Building Knowledge Base Management Systems*

John Mylopoulos, Vinay Chaudhri, Dimitris Plexousakis
Adel Shrufi and Thodoros Topaloglou
University of Toronto †

May 8, 1995

Abstract

Advanced applications in fields such as CAD, Software Engineering, Real-Time Process Control, Corporate Repositories and Digital Libraries require the construction, efficient access and management of large, shared knowledge bases. Such knowledge bases cannot be built using existing tools such as expert system shells, because these do not scale up; nor can they be built in terms of existing database technology because such technology does not support the rich representational structure and inference mechanisms required for knowledge based systems.

This paper proposes a generic architecture for a knowledge base management system intended for such applications. The architecture assumes an object-oriented knowledge representation language with an assertional sublanguage used to express constraints and rules. It also provides for general-purpose deductive inference and special-purpose temporal reasoning.

Results reported in the paper address several knowledge base management issues. For storage management, a new method is proposed for generating a logical schema for a given knowledge base. Query processing algorithms are offered for semantic and physical query optimization, along with an enhanced cost model for query cost estimation. On concurrency control, the paper describes a novel concurrency control policy which takes advantage of knowledge base structure and is shown to outperform two-phase locking for highly structured knowledge bases and update-intensive transactions. Finally, algorithms for compilation and efficient processing of constraints and rules during knowledge base operations are described. The paper describes original results, including novel data structures and algorithms, as well as preliminary performance evaluation data. Based on these results, we conclude that knowledge base management systems which can accommodate large knowledge bases are feasible.

Keywords: Knowledge bases, knowledge base management systems, knowledge representation, storage management, query processing, concurrency control, constraint enforcement, rule management

---

*Results reported in this paper are based on research conducted within the project titled "A Telos Knowledge Base Management System", funded by the Province of Ontario through the Information Technology Research Center; additional funding for the project has been received from the Department of Computer Science of the University of Toronto and the National Science and Engineering Research Council of Canada.

†Authors' full address: Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Canada, M5S 1A4.

1 Introduction

"...Databases will be exploited in many environments in the year 2000 and will offer many new functions and features. The combination of new environments and new application areas will pressure database technology to invent new functionality ... At the same time, the natural evolution of database technology will provide a technology push to make databases usable for a much broader range of applications...." Patricia G. Selinger.[1]

Many advanced applications in diverse areas such as CAD, Software Engineering, Real-Time Process Control, Corporate Repositories and Digital Libraries require the construction, efficient access and management of large, shared knowledge bases. For example,

- a CAD application for aircraft design may involve tens of thousands of generic objects, rules and constraints, and hundreds of thousands of instances of the design schema, where the generic objects describe aircraft parts (wings, engines, fuselage etc.) while constraints and rules specify policies that must be respected because they represent physical laws, government standards or company regulations;
- real-time knowledge based systems, which need to monitor incoming data for an industrial process or for traffic control and offer diagnostic assistance to human operators in case of an emergency; knowledge bases for such applications need to store information about the process being monitored, the problems that can arise and how to diagnose them; for an industrial process, such knowledge includes a plant schematic, knowledge about plant components (pipes, valves, boilers, etc.) and their operational characteristics, knowledge about hardwired functions (what does it mean when alarm 692 goes off) and diagnostic knowledge used by plant operators to determine the nature of an emergency (Mylopoulos et al., 1992);
- "grand challenges", such as information system support for environmental global change research (Stonebraker and Dozier, 1991) and the human GENOME project (Frenkel, 1991);
- knowledge sharing applications that involve construction of generic knowledge bases that include thousands of concept descriptions and are used as references in the construction of knowledge based systems (Neches et al., 1991).

Such knowledge bases may be built in terms of existing knowledge representation systems (expert system shells, for instance) or AI languages such as Lisp or Prolog. Unfortunately, such implementations do not scale up for several reasons, including inefficient memory management, lack of provisions for sharing, expensive (and sometimes ill-defined) knowledge base operations (Lockemann, Nagel and Walter, 1991; Ishikawa et al., 1993).

[1] VLDB-93 invited lecture, Dublin, Ireland.

Alternatively, such knowledge bases may be built on top of one or more existing database management tools. Unfortunately, this is not a satisfactory solution either. First, the modeling facilities provided by existing database management tools only support a subset of the rich representational structures and inference mechanisms of knowledge representation schemes. Second, available optimization mechanisms do not exploit the rich structure and semantic properties of knowledge bases. Finally, this approach delegates important managerial aids such as semantic integrity enforcement to the end-users of the knowledge base, rather than to the system that manages the knowledge base.

