INFORMATICA, 1997, Vol. 8, No. 1, 23–42

COMPLEXITY ANALYSIS OF LINK NAVIGATION IN DEXTER BASED HYPERMEDIA DATABASE SYSTEMS

Günther SPECHT

Technische Universität München, Department of Computer Science
Orleansstr. 34, D-81667 München
Email: specht@informatik.tu-muenchen.de

Abstract. Today's multimedia and hypermedia systems include such a huge amount of data and links, that they should be stored and maintained by a database system. Then a powerful and efficient database schema is needed. The Dexter hypertext reference model offers a widely accepted, powerful modelling technique for nodes and links. We present its stepwise conversion into a relational multimedia database schema. In the obtained hypermedia engine the most important and most time critical operation is the link navigation. We analyze its complexity in detail and optimize it by schema improvements. Finally we present an efficient implementation of the presented ideas: the System MultiMAP, developed at the TU Munich.

Key words: hypermedia, database systems, Dexter reference model, relational model, link navigation, complexity analysis.

1. Introduction. Today, a lot of hypertext and hypermedia systems refer to the Dexter hypertext reference model (Halasz and Schwartz, 1994). This is a widely accepted proposal for modeling hypermedia links. Initially the proposal was set up for file based systems. The main advantage of the Dexter model is its powerful link concept, including links as own entities, $n : m$ links, typed links, links on links, etc. This goes far beyond todays WWW techniques.

On the other side multimedia and hypermedia systems include such a huge amount of data and links that they should be stored and maintained by a database system. That offers additional benefits in opposite to file based systems as referential integrity of links, efficient and index supported access structures, transaction protected multi user access and recovery.

In this article we present the stepwise conversion of the Dexter model into a multimedia database system. Thus we like to combine the benefits of a powerful link model with the efficiency and benefits of databases. Since the final system has to be fast not only in data access but also in link navigation we give a detailed complexity analysis for the complex operation of link navigation in Dexter based hypermedia database systems.

Today two different database technologies are used for multimedia applications: relational and object-oriented ones. A detailed comparison analysis (Specht and Hofmann, 1996) pointed out, that in opposite to all benefits of object-oriented database systems (as easier modelling and implementation, less code, etc.), relational databases are still the more efficient ones according to hypermedia applications. Thus we focus on relational databases in this article.

In order to apply the Dexter model to relational databases, we first have to model the storage layer, which can be seen as the kernel of the Dexter model, as an Entity-Relationship Diagram. Applying the common conversion rules we get a relational database schema for hypermedia databases. Then link navigation can be expressed as a SQL-query. Since its performance decides about acceptability, our aim is to get universally valid complexity formulas for link navigation in Dexter based relational databases. Therefore we have to consider the internally processed query execution plan at the level of relational algebra expressions in order to figure out which joins and operations are most time consuming and which optimizations, even on the schema, can be applied.

This analysis is not only interesting in the theory of multimedia database systems but also in practice. We have implemented MultiMAP, a Dexter based multimedia database system, which is used in a series of applications, e.g., a city information system, based on city maps, a medical system, maintaining x-ray pictures, a multimedia library information system and others.

This article is organized as follows. Section 2 gives a brief recapitulation of the relevant parts of the Dexter reference model, mainly the storage layer and the representation of nodes and links there. Section 3 introduces the corresponding Entity-Relationship Model and the resulting relation schema. Section 4 presents the complexity analysis of the link navigation and Section 5 discusses optimizations of it. Finally in Section 6 we present a practical implementation of all these: our multimedia database system MultiMAP and some of its applications.

2. Representation of nodes and links in the Dexter reference model. The Dexter hypertext reference model (Halasz and Schwartz, 1994) is one of the most approved reference models for hypermedia systems. It has been developed during two workshops; the first one took place in Dexter Inn, New Hampshire, USA, thus the name. The overall goal was to specify a standard terminology and a reference model for hypertext systems, in order to be able to compare even quite different system implementations and notations to each other. The results are extensible to hypermedia systems including continuous (time dependent) data like audio and video, as for instance the extension of the Amsterdam hypermedia model (Hardmann *et al.*, 1994) shows. Although the Dexter model is a well-known standard today, only a few systems include all of its functionality. The main benefit of this model lays in the power of its link concept which goes far beyond usual WWW-links: Not only $1:1$, but arbitrary $n:m$ links are supported. This denotes an extension of classical hypertext structures, which can only manage $n:1$ links. Already at $1:n$ links, an additional selection component is necessary. In addition, links on links are definable, bidirectional links are supported, composite components are included, span to span links and typed links are contained, etc.

