Storage Management for Knowledge Bases

Thodoros Topaloglou
e-mail:thodoros@cs.toronto.edu
Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A4

Abstract

Secondary memory storage plays a major role in making large knowledge bases usable. This paper presents such a storage architecture for knowledge bases. In particular, the Controlled Decomposition Model, a flexible storage model which takes into account the numerous and expressive features of the knowledge representation model is defined. Second, the indexing problem for the knowledge base context is examined and the Temporal Join Index is proposed. An implementation technique for temporal indices, based on a spatial access method, is presented. Finally, an analytical and experimental performance study is conducted to account the performance limits of the proposed methods.

1 Introduction

The performance of knowledge base systems slows down significantly as the size of the employed knowledge base increases, if the knowledge base is stored in primary memory. Performance deterioration is even worse when the size of the knowledge base grows much beyond the size of the available main memory space, due to increased frequency of page faults when accessing the knowledge base. Moreover, demand for larger knowledge bases will continue to increase as expert systems and related technologies find more applications in areas such as software engineering [14], control systems [35], etc. Unfortunately, most of the existing knowledge base building tools such as expert system shells, Lisp and Prolog, will not support the development of large knowledge bases, precisely for the reasons given above.

A critical point for enabling knowledge base system technology that can deal with large knowledge bases, such as those referred in [19], [25], is to provide management tools whose performance will be independent of the size of the knowledge base. This paper, concentrates on the problem of storage management for knowledge bases. By storage management we mean a system's component which is responsible for storing and retrieving knowledge under conditions of predicted performance.

[0] *Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or special permission.*

Proceedings of the ISMM/ACM Second International Conference on Information and Knowledge Management (CIKM'93)

The most simple-minded storage management scheme for knowledge base systems, is the coupling with an external database [13]. Among the disadvantages of this approach is that it involves two separate systems which support mismatching representations models, connected with a high level communication. Instead, we propose an integrated system assuming a secondary memory based storage architecture. In the presence of such an architecture, the primary factor of achieving good performance is the minimization of disk accesses during knowledge retrieval.

The design of a storage model for knowledge bases is influenced by the variety of features of the underlying knowledge representation language. The approach presented here is closely tied to Telos [23] but our results are applicable to a class of knowledge representation languages with similar features, such as *classification, generalization, aggregation* and *time*. This work is part of a broader effort in providing generic management tools for efficient and robust implementations of large knowledge base systems [24].

The organization of the paper is as follows: Section 2 contains a quick overview of Telos and introduces an example to be used throughout. Section 3 summarizes related work from databases and addresses the problem to knowledge bases. Section 4, presents a logical storage model for knowledge bases. Section 5, discusses index organizations for temporal knowledge bases. Section 6, develops a cost model for studying the performance of the above methods and section 7 closes this paper with concluding remarks and an outlook for future research.

2 Highlights of Telos

Telos [23] offers a representational framework which includes structuring mechanisms analogous to those offered by semantic networks and semantic data models. It also provides an assertional sublanguage for the expression of deductive rules and integrity constraints. Two novel aspect of Telos are the treatment of attributes as first-class citizens and the provision of special representational and inferential capabilities for temporal knowledge. Structural and assertional knowledge is organized along *instantiation, generalization* and *aggregation* dimensions. Telos adopts Allen's[1] interval-based time framework for representing possible histories of generic events (historical time) and activities, as well as the system's knowledge of these histories (belief time).

An example Telos knowledge base, in the form of a semantic network is shown on figure 1. Dashed lines represent IsA relationships between class, shown in bold font, whereas solid lines represent attribute relationships. This knowledge base is used as a working example throughout

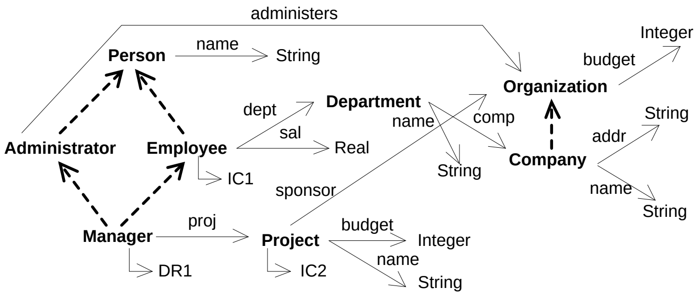

Figure 1: The employee/departments knowledge base

this paper.

3 From Databases to Knowledge Bases

The design of a storage model for knowledge bases is influenced by the features of the underlying knowledge representation language. Many of them have been studied separately in the context of structurally object-oriented databases, temporal databases and deductive databases. The increased difficulty of the problem in the context of knowledges bases is due to coexistence of more than one of these features.

Kim et al. [17] reported two impacts of the IsA relationships in object instantiation with data management consequences. The first is the increase of the access scope of classes i.e., if a query is targeted against class C all instances of its subclasses must be considered. The second impact is that the domain of an attribute whose values are instances of a class, becomes broader if this class has subclasses. Both of them influence the way in which instances of classes are stored and indexed in the presence of IsA hierarchies. Some solutions to this problem have been proposed in the fields of object-oriented databases [18], and semantic data models [26], [5].

The aggregation relationships suggest the way that the components of objects are structured together. Two techniques have been proposed for the implementation of aggregation in object-oriented databases: The use of object identifiers (OIDs) [15], [33], [16], [22] and the use of nested structures [27], [9], [8].

The access patterns with regards the temporal knowledge, stored in a knowledge base, define additional requirements for the logical and physical organization of the knowledge base storage.

Deductive rules and integrity constraints define logical connections between parts of the knowledge base. These connections need to be traversed any time that a rule or constraint is evaluated. Therefore, storing together the parts of the knowledge base that are accessed together results in fewer disk accesses.

The type and the frequency of the processed queries is yet another influential parameter of the problem. Appropriate design of the storage structures can therefore favor the evaluation of certain classes of queries. Workloads and knowledge base statistics are helpful in deciding among storage alternatives.

The problem being studied in this paper is stated as follows: Given a description of the knowledge base structure, derive a logical and a physical storage design. For the mapping of the object schema to a logical database design we developed the controlled decomposition algorithm which is motivated by the Controlled Decomposition storage Model (CDM). Physical data structures and techniques for implementing CDM are also developed and evaluated.

Our proposal towards storage management for knowledge bases is motivated by the fact that we do not intend to reinvent the data storage techniques but we are rather interested in extending them for the knowledge base context. Further, our position is that a revolutionary approach is hard to gain ground from the already commercially established database technology.

