]. ADTs offer many advantages for the implementation of the Librarian. Most importantly, the type catalogues are maintained *inside* the database instead of in external code. This reduces the overhead introduced by the type system, because the database can cache the type tables in memory. Because the database controls the type catalogues, this simplifies cache management.

Operations on the data are available in the query language. Queries that cannot be expressed as text expressions, like image similarity queries, can be now be expressed using method calls. The Chabot image database management system demonstrates how an image ADT in Postgres enables the usage of both traditional querying and approximate retrieval techniques [4].

With data extensions inside the database, we can improve performance of query processing in at least two ways. First, we can avoid some processing by encapsulating some relationships within an ADT. If we create an abstract data type that encapsulates all subtitles in one large object, like in figure 5, we can achieve higher performance. Second, the optimizer can choose specialized index structures provided by the type extensions. For instance, queries involving similar-

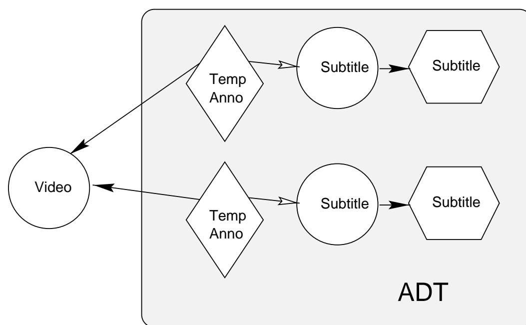

Figure 5. Encapsulate several subtitles in one ADT.

ity search can be processed more efficiently using spatial access methods [2].

Another advantage of the object-relational model is that we can build indices on function expressions. This functionality can be exploited to avoid the permanent storage of intermediate results. For example, the process shown in figure 3 uses the audio representation of a video to add temporal annotations to that audio representation. In the relational model, we need a table for the audio object related to the video, and a table to store the temporal annotation. With indices on function expressions, we can create an index on `speaker_detect(audio_select(video), person)` instead. Hence, we do not have to maintain the intermediate audio result. We answer queries for the temporal annotations from `speaker_detect` using the index structure.

7. Further work

The first task for further work is the development of a suitable query language for the digital library. Extensible databases, discussed in the previous section, seem to provide a good interface to the management of data. However, traversal of links in the semantic network cannot be expressed in a declarative nature in an object-relational database like Illustra™ . The definition of a query interface, that does not conflict with the extensible nature of the architecture, requires more research. The problems we found in the evaluation of the prototype, indicate that simply building some layers on top of an object-relational database (instead of a relational database) may not provide sufficient performance either. We have to perform query optimalization in these layers, similar to the approach taken in the Garlic heterogeneous database project [3].

The communication between the components of the digital library is another topic for further research. A so-called 'software bus' provides better control over the actors in our environment. We are currently redesigning this part of the digital library architecture based on CORBA's distributed object model. This approach provides better tools to provide security. In the current approach, any piece of software that knows the Librarian's API can change the meta data.

The combination of such a software bus with an extensible database creates an opportunity for a tighter integration of daemons to the database. Some of the daemons can be implemented as methods that operate from within the database. But, as we discussed before, we should realize that often the code to do meta data extraction is not under complete control of the party managing the meta data.

With an architecture based on CORBA, we can handle this situation as follows. In the database, we specify the input and output of the daemon as a method call of the data type of the objects it operates on. The daemon is implemented as a callback, executed from within the method defined in the database. The `get_work` query can be expressed in the same query language as other queries. Also, a daemon implemented as a distributed object, does not have to query the database to retrieve its input information, but performs its operations from within the context of the database. We do not have to define the location and implementation of the daemon, but we must specify the behaviour of the daemon. From a viewpoint of data management, this seems a better idea than the very loosely coupled structure we support now. Further research is necessary to evaluate this approach.

8. Conclusions

Collections of multimedia objects in digital libraries can only be managed in a loosely coupled infrastructure consisting of databases, advanced media servers, daemons, and web-based clients. We should not integrate presentation and access in a big monolithic database system, but separate media service from management of meta data. Similarly, the extraction of meta data should be relatively independent of the management of the extracted meta data.

The combination of an extensible data model with the proposed daemon paradigm has been shown to be capable of providing the functionality for both querying and annotating. However, the evaluation of the prototype implementation revealed several problems with the implementation on a relational platform. We analyzed the advantages offered by object-relational technology, and propose further research in this direction.

The problems with modelling nested relationships in relational, but also in object-relational databases, form a major problem for providing efficient management of meta data. This problem impedes the adoption of database technology to manage data in the digital library setting. Databases suited for a role in digital libraries efficiently handle query-

ing and model the complex relationships between the managed data.

References

[1] E. Bertino, B. Catania, and E. Ferrari. *Multimedia Databases in Perspective*, chapter Query Processing, pages 181–217. Springer Verlag, 1997.

[2] C. Faloutsos. *Searching multimedia databases by content*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1996.

[3] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. An optimizer for heterogeneous systems with nonstandard data and search capabilities. *Bulletin of the Technical Committee on Data Engineering*, 19(4):37–44, December 1996.

[4] V. Ogle and M. Stonebraker. Chabot: retrieval from a relational database of images. *IEEE Computer*, 28(9):40–48, September 1995.

[5] G. Sagerer and H. Niemann. *Semantic networks for understanding scenes*, chapter Knowledge representation. Plenum press, 1997.

[6] A. Silberschatz, S. Zdonik, and et al. Strategic directions in database systems - breaking out of the box. *ACM Computing Surveys*, 28(4):764–778, December 1996.

[7] M. Stonebraker and D. Moore. *Object-relational DBMSs: The next great wave*. Morgan Kaufmann Publishers, Inc., 1996.

[8] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[9] H. Wactlar, T. Kanade, M. Smith, and S. Stevens. Intelligent access to digital video: The informedia project. *IEEE Computer*, 29(5), May 1996.

[10] L. Wall, T. Christiansen, and R. Schwartz. *Programming PERL*. O'Reilly & Associates, Inc., 2nd edition edition, 1996.