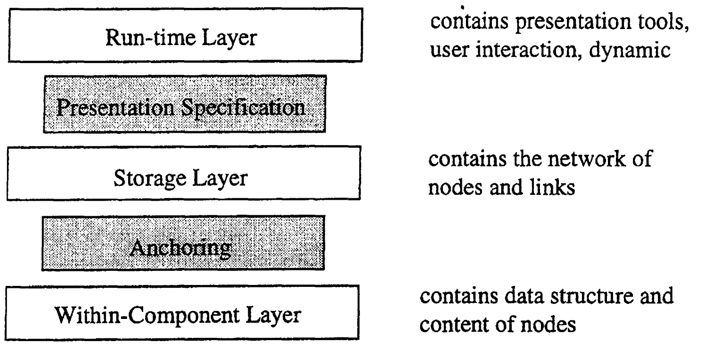
Run-time Layer
Presentation Specification
Storage Layer
Anchoring
Within-Component Layer
contains presentation tools, user interaction, dynamic
contains the network of nodes and links
contains data structure and content of nodes

Fig. 1. The three layers of the Dexter model.

The Dexter reference model consists of three layers. The upper one is the *Runtime Layer.* It manages the presentation tools, the user interactions and the dynamics. Different sessions at a time – one per user – are administered. In addition, for each used component several socalled instances have to be handled, which are the read/write copies of the components in the cache, since updates are possible.

Beneath it there is the *Storage Layer*, which contains the hypermedia net of nodes and links. It is the most important layer of the model and will be described below in more detail.

The lowest layer is the *Within-Component Layer*. It describes the content and structure of the components (nodes); e.g., the data structure for text, image, animation, etc. This layer is system specific, so it is not defined in the Dexter model in more detail (e.g., defined in ODA, IGES, etc.). The two interfaces between the three layers are the presentation specification and the anchoring.

The representation of nodes and links is kept inside the storage layer. Since this layer is the most important one for modelling power, storage and access performance of any hypermedia net, we will discuss it in greater detail now. The storable hypermedia objects are called components. There exist three types of components:

- atoms,
- links,
- composite components.

Atoms are simple nodes. They can be treated as primitives. Composite components are hierarchically structured nodes (particularly a directed acyclic graph (DAG)). Links are specified by two or more anchors (= endpoints). More precise, anchors are defined inside the components and included in links via the concept of specifiers. Each specifier corresponds to exactly one anchor including information on weather it is a source or a target anchor or both. Anchors can address whole components or substructures of a component. Thus we get the power of "span to span links".

For identification, each component has a "global unique identifier" (UID). Anchors are defined locally in the component header. They consist of an "anchor identifier" (AID), which is locally unique within a component, and an "anchor value", that specifies a location inside the component. Only the AID is visible from outside the component. Thus a globally unique link anchor consists of the tuple (UID, AID).

A sidenote. Since, according to the definition, links are components, it is possible to define links whose endpoints are other links or even parts of links (for instance a link on a link description or type or direction of another link).

Inside a link definition, specifiers contain besides the presentation specification (e.g., colour mark of the link anchors) also the direction, which may have the values 'from', 'to', 'bidirect' and 'none'. Thus, it is not only possible to define unidirectional and bidirectional links, but also $1 : n$ links and $n : m$ links, because a link may contain several source and target specifiers. Fig. 2 shows a resulting image for atoms and links, seen at the level of the storage layer.

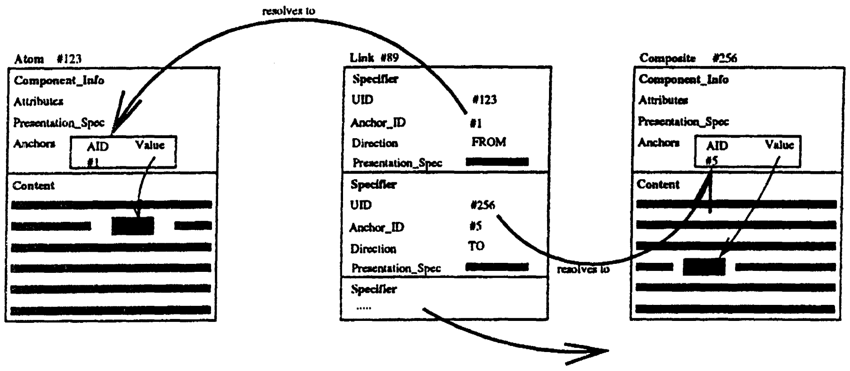

Fig. 2. Overall organization of the storage layer (adapted from Halasz and Schwartz (1994)).

Summing up, the Dexter model is a more powerful model than the models that are usually used in hypermedia systems. It includes:

- $1 : n$-links and even $n : m$-links,
- links to links (which are hard to implement),
- complex components,
- bidirectional links,
- a very powerful resolver function, which we have not mentioned by now.

3. Conversion of the Dexter model into the relational DB-model. Initially, the Dexter model has been developed for file systems. But facing the huge amounts of nodes and links in today's applications, a direct conversion to a database system would be of great advantage. In addition, transaction protected processing and referential integrity of links would also become possible. Both of these are missing in the file based WWW.