4 The Controlled Decomposition Storage Model

4.1 Background

The controlled decomposition storage model (CDM) is a logical storage model that we are proposing for knowledge bases. It is a flexible combination of the direct (n-ary) storage model (NSM) and the decomposition storage model (DSM) [7], [33].

The DSM has been proposed as an implementation technique for relational and complex object databases. It is based on the principle of *vertical partitioning*, i.e. for each attribute of a class it defines a separate relation. As a result, class instances have their components (i.e., values of their attributes) distributed over many relations/files. Object identifiers (OIDs) serve as the connectives of different attribute values of the same object. Among the positive features of the DSM are that it is easily implemented and secondly it does not require storage reorganization when the *schema* changes (i.e., new classes are added/terminated, new attributes are added/terminated). DSM has also some disadvantages. It requires more storage (usually double) than typical non-decomposision storage systems. In addition, operations accessing many attributes of objects are inefficient. However, the most important disadvantage of the model is that it does not benefit applications that process objects as a whole (e.g., CAD/CAM) since updates and retrievals of many attributes of an object or of an entire complex object are inefficient.

In the direct (n-ary) storage model, an object with n attributes, including inherited ones, is implemented by a record with $n+1$ fields: one field for the OID and a field for each attribute. If the value of an attribute of an object is another object (or set of objects), then two variations of the NSM are possible. In the first, the entire object (resp. set of objects) is stored directly as the attribute value. In the second variation an OID (resp. set of OIDs) is stored instead of the entire object. In the first variation it is very likely to store redundant information. In order to handle IsA relationships NSM performs *horizontal partitioning* i.e., splits an IsA hierarchy into a number of files, one for each class, whose record structure consists of an inherited part and a class specific part. This technique may result in many and small files when the IsA hierarchies grow large. The advantage of NSM is that it maintains many (relational) joins materialized and consequently it efficiently supports the accessing of a large number of attributes or subobjects of an object. Among its disadvantages are its inflexibility on schema

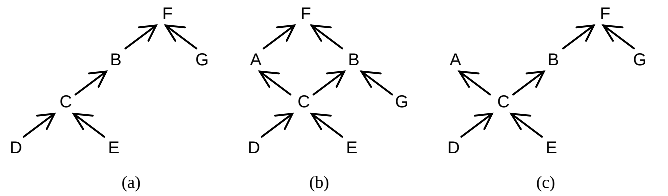

Figure 2: IsA hierarchies

updates and its poor performance on subobject updates.

4.2 Controlled Decomposition

The CDM offers a middle-of-the-road approach enjoying many of the benefits of both models. It performs a combination of vertical and horizontal partitioning which depends on the structure of the IsA hierarchies, and on knowledge base statistics. Additional benefits of the method are expected if information extracted from the access paths (queries, rules, constraints) is used to guide the decomposition. In the following, we outline the controlled decomposition partitions an object schema to a logical database schema.

• **Meta Level.**
For the "meta-class" level [1] of Telos we create a fixed number of relations. These relations hold the following properties: (a) they have known access paths, (b) can grow very large and (c) are updated very often.

• **Simple Class Level:**
As a first step, we resolve problems related to the IsA relationships such as multiple inheritance. The structure of IsA hierarchies determines the degree of horizontal and vertical partitioning that is performed. The method works for IsA hierarchies which are trees (figure 2.a). Graph-shaped hierarchies are treated as follows:

Graphs with multiple paths (figure 2.b): All class instances have a common most general class, whose structure is inherited to all of its subclasses. A particular problem that may appear is that of multiple inheritance, i.e. class `C` inherits an attribute with the same name and conflicting types from `A` and `B`. This situation is resolved by renaming the inherited attribute. Also the generation of the hierarchical class identifier becomes tricky.

DAG (figure 2.c): In this case there are classes belonging to two or more different IsA hierarchies and the question is under which of them the instances of the common classes must be stored. The options are to either disconnect the one or the other link (see class `C` in figure 2.c) or both of them. That will affect the cardinality and the record format of the resulting relations. When disconnecting, class `C` is rewritten as `C'` and the attributes it inherits through the disconnected link are made explicit to it. The decision of how to partition the IsA graph (and reformulate it into trees) depends on the available statistics and information about access paths. The empirical heuristic that we employ is to attach the "disputed" class to the hierarchy that is queried (or used in rules and constraints) more often.

After having reformed IsA hierarchies into trees, the root class of each tree becomes a candidate of being a storage relation. A class identifier is generated for the root and all other classes receive identifiers relative to it. Then, the following CDM rules are applied:

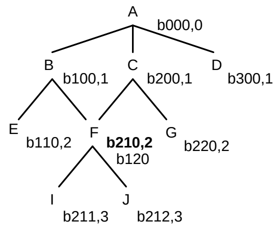

Figure 3: Hierarchical class identifier

1. Tokens[2] are stored at their most general class.
2. Subclasses store locally only their specialized attributes.
3. An object identifier's (OID) field is created for each relation. The values for this field are system generated hierarchical identifiers which are used as logical references for the tokens.
4. Attributes of classes with primitive domains (e.g., `Integer, String`,..) are stored in the same relation with the class instance information.
5. Attributes of classes with complex domains (e.g., classes) are stored as join relations.

Rule number 4 may occasionally be violated. Specifically, attributes that are accessed very often – such information is extracted from the access path analysis – and also have a large number of distinct values are candidates to form separate relations. Records of the same class instance that span over more than one relation, have the same OID value.

The temporal dimensions of Telos relationships is accommodated in the created database relations through an attribute versioning scheme. When a relation scheme `[RNAME,ATT1,.., ATTN]` is decided by the CDM, the following relation is created:
$\mathtt{RNAME(IN(OID,HT,BT),ATT_1(V,HT,BT),..,ATT_N(V,HT,BT))}$.

Join relations also involve time. A join relation between classes A and B will be:

`A_B(A_OID,B_OID,HT,BT).`

In the discussion that follows, we limit ourselves to history time only.

• **Token Level / Hierarchical Identifier:**
Token objects receive a unique hierarchical identifier. The hierarchical identifier determines the position of the object within an IsA hierarchy. The construction of the hierarchical identifier is based on a numbering of classes in a way which contains information about the parenthood relation within the hierarchy. The *class_number* of `C` consists of its depth first number, *dfs(*`C`*)*, and its level, *lv(*`C`*)*, within the hierarchy. Such a numbering provides information about the descendant and sibling relationships within the hierarchy which are used in query processing.

If a class has more than one parents (figure 3), then a number according to a single parent must be assigned.

[1] The "meta" level of a Telos knowledge base consists of all those system or user defined concepts which have classes as instances.