This paper proposes a generic architecture for a knowledge base management system (KBMS) and describes a body of results addressing issues that range from storage management, query processing and concurrency control to rule processing. The proposed system offers a rich representational framework including structuring mechanisms (generalization, aggregation, classification and others) as well as an assertion language for expressing deductive rules and integrity constraints. Moreover, the system supports reasoning mechanisms including deductive inference, constraint enforcement and temporal reasoning. The representation language adopted for the KBMS design is Telos (Mylopoulos et al., 1990).

The term "knowledge base" is used throughout the paper, instead of "database", mostly for historical reasons. There are no technical grounds for distinguishing between the two terms, in view of the fact that (extended) database systems (such as ones managing object-oriented, active and deductive databases) do support some deductive and non-deductive inference mechanisms and structuring facilities analogous to those found in knowledge bases. The difference in meaning, if any, between the two terms is mostly in the degree to which they support representational, structuring and inference capabilities.

The rest of the paper is organized as follows. Section 2 presents a brief overview of the knowledge representation framework of Telos. Section 3 proposes a generic architecture for a KBMS and its components. Sections 4-7 describe respectively research results on storage management, query processing, concurrency control and constraint and rule management. Each one of these sections defines the problem in the context of knowledge bases, identifies limitations of existing approaches (if such exist) and proposes solutions, along with an evaluation. The methodology used to evaluate the proposed research results varies with the results being evaluated. Section 8 summarizes the results of this work and outlines open problems for further research.

2 Overview of Telos

The representational framework of Telos (Mylopoulos et al., 1990) constitutes a generalization of graph-theoretic data structures used in semantic networks (Findler, 1979), semantic data models (Hull and King, 1987) and object-oriented representations (Zdonik and Maier, 1989). Telos treats attributes as first-class citizens, supports a powerful classification (or instantiation) mechanism which enhances extensibility and offers special representational and inferential mechanisms for temporal knowledge. In addition, there have been formal accounts of the semantics of the language based on an axiomatic approach (Stanley, 1986) or a possible-worlds model (Plexousakis, 1993b). This section introduces the core features of Telos which are divided into *structural, temporal* and *assertional* features. A more comprehensive description of the language can be found elsewhere (Mylopoulos et al., 1990).

2.1 Structural Component

A Telos knowledge base consists of structured objects built out of two kinds of primitive units, *individuals* and *attributes*. Individuals are intended to represent entities (concrete ones such as `John`, or abstract ones such as `Person`), whereas, attributes represent binary relationships between entities or other relationships. Individuals and attributes are referred to by a common term – *proposition*. As in object models, Telos propositions have their own internal identifiers.

Every proposition `p` consists of a *source*, a *label* and a *destination* which can be retrieved through the functions `from(p)`, `label(p)` and `to(p)`. A proposition can be represented by a 3-tuple [2] (e.g. `[Martin,age,35]`).

Propositions (individuals and attributes) are organized along three dimensions, referred to in the literature as *attribution* (Attardi and Simi, 1981), *classification* and *generalization* (Brodie, Mylopoulos and Schmidt, 1984).

Structured objects consist of collections of (possibly multi-valued) attributes that have a common proposition as a source, thus adding a simple form of *aggregation*. The structured object corresponding to an individual, for example, may consist of the following set of propositions:

```
{MTS, [MTS, InstanceOf, Employee], [MTS, name, Martin], [MTS, sal, 30000],
     [MTS, addr, '10 King's College Road'], [MTS, dept, 'Computer Science']}
```

In this case, `MTS`, an employee named `Martin`, has a `sal` attribute with value `30000`, an `addr` attribute with value `10 King's College Road` and an attribute `dept` with value `Computer Science`. Note that an attribute may also represent abstract relationships such as `[Person,addr,GeographicLocation]`, intended to represent the concept of the address relationship between persons and geographic locations.

Each proposition is an instance of one or more generic propositions called *classes* – thus giving rise to a classification hierarchy. Propositions are classified into *tokens* – propositions having no instances and intended to represent concrete entities in the domain of discourse, *simple classes* – propositions having only tokens as instances, *meta-classes* – having only simple classes as instances, *meta-meta-classes*, and so on.

Orthogonal to the classification dimension, classes can be organized in terms of *generalization* or `isA` hierarchies. A class may have incomparable generalizations leading to hierarchies that are directed acyclic graphs rather than trees. The attribute mechanism is also used for attaching assertions (deductive rules and integrity constraints) to Telos objects. Inheritance of attributes with respect to generalization is assumed to be strict in the sense that a class definition cannot override inherited attributes.

[2] Later, when the temporal dimension is introduced, propositions will be shown as 4-tuples.