Two database technologies are eligible for today's multimedia systems: relational and object-oriented database systems. We have converted the Dexter model into both: a relational database schema, which we used as a basis for our multimedia database system MultiMAP and an object-oriented database schema, which we extended to a multimedia database system implemented on top of a commercial OO-DBS. We used for the relational variant the Entity-Relationship Model, and for the object-oriented system the Object Modelling Technique (OMT) (Rumbaugh, 1991) as intermediate modelling stages. Due to its greater efficiency, we focus in this article on the relational conversion.

Although the Dexter model has not been developed for databases, the storage layer can easily be transformed into an Entity-Relationship diagram. See Fig. 3.

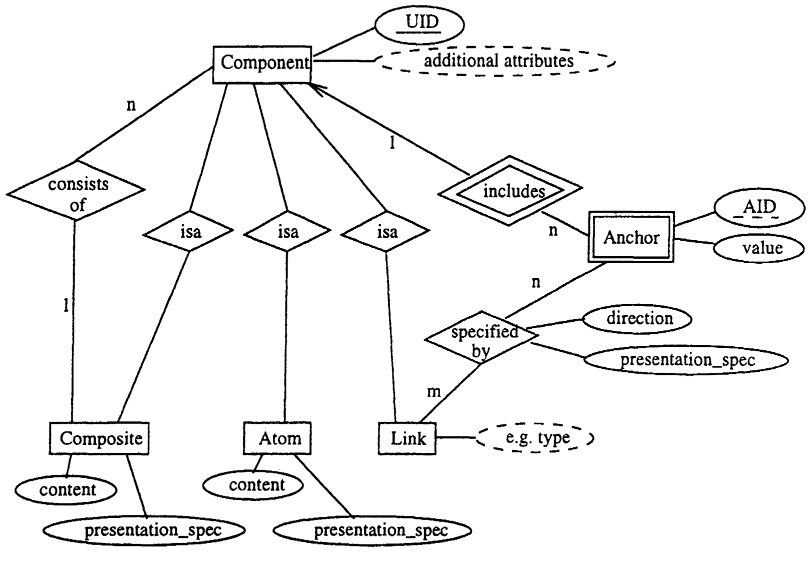

Fig. 3. ER-diagram of the storage layer.

Atom, link and composite are components, thus there is a *isa* relationship to components, via which all attributes of component, especially the key attribute UID, are inherited. One composite (-component) can consist of n (sub-) components. More interesting is the right side of the diagram. Components include anchors. Since anchors can not exist by themselves, only inside of components, they are connected to components via an existential dependency, denoted by a double rhombus. Anchors are weak entities, denoted by a double rectangle, since they can not be defined unambiguously by their own attributes, but only by their relationship to other entities, here the component. By the way, weak entities are always existentially dependent from others. Here the key attributes of anchors are the local key attribute AID together with the global key attribute UID, inherited via the existential dependency.

One link is specified by n anchors whereas 1 anchor can occur in m different links. Thus anchor and link are connected via an $n : m$ relationship, containing additional information about the direction and presentation (underlined, highlighted, blinking, etc.) of that anchor. This relationship maps exactly to the specifiers, since it gets the key attributes from the corresponding entities (AID and UID from anchor and Link_UID from link) in addition to its own attributes like direction (containing the values ‘from’, ’to’, ‘bidirect’ or ’none’).

Now the usual conversion rules from entity-types and relationship-types into relations can be applied. Let’s recapitulate them for short:

1. Mapping entity-types to relations:

   An entity type E with k attributes A_i from domains D_i, $(1 \leqslant i \leqslant k)$ is mapped into a k-ary relation $E(A_1 : D_1, A_2 : D_2, \ldots, A_k : D_k)$. If there also exists E *isa* F or if E is a weak entity type, existentially dependent on F, we also include all the key attributes of F.

2. Mapping relationship-types to relations:

2.1. $n : m$-relationships and all 3ary, 4ary, ..., etc. relationships.

   A relationship R between entity types $E_1, \ldots, E_n$ is mapped into the relation R. The attributes of this relations are the primary keys of all E_i. Equal attribute names have to be renamed in the relation R, in order to be unique. If R has attributes of its own, these will be added.

2.2. $1 : n$ relationship between only 2 entity types E and F.

   No relation R is generated from the relationship. We append the primary keys of E as foreign keys in the relation F instead. If the relationship R has own attributes, they have to be added to F.

2.3. $1 : 1$ relationship between only 2 entity types E and F.

   No relation R is generated from the relationship. We append the primary keys of E as foreign keys in the relation F or the primary keys of F as foreign keys in the relation E.

2.4. is-a relationship and existential dependencies.

E *isa* F and existential dependencies are not mapped into relations, because that would lead to redundant tuples like $isa(133, 133)$.