[2] In Telos, the term token stands for the instances of simple classes, i.e., those whose instances are individual objects

In such case the principle of most preferable parent link is applied before the numbering. The prioritization of the parenthood of a class uses information about the access paths. The parent whose instances or its attributes are used more often in rules and constrains and in frequently asked queries becomes the primarily preferred parent. Class `F`, in figure 3, is an example of numbering after prioritizing the parenthood. The class numbers are stored in a structure called *hierarchy directory*. The hierarchy directory, one for each hierarchy, is main-memory resident structure and is used when querying or updating the hierarchy.

The *token identifier* is the concatenation of a unique for each hierarchy number, called *base*, the class-number and the value of its hierarchy's instance counter. The produced identifier is logical in the sense that does not say anything about physical location of the token. As matter of fact, the class numbering helps to linearize the instances of a hierarchy for addressing purposes. For instance, the query *all* `C`'s in figure 3 addresses all tokens whose identifier is prefixed by `b2`.

4.3 Example

Assume the example knowledge base, in the form of a semantic network, of figure 1. The storage relations which are generated by the CDM algorithm for part of this knowledge base are shown on figure 4. As shown, the `EMPLOYEE` relation stores only the local attribute (e.g., `SAL`) while the employee name is stored in the `PERSON` relation. The identifiers of employees have as a prefix `P2` denoting that employees are also persons. The only-persons have identifiers starting with `P0`. [3]

4.4 Physical Storage

Each relation generated by the CDM algorithm is stored in a separate file. Now, each relation tuple (file record) receives a physical tuple identifier. This is a physical address made of the page and the slot number (e.g., `tid`=⟨page#, slot#⟩) where the tuple is stored. A record addressing index (RAI) is responsible for materializing the token given its logical identifier and the temporal interval of interest (e.g., ⟨*oid, time*⟩). It returns a list of physical addresses where fragments of the given token are stored for the specified time interval; then, the corresponding pages are fetched. RAI should be understood as a multidimensional access method. A separate RAI is maintained for each IsA hierarchy.

The available index structures include the temporal join index (TJI), and the simple temporal index (STI). Both of them are implemented as spatial access methods. TJI which appears to be the most interesting, is discussed in detail in section (sec. 5).

The STI is used on attributes with primitive domains. Its searching key is a pair of a domain value and a time segment i.e., ⟨*value, time*⟩ and the returned value is a list of qualified token identifiers. The STI is defined over all the classes of an IsA tree which have the indexed attribute defined, i.e., it is a hierarchical index [17]. The purpose of such an index is to return qualified (logical) identifiers on a selection on the indexed attribute. The placement of the appropriate STIs is subject of an index selection algorithm as those proposed in databases [11],

[3]For the simplicity of the presentation we use a discrete time domain for the temporal fields.

PERSON

P_ID		ENAME	
P31	1..6	Alex	–
P12	2..10	Helen	–
P03	4..8	Mary	–
P24	8..10	Mike	–
P25	6..9	Nick	–
P26	5..9	Tony	–
P37	6..10	Karen	–
P28	5..10	John	–

EMPLOYEE (IsA PERSON)

E_ID		SAL	
P31	1..6	{45k	1..4,
		55k	5..6}
P24	8..10	{60k	8..10}
P25	6..9	{55k	6..9}
P26	5..9	{50k	5..9}
P37	6..10	{60k	6..8}
P28	5..10	{45k	5..8,
		60k	9..10}

DEPARTMENT

D_ID		DNAME	
D1	1..6	{Res	1..4,
		Sales	5..6}
D2	1..10	{Adm	1..10}
D3	4..10	{Dev	4..10}
D4	4..8	{Res	4..8}
D5	4.10	{Adm	4..10}
D6	9..10	{Res	9..10}
D7	9..10	{Dev	9..10}

ORGANIZATION

O_ID		BUDGET	
O05	3..9	{30M	3..4,
		60M	5..9}

COMPANY (IsA ORGANIZATION)

C_ID		CNAME		CADDR	
O11	1..10	{ML	1..8,	{Toronto	1..10}
		MA	9..10}		
O12	4..10	{KC	6..10}	{Toronto	4..8
				Paris	9..10}

EMP_DEPT

E_ID	D_ID	TIME
P31	D1	1..6
P34	D2	6..8
P34	D3	9..10
P25	D3	6..9
P26	D4	5..9
P37	D5	6..8
P37	D6	9..10
P28	D7	9..10

DEPT_COMP

D_ID	C_ID	TIME
D1	O11	1..6
D2	O11	1..10
D3	O11	4..8
D3	O12	9..10
D4	O12	4..10
D5	O12	4..10
D6	O13	9..10
D7	O14	9..10

Figure 4: Relations created by the CDM

[12]. However, a solution for knowledge bases needs to be revised by taking into account the semantic information contained in the rules and the constrains. So far, we attach an index to frequently queried attributes with substantial domain cardinality, e.g., `name` and `salary`.

For the implementation of the physical layer we can utilize an existing database storage kernel (EXODUS [4], DASDBS [27] and EOS [3] are considered as candidates).

5 Temporal Indices

5.1 The Temporal Join Index

Given two classes $C1$ and $C2$ and a time-dependent attribute, *attr*, of $C1$ which takes as values instances of $C2$, a *temporal join index relation* (TJI hereafter) is defined as:

$TJI = \{\langle oid(r), oid(s), t\rangle | r \in C1, s \in C2, s \in r.attr \text{ [at } t]\}$

A TJI is used to describe a time-dependent relationship and, in addition, to provide fast access to the instances of the joined classes. The relations `DEPT_COMP` and `EMP_DEPT` on figure 4 are examples of temporal join relations implemented as TJIs.