Figure 1: AN EXAMPLE TELOS KNOWLEDGE BASE

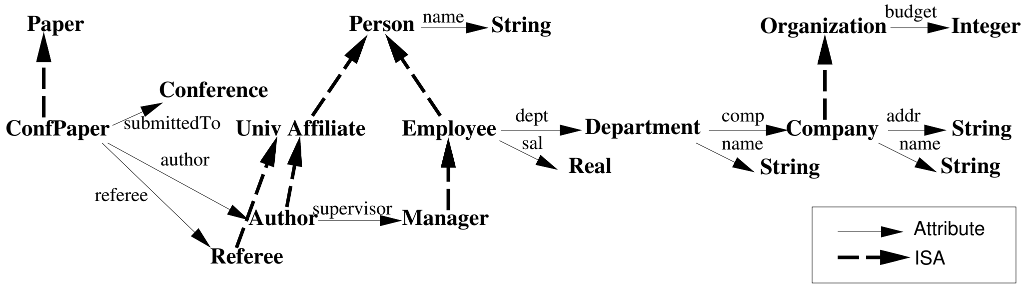


Figure 1 shows an example Telos knowledge base in the form of a labeled directed graph. Instantiation, generalization and attribution relationships are represented as edges in the graph. For example, in Figure 1 the attribute `[Employee, dept, Department]` is represented in this graph as an edge between `Employee` and `Department` which is labeled by `dept`.

2.2 The Temporal Component

Every Telos proposition has an associated *history* time and a *belief* time. The history time of a proposition represents the lifetime of a proposition in the application domain (i.e., the lifetime of an entity or a relationship). A proposition's belief time, on the other hand, refers to the time when the proposition is believed by the knowledge base, i.e., the interval between the moment the proposition is added to the knowledge base and the time when its belief is terminated. Both history and belief time are represented by means of time *intervals*. The model of time adopted is a modification of Allen's framework (Allen, 1983). Seven exclusive temporal relations (e.g, `equal, meet, before, after, during, start, end`) together with their inverses are used to characterize the possible positions of two intervals on a linear time line. Temporal relationships participate in the expression of deductive rules and integrity constraints in the assertion language. Disjunction of temporal relationships is disallowed in order to facilitate efficient temporal reasoning.

A proposition with history time will now be a 4-tuple. For example, the proposition, `[Martian, addr, '10 King's College Road', 1/1/89..3/10/89]` means that `Martian` had an `addr` of `10 King's College Road` during the period `1/1/89` to `3/10/89`. The time component of a proposition `p` may be retrieved using the function `when(p)`.

Telos has several built-in temporal constants, such as dates and times (e.g., `1988/12/07` denoting December 7, 1988, `1988` denoting the interval starting at the beginning of 1988 and ending at the end of 1988), semi-infinite intervals having conventional dates or times as one endpoint (e.g., `1986/10/25` .. *) and the infinite interval `Alltime`.

2.3 The Assertional Component

Telos provides an assertion language for the expression of deductive rules and integrity constraints. The assertion language is a first-order language with equality.

Function symbols of the language that have already been mentioned in the previous sections are `from(p)`, `label(p)`, `to(p)` and `when(p)`, returning the source, label, destination and duration of `p` respectively. In addition, the assertion language supports selection operations which make it possible to "navigate" through a Telos knowledge base. The selection operations include a dot function operation `x.l [r1 t1]` which evaluates to the set of `to`-values of the attributes of proposition with source `x` which belong to the attribute class labeled by `l` during intervals which are in relation `r1` with time interval `t1`. The definitions of other functions can be found elsewhere (Mylopoulos et al., 1990). The terms of the language include variables, constants (including conventional dates) and the result of applying functions to terms.

The atomic formulae of the assertion language include the predicates `prop()`, `instanceOf()`, `isA()`, $att()$ for an attribute att, the temporal predicates `before`, `during`, `overlaps`, `meets`, `starts`, `finishes`, `equal` and their inverses `after`, `contains`, `overlapped by`, `met by`, `started by` and `finished by`. The predicate `prop(p,x,y,z,t)`, means that `p` is a proposition with components `x,y,z` and `t`. The predicate `instanceOf(x,y,t1,t2)` means that `x` is an instance of `y` for the time period `t1` and is believed by the system for the time period `t2`. Similarly, the predicate `isA(x y,t1,t2)` means that `x` is a specialization of `y` for the time `t1` and is believed by the system for time `t2`. Finally, att`(x y,t1,t2)` denotes that `y` is a value of the attribute att of `x` for `t1` and is believed for `t2`. Also, for any terms x and y and any evaluable predicate θ, $x\ \theta\ y$ is an atomic formula with the obvious meaning.