Applying these rules to the above entity-relationship diagram results in the following relations:

```
Atom (UID : integer,
      presentation_spec : string,
      content : BLOB)  /* here simplified as BLOB, in detail: IMAGE,
                          STRING, AUDIO etc.) */

Anchor (UID : integer,
        AID : integer,
        value: string)

Link (UID : integer,
      type : string)  /* optional */

Specifier (link_UID : integer,
           anchor_UID : integer,
           anchor_AID : integer,
           direction : string,
           presentation_spec : string)

Composite (UID : integer,
           presentation_spec : string,
           content : BLOB)

Composite_consists_of (UID_father : integer,
                       UID_child : integer)
```

We omitted the relation component, since it is useless afterwards. Only its instances atom, link and composite are furthermore needed. Additional attributes normally included in the relations are, e.g., author, creation_date, access_rights, etc. Primary keys are underlined. As usual in relational systems, in the following we will assume that there is an index on primary keys for faster access, a prefix-B*-tree to be exact.

A sidenote. Extending the Dexter hypertext (!) model to hypermedia, including, e.g., images, implies that the anchor value becomes more complicated. While for texts, audios and videos (if anchors address only single images in it) an anchor value of type string or integer is still sufficient, marking arbitrary surrounded objects within pictures as anchors need a further relation. Then we extend the relation anchor by attributes for the bounding box of the objects, in order to get a fast preselection while clicking in the picture and introduce an additional relation containing the precise polygone coordinates.

4. Complexity analysis of link navigation. Link navigation is the most common operation in a hypermedia database. In the following, we will analyze and optimize the complexity of link navigation. What happens if we click on a link source on the screen? In essence the following steps take place:

- in the runtime layer:
  - determination of the component instance, that has been clicked on,
  - selection of the link-marker (= the instance of the link),
  - mapping instance (cache) to component: instanceID $\longrightarrow$ UID,
  - mapping link-marker (cache) to anchor: link-marker $\longrightarrow$ AID;
- in the storage layer:
  - search for the link source AID in the relation specifier to determine the link_UID,
  - search for all corresponding link target specifiers (= selfjoin over link_UID),
  - join with the relation anchor to get the anchor value (= exact link target specification),
  - get the component that contains the link target (selection on atom or composite or link);
- in the runtime layer:
  - create an instance for the target component (UID $\longrightarrow$ instanceID),
  - create an instance for the link target marker (AID $\longrightarrow$ link marker),
  - representation on the screen (e.g., blinking).

Let's take a closer look at the events in the storage layer, since those are the ones with database access. The above access sequence within the storage layer can be expressed in SQL according to the above presented relations:

```
SELECT  a.UID, a.AID, s2.presentation_spec
FROM    Specifier s1, Specifier s2, Anchor a
WHERE   s1.anchor_UID = <$input>                /* Selection */
AND     s1.anchor_AID = <$input>
AND     s1.direction IN ('from', 'bidirect')
AND     s1.link_UID = s2.link_UID               /* Selfjoin */
AND    s2.direction IN ('to', 'bidirect')
AND    (s1.anchor_UID ≠ s2.anchor_UID      /* Postselection */
        OR s1.anchor_AID ≠ s2.anchor_AID)
AND    s2.anchor_UID = a.UID               /* 2nd Join */
AND    s2.anchor_AID = a.AID;
```

Followed by a selection of the component of the found UID. Let, e.g., UID $\in$ Atom UIDs:

SELECT * FROM Atom WHERE Atom.UID = <found UID(s)>

Analysis. We get two selection operations and two joins (maybe even three, if the attributes of the relation link are relevant (e.g., link type)) in the first SQL query. For an exact analysis, the structure of the internal query execution plan (QEP) on the level of relational algebra operations is important. See Fig. 4.

This QEP is highly optimized, provided that the lines marked as post selections in the SQL-query above will be translated to selections and not to joins. An integration of the postselection into the last join would be worse, since then no index support could be used, and nested loop joins are more expensive than index joins. We marked the most costly and interesting operations within the QEP. All others can be executed "on the fly" (like in a pipe), thus their costs are at most linear and do not give a significant factor to the end result.

1. **Selection.**
   s1.anchor_UID = <$input>,
   s1.anchor_AID = <$input>.
   This is a selection on non-key attributes, therefore if no optimization takes place, a relation scan of the order $O(|\text{specifier}|)$ will be necessary. But introducing secondary indexes, the complexity can be reduced to: $O(\log_k(|\text{specifier}|))$, with k beeing the branching factor of the B-tree index (mostly $k \approx 100$).
2. **Selfjoin.**
   specifier $\triangleright\!\triangleleft_{link\_UID=link\_UID}$ specifier.
   Link_UID is a primary index, thus an index join can be used. Its complexity is:
   $O(n * \log_k(n))$, with n = |specifiers|.
   Precise: That's an upper bound, usually the first n is less than the second n due to above preselection.