The searching key in a TJI organization is a pair of an object identifier (or range of OIDs) and a temporal

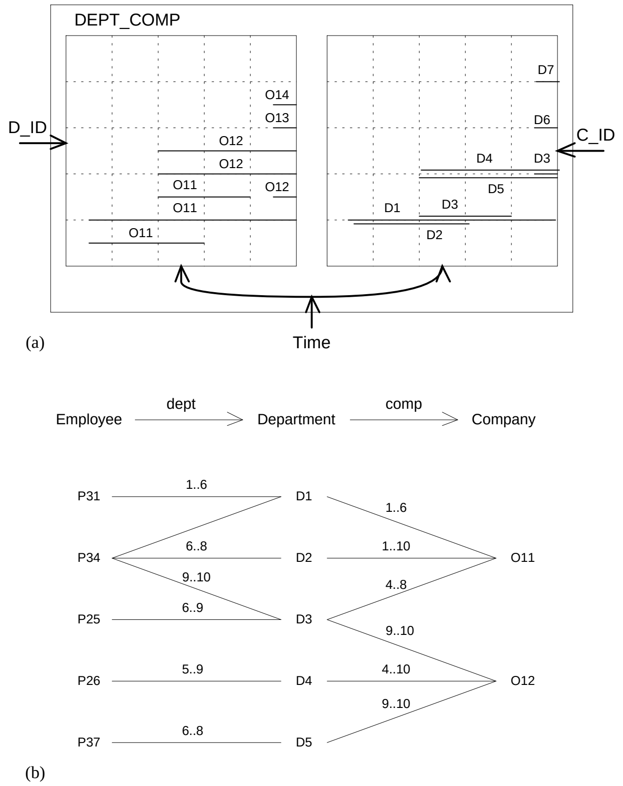

Figure 5: (a) Temporal join index, (b) Path index

interval, i.e., $\langle OID,\ time \rangle$, such that the object identifiers of instances of the second class which are in a TJI tuple with *OID* and for a time interval overlapping with *time* are returned.

There are many good properties that TJI possesses. First, it performs two types of selections, a temporal selection and a selection on the OIDs, at the same time, i.e., it is a single index for both uses. Second, it can be accessed in both directions (see figure 5a). Finally, sequence of TJIs can be used to implement a temporal index over paths attributes (see figure 5b) thus forming a temporal multi join index organization (section 5.2).

If a class which participates in a join relation has subclasses, then the instances of its subclasses are joined with the second class instances in a single index. For instance, in the join relation `EMP_DEPT` both only-employees and manager-employees are joined with departments. Essentially, the index is established between hierarchies of classes rather than between single classes. The reason for doing this is twofold: First, according to the CDM, tokens are stored at their more general class, therefore the subclasses' extensions (e.g., Manager) become explicit at their superclasses (e.g., Employee). Second, a result by Kim et al [17] insists that a hierarchical index always outperforms the simple class index.

The tuples of a TJI relation have fixed size because they consist of two object identifier fields and a temporal identifier field. The length of TJI can grow large; because of the temporal nature of the Telos attributes, all updates are implemented with an "add-only" policy, i.e., updates limit the temporal interval of a tuple and further introduce a similar tuple with a new temporal interval. The possibly very long length of a TJI suggests a secondary storage implementation of it. Similarly to the join index [34], we decided to store two copies of the TJI (see also figure 5) one clustered on OID1 and one clustered on OID2, making thus access in both directions equally efficient. Each copy of the TJI is implemented by a spatial data structure. The reason for choosing a spatial data structure is that the indexed values are not just points on a one dimensional value space but line segments on a two dimensional space where the time component specifies the lifetime of the joined identifiers.

5.2 Indices on Paths

The complex object-valued attributes may lead to deeply nested object compositions (see the Employee→ Department → Company chain in figure 1). Selections on nested-attributes or path queries are very common in Telos. In object-oriented databases, three different indexing methods were proposed for the very same problem [2]: the *nested index*, the *path index* and the *multi-index*. The role of time in attribute relationships make all the above indexing methods inapplicable in Telos case. Figure 5b shows the way in which identifiers of company instances are connected to identifiers of employees. The nested index over the path attribute dept.comp of class Employee, whose domain is the extension of class Company, will associate OIDs of the latter with sets of OIDs of the former class, i.e.,
$\{(O11, \{P31, P34, P25\}), (O12, \{P34, P25, P26, P37\})\}$
This organization does not verify the availability of the connections between OIDs from the temporal viewpoint. For example, the version of the above index for the interval `1..4` is different than the one for the interval `9..10`. In order for the index to be valid over the entire history of the application, one must keep all its versions. That implies an obvious overhead on the storage utilization and the update cost.

The path index on the same example would record the entire paths from the end class's OID to the beginning class's OID, i.e.,

$$\{(O11, \{D1[1..6].P31[1..6],\ D2[1..10].P34[6..8], ...\}),$$
$$(O22, \{D3[9..10].P34[9..10], ...\})\}$$

Storing the entire path, it is easier to verify the temporal availability of the connection, especially when the validity time of the OID connections is stored. The path index has better response time than the nested index but much higher storage cost which can be even worst when the degree of sharing[4] is high.

The multi index maintains the steps of the path separately, i.e., an index for attribute comp of class Department, and an index for dept of Employee. Each step of the path index can be seen as a TJI. A sequence of TJIs can establish an index between a source and a destination class which in addition includes the temporal component of the attribute relationships (figure 5b). By adopting TJI as the step index of the multi-index, we derive the *M(ulti, temporal) J(oin) I(ndex)* (MJI).

Formally, MJI is defined as follows: Assume a nested attribute $a_1.a_2...a_{n-1}$ between classes C_1 and C_n. A MJI between C_1 and C_n consists of an a TJI between

[4] Assume an attribute $attr$ from class A to class B, i.e., $A \overset{attr}{\rightarrow} B$. Given an instance of class B, the *degree of sharing* is the average number of referencing instances of A that share the same reference through the $attr$ relationship.

C	number of classes in $\mathcal{H}$
a	number of attributes defined at the root class
p	total num. of attr. in $\mathcal{H}$ having primitive domain
c	total num. of attr. in $\mathcal{H}$ having complex domain
l	avg num. of locally defined attr. per each class
$I_{\mathcal{H}}$	total number of instances in $\mathcal{H}$
I_C	avg num. of instances per class ($I_C = I_{\mathcal{H}}/C$)
b	branching factor of the IsA tree
h	height of the IsA tree

Table 1: Knowledge model parameters

C_{n-1} and C_n on attribute a_{n-1} and a MJI between C_1 and C_{n-1} on the path $a_1.a_2...a_{n-2}$. In the above example the MJI(Employee, Company, dept.comp) consists of TJI(Employee, Department, comp) and TJI(Department, Company, comp). In fact, if a MJI over a path is available, indices on any intermediate subpath are available as well. In the query "find all employees who worked for `'KCC'` between time 6 and 8", the processing steps using the MJI are the following: First, (using or not an index on comp.cname) we evaluate the predicate comp.cname=`'KCC', 6..8` and get the identifier of the qualified company which is $O12$. Then, by accessing in the reverse direction the `DEPT_COMP` TJI we obtain the qualified department identifiers $\{D3, D4, D5\}$ for time 6..8. and lastly, using `EMP_DEPT` again in the reverse direction we obtain the identifiers of the requested employees, $\{P25, P26, P37\}$ for the time interval 6..8. Note that the temporal filtering rejected employee $P34$ who worked at a department of `'KCC'` at 9..10. Exactly the same index can be used to determine the company of a given employee if it is accessed in the forward direction.