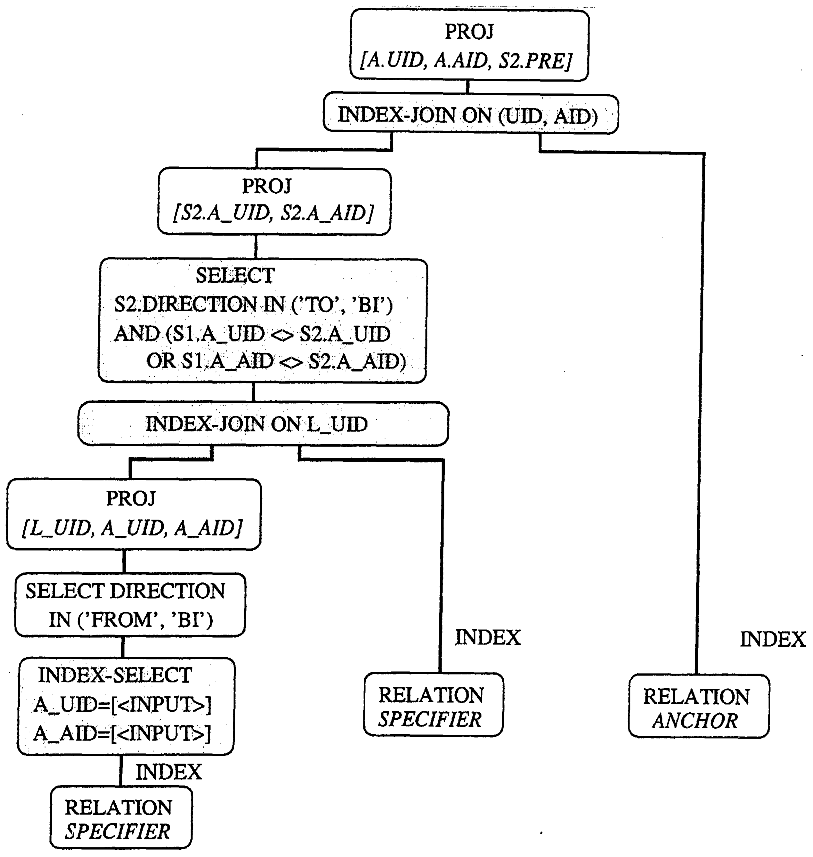

Fig. 4. Query execution plan for a link navigation query.

3. **Postselection.**
   s1.(anchor_UID, anchor_AID) $\neq$ s2.(anchor_UID, anchor_AID).
   The inequality test can never be executed index supported, so the complexity will always be linear; $O(s)$ with s= |result tuple of the selfjoin|.
4. **Join with anchor.**
   selfjoin $\triangleright\!\triangleleft_{\substack{A\_UID=UID\\A\_AID=AID}}$ anchor.
   Since the join attributes are primary key in anchor, an index join can be used again, resulting in the complexity $O(|\text{selfjoin}| * \log(|\text{anchor}|))$.

So far, we did not discuss the selection on direction with linear complexity $O(|\text{specifier}|)$ and the projections. A special unique node for duplicate elimination is not necessary (refer Fig. 4), since all projections include primary key attributes in the output. In advanced implementations the projection can be included in the previous selection as output projection. Thus they are done on the fly and cost almost nothing. In all other implementations projections (without duplicate elimination as in SQL) are of linear complexity by the number of input tuples. Even if we assume the last variant, we get as an *upper bound* the following resulting complexity:

let $n = |\text{specifier}|$ and $s = |\text{result tuple of the selfjoin}|$,

$O(\log n + 2n + n * \log n + s + s + s * \log(|\text{anchor}|) + |\text{ answer\_tuples}|)$.

A closer look on the joins even shows that the incoming m of the used $m * log(n)$ joins (m is always on the left side of the joins in Fig. 4) is always very small! For the first join m would be rather 1, since the first selection is on the source anchor, and usually only one source anchor is used in one link, although $n : m$ links are supported by the model. For the second join the incoming m is in average about 1.8, since mostly $1 : 1$ links are used and if $1 : k$ links are used, the k is small (in avg. < 10) again. Thus, what is the *average complexity* of link navigation?

$O(\log n + 1 * \log n + s + 1.8 * \log(|\text{anchor}|) + 1.8)$
$= O(2 \log n + s + \log(|\text{anchor}|))$.

Since the intermediate results are always very small, and the corresponding join relation usually fits in the main memory, and is already index supported, link navigation based on a Dexter model can be very efficiently implemented within relational multimedia database systems.

But there are still some more optimizations and modelling variants which are worth to think about.

5. Optimization and modelling variants

5.1. Local or global Anchor_IDs? In the Dexter model, anchor_IDs (AIDs for short) are unique only within a component. I.e., a unique address is denoted by the AID together with the UID for the component. Hence some SQL queries seem to be more complicated as with globally unique AIDs. The reason for local AIDs in the Dexter model is simply that in non database systems the authors do not know which AIDs are currently created by others. (I.e., without a transaction concept, there are access conflicts and inconsistency). Therefore it is safer to implement local AIDs inside the components which can be easily overlooked by the authors.

As an opposite, in database systems global Anchor_IDs (gAID) are easy to implement, because each new allocation takes place within a transaction. Thus no double assignments are possible. Global AIDs have some advantages: first of all, less SQL code is needed for link navigation, and second memory can be saved by one column in the relation Specifier (only). Runtime savings are minimal, because multiattribute-joins and multiattribute-selections are available, even index-supported.

Result. Shorter SQL-code, "thinner" relation specifier, for the rest the advantages are small.

5.2. Discussion n : m-links versus n ∗ 1 : m-links. The Dexter model offers the possibility to define n : m-links as entities, because it is possible to configurate several specifiers within a link with the same direction (e.g., 3 'from' anchors and 4 'to' anchors). We have adopted this functionality for our relations. On the other hand, it is also possible to decompose n : m-links in n 1 : m-links.

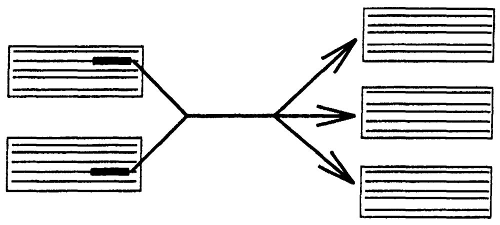

Fig. 5. n : m-link or MSMD-link (multi source multi destination) with several source and target anchors.

The experience gained with MultiMAP applications[1], in which we have put at disposal the full n : m functionality based on the Dexter model, taught us that on the one hand, application developers rarely define n : m links and, on the other hand, queries on link source siblings are very rare. But this additional query possibility is the only advantage of n : m links over the implementation as n 1 : m links. Thus it is worth to think about interesting optimization

[1] The applications of the MultiMAP system are presented in Section 6.2.

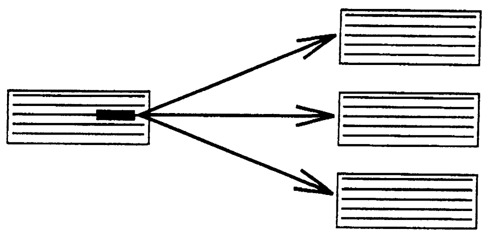

Fig. 6. 1 : n-link, from one source, several link targets are reached.

possibilities, which are obtained, if all the links are internally represented as 1 : n links.

For 1 : m links, there is one distinguished source-specifier[2] per link. This makes it possible to combine the relations Specifier and Link to one relation. The new relation "Links" contains the link source anchor, followed by one of the destination specifiers and optional general link informations. We will get an own tuple entry for each destination anchor within the 1 : m relationship. Thus for real n : m links a bit less than n times more storage is needed. If we still want to handle n : m links, the sequence link_UID, source_AnchorID[3], dest_AnchorID becomes primary key, or if we want to give each 1 : m link its own link_UID only the sequence link_UID, dest_AnchorID or (equal) the sequence source_AnchorID, dest_AnchorID are primary key candidates. The choice is easy. Since the incoming join always refers to source_AnchorID, there should be an index support on this attribute. If we choose source_AnchorID instead of link_UID as first part of the primary key, this index is automatically supported as a primary index (i.e., no secondary index will be necessary on that relation). Our new relation Links, being unified from the former relations Link and Specifier looks like this:

```
Links (Link_UID: integer,
        global_source_AnchorID: integer,
                                        /* otherwise: source_UID, source_AID */
        global_dest_AnchorID: integer,
```

[2] This can be the 'from' specifier or one of the specifiers labelled with 'bidirect'. For bidirectional links see Section 5.3.

[3] We omit the prefix 'global_' for readability reasons during this discussion.

```
    direction: string,            /* still can be 'bidirect', see below */
    presentation_spec: string,
    [type: string,]
    [date: date,]
    [author: string] )
```

As an important result, every target anchor can be reached directly from its source anchor and the indirect step via the selfjoin over the relation specifier becomes unnecessary.

Result. It is possible to omit the selfjoin! Only for real $n : m$ links a bit less than n times more storage is needed. But this occurs rarely.