6 Performance Analysis

The aim in this section is first to prove the viability of the methods defined above, by studying their performance bounds, second, to extent performance results from databases, and finally, to provide a cost model for estimating performance in knowledge bases.

6.1 Storage Cost

Let us consider a tree structured IsA hierarchy $\mathcal{H}$. Table 1 summarizes the parameters which characterize $\mathcal{H}$.

The total number of attributes in $\mathcal{H}$ is $a + (C-1)l = p + c$. By also assuming that the size of an object identifier, a temporal identifier and average size of an attribute's value are i, t and v bytes, respectively, then we can compute the *required disk storage* (*rds*) for the previously defined storage models.

NSM stores one relations per class with its record consisting of $n+1$ fields, where n is the sum of the number of the locally defined attributes plus the number of the inherited attributes. DSM stores one relation per attribute which amounts to $a + \sum_{k=1}^{h} b^k l$ relations in which the number of tuples changes as a function of the attribute's depth in the tree structure. The total number of DSM tuples is computed as $N = aI_H + \sum_{k=1}^{h}(b^k l)(1 + \sum_{j=k+1}^{h} b^j)I_C$, where $p' = p/(p+c)$ portion of them belongs to regular relations and $c' = c/(p+c)$ belongs to join relations. The following equations show the required storage of NSM and DSM respectively.

$$\text{rds(NSM)} = \sum_{k=0}^{h}[i + (a+lk)v + (1+a+lk)t]b^k I_C \quad (1)$$

$$\text{rds(DSM)} = p'(v+i+t)N + 2c'(2i+t)N \quad (2)$$

Note that equations 1 and 2 modify the formulas for NSM and DSM storage of [7], taking into account attributes defined over IsA hierarchies.

In CDM, the degree of decomposition depends on (a) the average enumber of attributes with complex domains (parameter c') and (b) the average number of attributes that are introduced at the subclasses (parameter l). The root class is decomposed to one relation with $p'a$ primitive attributes and $c'a$ join relations with $I_{\mathcal{H}}$ tuples each, i.e.,

$$\text{rds(Root)} = [(1+2c'a)i + p'av + (1+p'a+2c'a)t]I_{\mathcal{H}}$$

The storage cost for the subclasses is given by the equation:

$$\text{rds(subClass)} = \sum_{k=1}^{h}[(i + p'lv + (1+p'l)t) + 2c'l(2i+t)]b^k(1 + \sum_{j=k+1}^{h} b^j)I_C$$

where the first term in the brackets stands for the relation per class which stores the primitive attribute part and the second term denotes the resulted join relations. The aggregate cost will be:

$$\text{rds(CDM)} = rds(Root) + rds(subClass) \quad (3)$$

In figure 6 we plot the storage cost curves of DSM and CDM relatively to NSM. The employed IsA hierarchy has C=13 classes and I_C=1,000 tokens per class. The rest of the parameters are set as follows: i=8, t=4, v=20, b=2. In the experiments we set the number of attributes of the root class and the number of attributes with complex domain and we observe the behaviour of the three storage models as the number local attributes per subclass increases. We considered IsA trees up to height of 4. The storage cost of the CDM and the NSM is computed relatively to the NSM cost. These results obtained verify the claim of [7] that DSM requires 2 to 4 times more data storage than NSM. After a large number of experiments we found that CDM needs in average 65% of the required space of DSM.

6.2 Update Cost

Among the design criteria for the CDM is its performance on updates. This is desirable because, in a knowledge base environment, both the classes and the tokens are expected to be updated very often. The most frequent types of updates in Telos include:

1. Add a new attribute to a class: This is done incrementally to the existing storage organization by creating a new relation:

`CLASS_NEWATTR(C_ID,NEW_ATTR).`

and also update the schema description relation. Note, that the followed tactic here is decomposition – vertical partitioning.

2. Add a subclass/superclass: First, the new class is receiving a *class_number* and the respective IsA hierarchy directory is updated appropriately. Then, a new relation with $n+1$ fields is created; n is the number of the specialized attributes defined at the new class with primitive attribute domains, only. Those with complex domains form additional join relations.

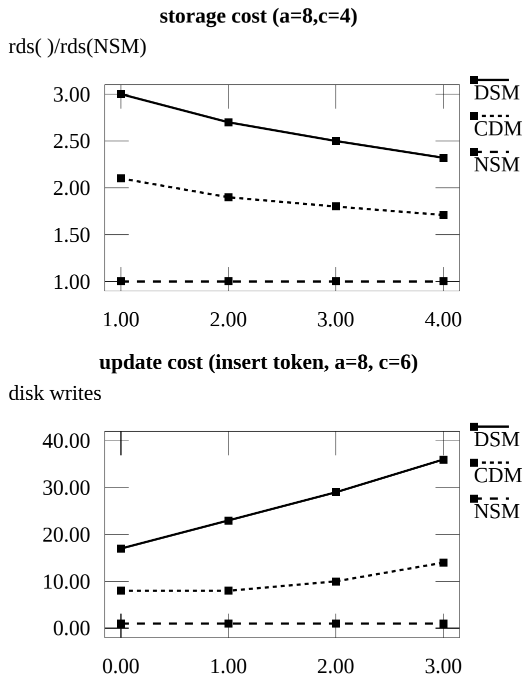

Figure 6: Storage and update cost

3. Insert a new token: Usually a token's insertion results in more than one record updates. In NSM this would be a single tuple insertion which corresponds to 1 disk access. In DSM, it would require $1+2a$ writes, where a is the number of attributes which are applicable to the class where the token belongs to. The cost in CDM is the $l/(a+l)+c/(p+c)$ fraction of DSM cost. On top of this cost the update cost of the associated indices must be added.
4. Update the value of an attribute: First, the relevant storage record is located by accessing an index. Then, the block which contains that record is written and the attribute's index is updated. For the DSM this is an average of three writes [15] and so it is for CDM. Appropriate buffering policy may further minimize that cost.

In summary, CDM performs as good as DSM on attribute-value and schema updates, and wins DSM on token updates. CDM is weaker than NSM on token updates, but a severe drawback of NSM is that it cannot handle schema updates at all.

6.3 Query Cost

In this section we define a cost model to study the performance of CDM on various types of retrievals using the available index structures.

For the scope of this analysis, we consider only three types of access level operations: relation scan, indexed access, and time-indexed access. The performance metric that we study is the number of page accesses during retrievals. This analysis focuses on the performance implications of time which the central feature in our methods.