5.3. Modelling bidirectional links. Our new relation Links still contains the attribute direction. If we have a bidirectional link, then during link navigation both have to be tested: either global_source_AnchorID can be the link source and global_dest_AnchorID the target, or the other way round. Bidirectional links modelled in this manner means minimal storage but more difficult SQL querys for the link navigation. An alternative solution would be to store each direction as an own tuple. Then we have to store a few more tuples (at most twice as much), but get easier SQL-queries, since global_source_AnchorID is the only referenced attribute for the incoming join. Summing up, we get two possible variants:

a) Storing one tuple for both directions (as until now)
Link navigation in SQL:

```
SELECT a.UID, a.AID, l.presentation_spec
FROM   Links l, Anchor a
WHERE  (l.global_source_AnchorID = <$input>
         AND l.global_dest_AnchorID = a.global_AID )
OR     (l.direction = 'bidirect'
         AND l.global_dest_AnchorID = <input>
         AND l.global_source_AnchorID = a.global_AID )
```

b) Storing one tuple for each direction in the link:
Link navigation in SQL:

```
SELECT a.UID, a.AID, l.presentation_spec
FROM   Links l, Anchor a
WHERE  l.global_source_AnchorID = <$input>
AND    l.global_dest_AnchorID = a.global_AID
```

Result.

Variant a. Attribute direction:

pro:

- compact storage of the link information in one tuple

contra:

- more complex SQL-query
- higher memory consumption for access structures, because a secondary index is needed on global_dest_AnchorID.

Variant b. One tuple per direction:

pro:

- simple query for link navigation
- less memory needed for access structures

contra:

- up to twice as much storage space for link data. But since that are only some integers, that is not significant in comparison to the storage needed for nodes, containing images, audios and videos.

Finally the complexity for link navigation can be reduced to
$O(\log(|\text{links}|) + |\text{links}| * \log(|\text{anchors}|))$.

Final remark to Section 5. We have analyzed and discussed link navigation and its optimization. It is the most frequent and therefore most important operation in hypermedia database systems. Its efficiency is crucial for user acceptance. We found a highly optimized solution. But still, the same cost analysis has to be considered for

- creation of links,
- deletion of links,
- creation of components,
- deletion of components (includes possibly some link deletions)

in order to get an optimal, balanced system behaviour, depending on the application scenario.

6. The multimedia database system MultiMAP

6.1. The system. Based on the Dexter model we have developed the multimedia database system MultiMAP. In addition to the powerful link concept of the Dexter model, MultiMAP also allows fulltext search as an entry to the hypermedia net. Thus, the underlying relational schema is a bit larger and more complex as presented above.

MultiMAP is an interactive, extensible hypermedia database system, in which texts, images, arbitrary objects on the images, audios and videos can be stored and connected by links. MultiMAP runs on Unix workstations (Sun4 Sparc, Hp, etc.) using a client/server architecture, and the relational database system TransBase^{TM} as backend for internal data management. The main focus of MultiMAP is the support of fast and simple (mouse supported) creation of applications, since today, not the set up of a system, but the input of all relevant data and the long-term maintenance are the most costly factors in multimedia applications. Due to its database functionality even deletion of nodes does not touch the referntial integrity of links. Thus always a consistent application is presented. Further advantages of MultiMAP over file based systems are:

- integrated processing of big amounts of multimedia data. All multimedia data is completely stored inside the database. There are no pointers to the file system for BLOBs;
- optimized storage due to efficient access paths and index structures;
- multiple complex search possibilities;
- referential integrity of links;
- transaction protected multi user mode;
- full recovery capability.

The link concept of MultiMAP is an extension of the Dexter model. It goes far beyond usual WWW-links:

1. Support of uni- and bidirectional links and arbitrary $n : m$ links. That includes the heavily used $1 : n$ links in our applications.
2. Extension of the hypertext concept on arbitrary graphical objects: link source and target anchors can be arbitrarily outlined objects on images (e.g., the course of a river or a plot of land on a map). In particular, these do not need to be approximated by rectangles.
3. In addition to links, it is possible to execute full text search (even truncated and nested) on all text, image and object names of the database. The full text search is integrated in the object recherche and behaves like an additional dynamic link.

We have also implemented the complete system as an object-oriented database, using the object-oriented database system O2 as a backend. We did this in order to obtain comparisons regarding modelling, easy development, effectiveness and performance. A detailed examination (Specht and Hofmann, 1996) showed that object-oriented databases do not show the desired performance and user interactivity at huge amounts of data. Thus the relational variant of MultiMAP is still the more efficient one.

6.2. Applications. MultiMAP is already used in a series of applications, partially with large amounts of data and high user activity. We present only a few of them:

1. The first field of application was the multimedia processing of maps and city maps for urban information systems, mapping out biotopes, or administrative domains for environmental planning. A Munich city guide is already completed in most parts.
2. MultiMED. This application deals with multimedia processing of X-ray images in medicine, including detail images and verbal or written medical reports. A prototype has been developed in collaboration with the St. Bernward hospital in Hildesheim, Germany (Specht and Bauer, 1995).
3. MultiBHT. A third field of application lies in linguistics, in multimedia processing of results of language analysis, in order to construct correct grammar and the development of text-critical editions. An application for Old-Hebrew exists in the Institute for Assyriology and Hethitology of the University of Munich.
4. MultiLIB is a multimedia guide through the university library of the University of Munich and its branches. The purpose is to enable the students to find books, their location and access rights, opening times of the library, and to offer support in catalogue queries. Therefore the system must be accessible from all the branches.