Table 2 summarizes the data-dependent parameters. The cardinality of an attribute relationship is related with the cardinalities of the participant classes through the equation: $D_{att_A} = m \times r \times |A| = m \times r \times d \times |B|$,

$\lvert C\rvert$	cardinality of a class C (including subclasses)
V_{attr}	num. of unique values for attribute $attr$
D_{att_C}	num. of recorded attributes att at class C
H	length of knowledge base's history
d	degree of reference of $attr$ from class A to B
m	multivaluedness: #values per attr. occurrence
r	temporal repititions: #versions per attr. occ.
g	average length (lifetime) of a temporal value
T_q	query time
σ	selectivity of an operation
τ	temporal selectivity

Table 2: Data parameters

N	number of tuples in the indexed file
F	average number of entries per index node
$size(x)$	size of x in bytes
u	average node occupancy
B	effective storage capacity of a page
h	height of the index tree

Table 3: Index parameters

where $d = |A|/|B|$. The product $m \times r \times d$ quantifies the degree of sharing in [2]. Without restriction of generality, we assume that multivaluedness is caused only by the time, i.e. m=1.

The index access operations make use of the indices defined in section 5. For the non-temporal index we assume a B-tree like organization whereas the temporal indices assume an R-tree one as has already been described. Table 3 summarizes the index parameters. In the rest, these parameters are used with an R or a B subscript denoting R-tree and B-tree type indices respectively. The searching key for an R-tree (R) index is a pair $\langle segment, value\rangle$ whereas for a B-tree is just $value$.

The type of queries studied bellow are temporal selection on a direct primitive attribute (query 1) and on a nested attribute, i.e., path query (query 2).

Selectivity and temporal selectivity
The selectivity of a non-temporal qualification is estimated as in [29] [5]. In temporal attributes, the value space is two-dimensional. Obviously, the distribution of values in the two-dimensional space affects the performance and the size of the answer set of temporal queries. This effect is captured in our analysis through the parameter of *temporal selectivity*. When in addition to an attribute qualification, a qualification to its temporal component is applied, the selectivity σ is multiplied by the temporal selectivity τ. The estimation of τ is done in relation to the temporal distribution. Here we introduce the formulae of temporal selectivity for the sample distributions shown on figure 7.

- *distribution-1:* Uniform distribution of values without duration i.e., points. This is an easy case with selectivity:

$$\tau = \frac{T_q}{H}$$

- *distribution-2:* Uniform distribution of intervals of equal size g. According to the analysis of [10] on an R+-tree index for line segments, the number of selected

[5] The selectivity of the single-key query on an attribute with primitive domain V_a is $1/V_a$ and for the range query is $k_2 - k_1/k_{max} - k_{min}$, where k_{max} and k_{min} are the maximum and minimum key values in V_a and $k_1..k_2$ the searching range.

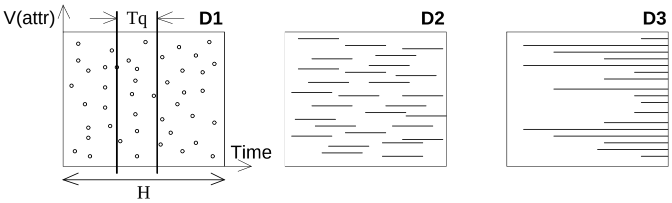

Figure 7: Temporal distributions

segments on a query segment T_q is:

$$\tau = \frac{g+T_q}{1+g}(N+1)$$

• *distribution-3:* Right-ended intervals. Such a distribution is possible in domains where a default rule of the type *"the salary of an employee is assumed to be the same as it was at her last promotion"*. Here, the size of intervals vary. Assuming that the beginning point of intervals is uniformly distributed over H and $T_q = |T_2 - T_1|$ is the querying time. The number of selected segments grows linearly as the endpoint T_2 approaches the right-end of H. Note that, selectivity in this case is independent of the length of the query segment and only depends on its right endpoint ($T_2 \leq H$), i.e.:

$$\tau = \frac{T_2}{H}$$

Query 1: temporal selection on a direct attribute (in IsA hierarchy). E.g.,

return `e : e/Employee : e.salary = 50K` **on** 1/90 .. 2/91

By varying the indexing options the following four access plans are received:

`Plan1.` Scan relation and filter out the qualifying tuples.

$$cost(P1) = cost(scan) + cost(filtering)$$

where filtering is a $O(N)$ operation, N is the number of attribute occurrences.

`Plan2.` Perform a selection using a non-temporal index and then filter out those answers with a qualifying time interval.[6]

$$cost(P2) = h_B + \sigma L_B + cost(filtering)$$

`Plan3.` Perform a selection on the temporal field only and then filter out the answers for which the non-temporal qualification holds.

$$cost(P3) = h_{R_1} + \tau L_{R_1} + cost(filtering)$$

`Plan4.` Use the temporal index. The temporal index, regardless its implementation, is a 2-dimensional access method which is able to select the set of tuples which satisfy two selection criteria (a temporal and a non-temporal one) in one step. The selectivity of the temporal index is $\sigma_T = \tau\sigma$.

$$cost(P4) = h_{R_2} + \sigma_T L_{R_2}$$

Query 2: temporal selection on a nested attribute (in IsA hierarchy): $C_1 \xrightarrow{a_1} C_2 \xrightarrow{a_2} D$. E.g.,

return `e : e/Employee : e.dept.name='res'` **on** 2/86 .. 8/89

The degree of sharing for attributes a_1 and a_2 are $s_1 = \frac{D_{a1}}{|C_1|} = d_1 m_1 r_1$ and $s_2 = \frac{V_{a2}}{|C_2|} = d_2 m_2 r_2$ respectively. We also assume that $m_1 = m_2 = 1$ and time is the only cause for multivaluedness.

Assuming the MJI type of index, one possible execution plan is to access first the index on a_2 and then exploit the index on a_1. The Telos access planner searches for a plan in a space of possible executions using machine learning techniques and exploiting past experience [32]. Here, we restrict ourselves to the simple heuristic that is to pick the plan which uses the available indices – the analysis of the previous query validates this choice.