In its different applications, the system must prove to be worthwhile in very variable user environments: MultiLIB is directed to the broad public, especially to students, and most applications are read-only. MultiBHT is an application in research. Here there are only a few users, but with intensive sessions, large result sets and almost every access is also a write-access. Thus there are peak loads in both directions. In addition, extraction of multimedia objects like images, text, sound and blinking requires higher CPU and net usage as in conventional databases. Current research is ongoing in setting up an efficient WWW-interface on MultiMAP, with an client implementation in JAVA.

7. Conclusion. We have presented the Dexter reference model and its conversion into a relational database, using entity-relationship modelling as an intermediate step. A complete Dexter hypermedia engine can run on the obtained relational schema. The most important (the most frequent) operation is the link navigation. Therefore we have undertaken a detailed analysis of its complexity and have presented variants of optimization. The best result we reached for the complexity for link navigation was $O(\log(|\text{links}|) + |\text{links}| * \log(|\text{anchors}|))$. Finally we have shown the formation of the multimedia database MultiMAP and its various applications. The system MultiMAP has been developed at the University of Technology, Munich.

REFERENCES

Halasz, F., and M. Schwartz (1994). The dexter hypertext reference model. *Communications of the ACM,* **37**(2), 30–39.

Hardmann, L., *et al.* (1994). The Amsterdam hypermedia model. *Communications of the ACM,* **37**(2), 50–62.

Khoshafian, S., and A. Baker (1996). *Multimedia and Imaging Databases.* Morgan Kaufmann Publishers.

Korkea-aho, M., and G. Specht (Eds.) (1996). *Trends in Multimedia Database Systems.* Helsinki University of Technology Press, Otaniemi.

Rumbaugh, J. (1991). *Object-Oriented Modelling and Design.* Prentice Hall.

Specht, G. (1996). *Foundations and Trends in Multimedia Database Systems.* Lecture Scriptum, Helsinki University of Technology.

Specht, G., and M. Bauer (1995). MultiMED – a multimedia databases system for education and tutoring in diagnosing X-ray pictures in orthopedics. In Schoop, Witt and Glowalla (Eds.), *Hypermedia in der Aus- und Weiterbildung.* UVK-Verlag, Konstanz. pp. 209–210 (in German).

Specht, G., and M. Hofmann (1996). Migration evaluation of a multimedia information system from a relational into an object-oriented database system. In Mayr (Ed.), *Beherrschung von Informationssystemen.* Oldenbourg-Verlag. pp. 233–251 (in German).

Received January 1997

G. Specht is a member of the research staff of computer science at the Technische Universität München, Germany. His research interests include multimedia databases, deductive databases, object-oriented programming systems and natural language parsing. Dr. Günther Specht received his Ph. D. in 1992. Since 1993 he leads the multimedia database project MultiMAP, now founded by the Germany Research Network (DFN Association). He teaches graduate cources on multimedia database systems, deductive and object-oriented database system. He is author of several international articles and two German books.

RYŠIŲ NAVIGACIJOS SUDĖTINGUMO ANALIZĖ DEKSTERIO ETALONINIU MODELIU GRINDŽIAMOSE HIPERMEDIA DUOMENŲ SISTEMOSE

Giunteris ŠPECHTAS

Šiandien multimedia ir hipermedia sistemose naudojama tiek daug duomenų ir ryšių, kad juos tenka saugoti duomenų bazėse. Tam reikalingos duomenų bazių valdymo sistemos su efektyviai realizuojamomis didelės raiškos gebos schemomis. Vienu iš plačiausiai pripažintų hiperduomenų modeliavimo būdų yra Deksterio etaloninis hiperteksto modelis. Jame naudojama galinga mazgų ir ryšių modeliavimo technika. Šis modelis yra plačiai pripažintas. Straipsnyje pasiūlyta, kaip pažingsniui pertvarkyti tokį modelį į reliacinę multimedia duomenų bazės schemą. Šitaip gautai hipermedia mašinai svarbiausia ir kritiškiausia laiko požiūriu yra ryšių navigavino operacija. Straipsnyje detaliai analizuojamas tos operacijos sudėtingumas ir siūloma, kaip reikia patobulinti schemą, kad tą operaciją optimizuoti. Straipsnio pabaigoje aprašyta Miuncheno technikos universitete sukurta sistema MultiMAP, kurioje efektyviai įgyvendintos straipsnyje siūlomos idėjos.