Let, $Q(C_i)$ be the number of C_i identifiers returned when an access operation on C_i is executed. This number is used for calculation of the access cost on C_{i-1} and the process works recursively until i =1. The cost of the access plan for a path of length 2 for the single-key and range query[7], is:

step	parameter	single-key	range-query
idx on a_2:	$cost(TA_{a2})$	$h_2 + 1$	$h_2 + \sigma_T \times L$
	$Q(C_2)$	$d_2 r_2 \tau$	$d_2 r_2 \tau \lvert k_2 - k_1 \rvert$
idx on a_1:	$cost(a_1)$	$h_1 + \text{Yao}(Q(C_2), L, \lvert C_1 \rvert)$	

The aggregate cost for both steps becomes: [8]

$$cost(\text{query}) = cost(TA_{a2}) + cost(a_1)$$

We quantify the above obtained analytical results by conducting two experiments. In the first experiment, we apply query 1 on a binary relation – vertical partitioning is assumed – with three different implementations, namely, sequential file, clustered B-tree and clustered R-tree. The values of the index parameters are set as follows: N=60000, $size(page)$=4096 bytes, B=.9 $size(page)$, u=.8, $size(tuple)$=40, $size(pointer)$=8. The attribute values span over a history with length H=100. For the experiments we vary the selectivity (σ) of the query's non-temporal qualification and the temporal selectivity (τ). For distribution-2, the average lifetime is set g=10. The quantity measured is the number of page accesses for processing the query. The purpose of the experiment is to show the advantages of temporal indices in value-based queries. The results are summarized on the graphs of figure 8(a-b). As it shown, `plan1` and `plan2` are independent of the temporal distribution. The performance of `plan3` is also independent of the selectivity of the non-temporal qualification. `Plan4` which applies both a temporal and a non-temporal selection always outperforms all other plans.

The second experiment intends to show the use of temporal indices on path queries. The existence of the temporal index at the right-end of the path suggests to use `plan4` from the previous experiment. Then, random access is used to access the next step (in reverse order) of the path. At this point the second benefit of exploiting time indices is in effect. The selection on time results in more refined temporary answer set which is passed to the next step. As the curves on figure 8(c) show the cost of path queries is always lower if the temporal index is used.

The above results apply to CDM and DSM but do not apply to NSM. NSM does not support join (and subsequently temporal join) relations which benefit the path queries. Second, there is only one clustering possibility per relation in NSM which will favor queries on one of its attributes. The decomposition which takes place in CDM and DSM allows for more efficient data structures

[6] The cost of filtering is linear to the number of accessed tuples i.e., $selectivity \times index\_leaf\_pages \times \#tuples\_per\_page$.

[7] The search key is $\langle k_1..k_2, t_q \rangle$ and $|k_2 - k_1|$ denotes the number of distinct key values in the key range.

[8] Yao's formula Yao(m,p,N) [36] quantifies the number of pages accesses when m out of N randomly distributed on p pages records are accessed.

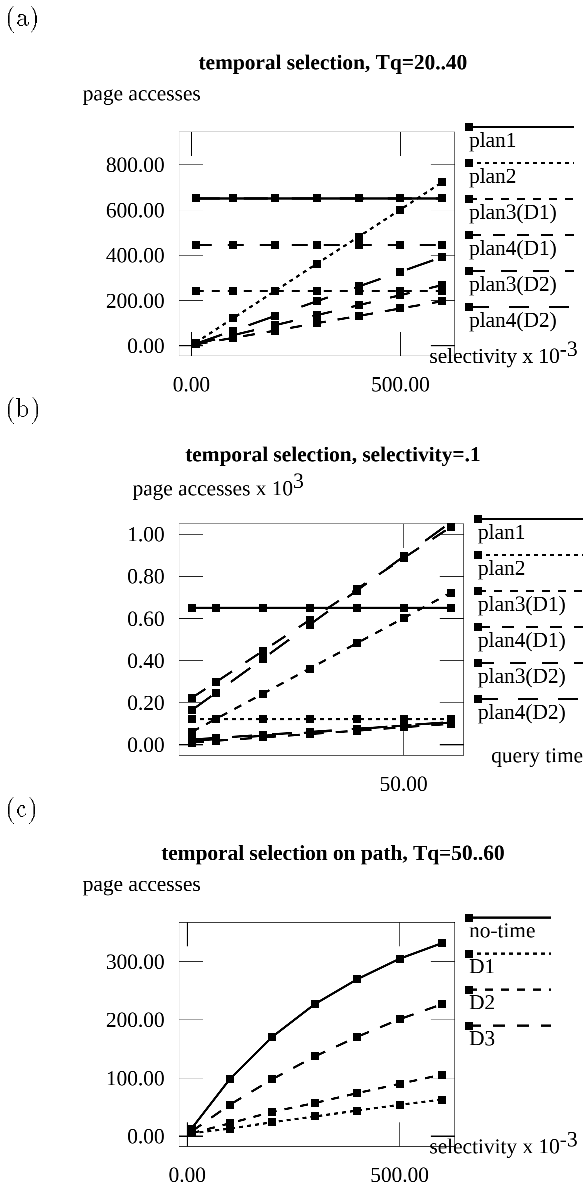

Figure 8: Query Cost

for associative retrievals and attribute queries. However, NSM provides better performance when an object as a whole is retrieved. For the same task, DSM demands a number of join operations to take place in order to combine the attribute components. The flexible indexing approach that we have taken in CDM saves us from computing these joins since the RAI index (described in section 4.4) materializes them. Another advantage that CDM possesses is its hierarchical object identifier. This construct helps for fast accesses within IsA hierarchies.

In summary, CDM is an attempt to combine the advantages of both DSM and NSM and to avoid their drawbacks. For knowledge bases where the cost of storage model has to include the storage, update and query cost, CDM is a viable solution. Integration of time, responsiveness to a broad range of retrievals and tunable performance are its very benefits.

7 Summary and Future Work

In this paper, we proposed techniques for the efficient storage management of knowledge bases. In particular, we defined the Controlled Decomposition Storage model and illustrated a method for deriving a database schema from the knowledge base structure. We discussed the indexing problem for knowledge bases and proposed an efficient index structure which accommodates time. Finally, we developed a cost model to analyze the behaviour of the proposed methods.

The position promoted throughout this paper which constitutes the Telos/KBMS approach towards knowledge management [24], is that robust and efficient generic tools for implementing knowledge bases are necessary. Extensions to database technology is a way to approach it. Although databases have been used in the past for the implementation of knowledge representation and similar technical problems have been discussed [20], [30], [13], most of the effort has been directed at the issue of coupling the two systems and not towards an integrated architecture. The Telos/KBMS approach reforms all the issues starting from the storage manager and the cost model and going to query processing and optimization [31], concurrency control [6], and integrity enforcement [28].

As for future research, there exist a number of issues which demand further work. In storage management, the adoption of a physical storage kernel can give us a more precise clue of the record placement problem and the potential clustering policies. We will also be able to verify through experimentation the claims that our analytical cost model suggests. The index selection problem is also of great importance, especially in the presence of rules and constraints which contain significant information about the way that the knowledge base is accessed. So far, our solution to this problem is based on empirical heuristics. The access method which is employed for translating from logical identifiers to physical addresses is another problem which needs attention.

Further optimizations to the TJI are also possible. A particular innovation is to define a nested form of TJI adapting the idea of the nested hierarchical index of Low et al [21]. Additional spatial access methods can be tested for the TJI implementation. The performance analysis must be extended to a wider range of index key distributions. The combined distribution of tokens across classes in an IsA hierarchy and across the history of knowledge base is yet another performance parameter to be studied. Query processing and optimization are currently studied for CDM in the presence of TJI [32].

Acknowledgements. The author wishes to thank Prof. John Mylopoulos for his valuable advice as well as Dimitris Plexousakis, Manolis Koubarakis and Adel Shrufi for their comments on the paper. Financial support was provided by the Department of Computer Science and the Ontario Information Technology Research Center.

References

[1] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *CACM*, 26(11):832–843, 1983.

[2] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge & Data Engineering*, 1(2):196–214, 1989.

[3] Alexandros Biliris. The Performance of Three Database Storage Structures for Managing Large Objects. In *Proc. of ACM/SIGMOD*, 1992.

[4] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proc. of 12th VLDB*, 91–100, 1986.

[5] A. Chan, S. Danberg, S. Fox, W.-T. Lin, A. Nori, and D. Ries. Storage and Access Structures to Support a Semantic Data Model. In *Proc. of 8th VLDB*, 308–316, 1982.

[6] V. Chaudhri, V. Hadzilacos, and J. Mylopoulos. Concurrency Control for Knowledge Bases. In Proc. of *KRR-92*, 1992.

[7] G.P. Copeland and S.N. Khoshafian. A Decomposition Storage Model. In *Proc. of ACM/SIGMOD*, 268–279, 1985.

[8] P. Dadam, K. Kuspert, F. Anderson, H. Blanken, R. Erbe, J. Gunauer, V. Lum, P. Pistor, G. Walch. A DBMS prototype to surt extended NF2-relations: An integrated view on flat tables and hierarchies. In *Proc. of ACM/SIGMOD* , 376–387, 1986.

[9] A. Deshpande. An Implementation for Nested Relational Databases. Technical Report 281, Computer Science Dept, Univ. of Indiana, PhD Thesis, 1989.

[10] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of Object Oriented Spatial Access Methods. In *Proc. of ACM/SIGMOD* , 426–439, 1987.

[11] S. Finkelstein, M. Schkolnick, P. Tiberio. Physical Databases Design for Relational Databases. *ACM Trans. on Database Systems*,13(1):91–128,1988.

[12] M. Frank, E. Omiecinski, and S. Navathe. Adaptive and Automated Index Selection in RDBMS. In *Proc. of EDBT Conf.* , 277–292, 1992.

[13] K. Higa, M. Morrison, J. Morrison, and O.L. Sheng. Object-Oriented Methodology for Knowledge Base/ Database Coupling. *CACM*, 35(6):99–113, 1992.

[14] M. Jarke, J. Mylopoulos, J. Schmidt, Y. Vassiliou. DAIDA: An Environment for Evolving Information Systems. *ACM Trans. on Office Systems*, 9(1), 1991.

[15] S.N. Khoshafian and G.P. Copeland. Object identity. In *Proc. of OOPSLA-86*, 406–416, 1986.

[16] W. Kim, H.-T. Chou, and J. Banerjee. Operations and Implementation of Complex Objects. In *Int. Conf. on Data Engineering*, 626–633, 1987.

[17] W. Kim, K.-C. Kim, and A. Dale. Indexing Techniques for Object-Oriented Databases. In *Object-Oriented Concepts, Databases and Applications*. ACM Press, 1989.

[18] Won Kim. *Introduction to Object-Oriented Databases*. The MIT Press, 1990.

[19] D. Lenat and R. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley, 1989.

[20] P. Lockemann, H.-H. Nagel, I. Walter. Databases for knowledge bases: empirical study of a knowledge base management system for a semantic net. *Data & Knowledge Engineering*, 7:115–154, 1991.

[21] C.C. Low, B.C. Ooi, and H. Lu. H-trees: A Dynamic Associative Search Index for OODB. In *SIGMOD-92*, 134–143, 1992.

[22] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *Int. Conf. on Data Engineering*, 171–181, 1986.

[23] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: A Language for Representing Knowledge About Information Systems. *ACM Transactions on Office Information Systems*, 8(4):325–362, 1990.

[24] J. Mylopoulos, V. Chaudhri, D. Plexousakis, T. Topaloglou. A Performance Oriented Approach to Knowledge Base Management. In *CIKM-92*, 68–75,1992.

[25] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, W. Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*,12(3):36–56,1991.

[26] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos, and M. Stanley. Implementation of a Compiler for a Semantic Data Model: Experiences with Taxis. In *Proc. of ACM/SIGMOD*, 1987.

[27] H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, and U. Desh. Architecture and Implementation of a Darmstadt Database Kernel System. In *Proc. of ACM/SIGMOD*, 196–207, 1987.

[28] Dimitris Plexousakis. Integrity Constraint and Rule Management in Temporal Deductive Knowledge Bases . In *Proc. of 19th VLDB*, 1993.

[29] G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. of ACM/SIGMOD* , 23–34, 1979.

[30] R. Studer and S. Borner. An Approach to Manage Large Inheritance Networks within a DBMS Surting Nested Relations. In *Nested Relation and Complex Obejects in Databases*, 229–239, 1989.

[31] T. Topaloglou, A. Illarramendi, and L. Sbattella. Query Optimization for KBMSs: Temporal, Syntactic and Semantic Transformation. In *Proc. of Int. Conf. on Data Engineering*, 310–319, 1992.

[32] T. Topaloglou, I. Jurisica, and A. Shrufi. Query Processing in Telos. manuscript in preparation .

[33] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proc. of 12th VLDB*, 101–109, 1986.

[34] Patrick Valduriez. Join Indices. *ACM Transaction on Database Systems*, 12(2):218–246, June 1987.

[35] H. Wang, J. Mylopoulos, A. Kushniruk, B. Kramer, and M. Stanley. KNOWBEL: New Tools for Exprest System Development. In *Development of Knowledge-Based Shells*, N. Bourbakis (ed). 1991.

[36] S.B. Yao. Approximating Block Accesses in Database Organizations. *Communications of ACM*, 20(4):260–261, 1977.