A meta-predicate `Holds` denotes the truth of an atomic formula during a time interval. For example, the history-time assertion $\texttt{Holds}(p(x,y),t)$ means that there exists a time interval t_0 such that $\texttt{prop}(pid,x,p,y,t_0) \wedge (t_0\ \texttt{contains}\ t)$ is true, if p is a basic predicate, or that the truth of $p(x,y)$ at time t is derivable from the knowledge base via the deductive rules, if p is a derived predicate. The predicate p is restricted to be a non-evaluable and non-temporal predicate[3] since the truth of evaluable or temporal predicates is not dependent on their evaluation over a particular time interval. A belief-time assertion has the form $\texttt{Believed}(HT,t')$, where `Believed` is a meta-predicate denoting that a history-time assertion HT is believed by the system throughout a belief time interval t'. Using these predicates leads to more succinct temporal expressions.

Well-formed formulae of the assertion language are formed by using the meta-predicates `Holds` and `Believed`, logical connectives and restricted quantification. Restricted quantification is of the form `Forall` x/C and `Exists` x/C for a Telos class C. The assertion language is used to pose queries or to specify the integrity constraints and deductive rules in the knowledge base.

With respect to the knowledge base of Figure 1, the following query retrieves all the employees who have had an increase of more than $5000 in their salary in 1988 and according to the beliefs of the system from

[3] Evaluable predicates are those consisting of a comparison or set membership operator. Temporal predicates correspond to the 13 possible interval relationship. The remaining predicates are called non-evaluable.

the beginning of 1988 and on.

$$\begin{array}{l} \texttt{ASK e/Employee : Exist } \texttt{t}_1, \texttt{t}_2\texttt{/TimeInterval } (\texttt{e[t}_1\texttt{].sal} \leq \texttt{e[t}_2\texttt{].sal} - 5000) \\ \qquad\quad \texttt{and } (\texttt{t}_1 \texttt{ before } \texttt{t}_2) \\ \qquad \texttt{ON 1988} \\ \qquad \texttt{AS OF } (1988..*) \end{array}$$

Similarly, referring to the example knowledge base of Figure 1, the following formula expresses the constraint that "*no author of a conference paper can be a referee for it*". Moreover, the constraint is assumed to be valid from (the beginning of) 1988 and on.

$$\begin{array}{l} \texttt{Forall } p/\texttt{ConfPaper Forall } x/\texttt{Author Forall } r/\texttt{Referee Forall } t/\texttt{TimeInterval} \\ \quad (\texttt{Holds}(author(p,x),t) \wedge \texttt{Holds}(referee(p,r),t) \Rightarrow (r \neq x)) \ (\texttt{at } 1988..*) \end{array}$$

The same constraint could also be expressed without the meta predicate `Holds` as:

$$\begin{array}{l} \texttt{Forall}p/\texttt{ConfPaper } \forall x/\texttt{Author } \forall r/\texttt{Referee}/\forall t_1,t_2,t_3,t_4/\texttt{TimeInterval} \\ \quad [author(p,x,t_1,t_2) \wedge referee(p,r,t_3,t_4) \wedge \texttt{during}(t_3,t_1) \wedge \texttt{during}(t_4,t_2)] \Rightarrow [r \neq x](\texttt{at } 1988..*) \end{array}$$

The two forms of this constraint are equivalent, but as we can see, the former is more succinct than the later. Also, the latter form includes explicit quantification over belief time intervals. For the purpose of this section, we will only use the succinct form.

The above constraint is an example of a *static* constraint, i.e., a property applicable to all states of the domain of discourse. The canonical example of a *dynamic* integrity constraint, expressing the property that "*an employee's salary should never decrease*" can be expressed by the following assertion language formula:

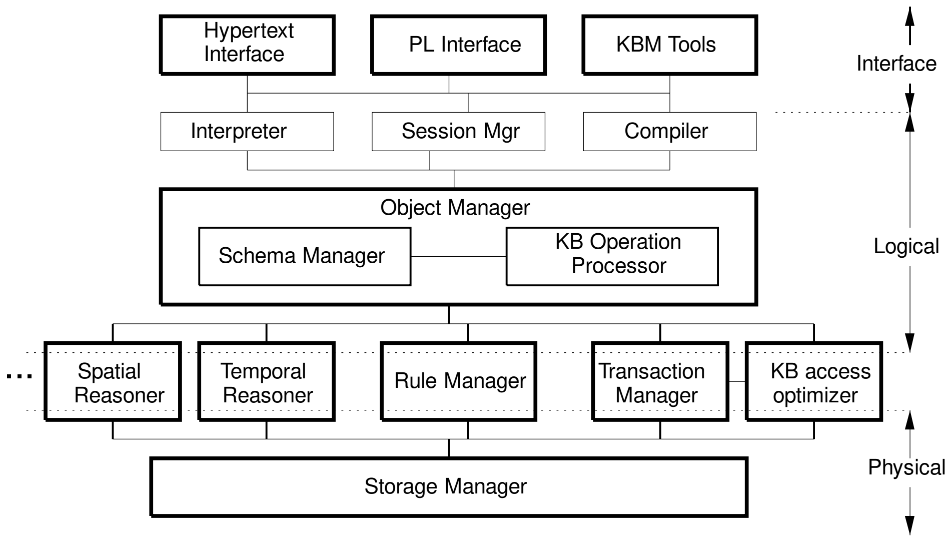
$$\begin{array}{l} \texttt{Forall } p/\texttt{Employee Forall } s,s'/\texttt{Integer Forall } t_1,t_2/\texttt{TimeInterval} \\ \quad (\texttt{Holds}(salary(p,s),t_1) \wedge \texttt{Holds}(salary(p,s'),t_2) \wedge \texttt{before}(t_1,t_2) \Rightarrow (s \leq s')) \ (\texttt{at } 02/01/1988..*) \end{array}$$

Notice the absence of belief-time assertions from the expressions of the above constraints. Both formulae are interpreted over the same belief time interval.

Constraints, as well as deductive rules, are associated with history and belief time intervals. In the previous examples, the history time intervals associated with the constraints are the semi-infinite intervals `(1988 .. *)` and `(02/01/1988 .. *)` respectively. If no such association appears explicitly with their definition, both intervals are assumed to be equal to `(systime..*)`, where `systime` denotes the current system time. Examples of different kinds of constraints expressible in the assertion language are given in Section 7.

Syntactically, deductive rules are considered to be special cases of integrity constraints. The general form of a deductive rule is $\texttt{Forall } x_1/\texttt{C}_1 \ldots \texttt{Forall } x_n/\texttt{C}_n \ (F \Rightarrow A)$, where F is a well-formed formula and A is a history-time assertion or an evaluable predicate. As an example, consider the rule "*A university affiliate works in the department that has the same address as she does*", expressed by the formula:

Figure 2: GLOBAL KBMS ARCHITECTURE



$$\texttt{Forall } u/\texttt{UnivAffiliate Forall } d/\texttt{Department Forall } s,s'/\texttt{Address Forall } t/\texttt{TimeInterval}$$
$$(\texttt{Holds}(address(u,s),t) \wedge \texttt{Holds}(D\_addr(d,s'),t) \wedge (s=s')] \Rightarrow \texttt{Holds}(works\_in(u,d),t))\ (\texttt{at 1988..}*)$$

3 System Architecture

The KBMS design is based on an extensible and layered architecture. An extensible architecture is necessary because the KBMS is intended to support both general-purpose and special-purpose inference mechanisms. Special-purpose inference mechanisms, for example, spatial reasoning, case-based reasoning, need to be incorporated depending on specific application needs, whereas, general-purpose mechanisms will be common across all applications. A layered architecture supports design based on increasing levels of abstraction, thereby partitioning the overall design problem of KBMSs into several sub-problems. In the long term, such an architecture can lead to standard interfaces for each layer and its components, so that layer implementations can be re-used across different KBMSs.

The system architecture adopted is shown in Figure 2.[4] The architecture provides for three different layers: an interface layer, which offers different types of user interfaces, a logical layer which handles primitive knowledge base operations for retrieval and update, a physical layer which manages the data structures used to store the knowledge base, and a variety of indices and other auxiliary information.

The interface layer offers knowledge base users a variety of services, including a hypertext interface for ad hoc user interactions and a programming language (PL) interface which supports the execution of application programs that include knowledge base operations. In addition, the interface layer may include knowledge base management (KBM) tools for knowledge acquisition, knowledge base verification, validation, evolution

[4]It has to be noted that this is a fairly abstract and high level design. Implementation is in a very primitive stage. The performance results reported were obtained by simulation.

and knowledge sharing (Neches et al., 1991; Buchanan and Wilkins, 1993). These services are interfaced with the logical layer through the knowledge representation language interpreter or compiler and a session manager.

The logical layer maintains information on class definitions, including rules and constraints, and supports primitive knowledge base operations such as `TELL` and `ASK` (Mylopoulos et al., 1990). Its services are implemented on top of the physical layer in terms of a collection of modules which provide for basic data management functions, such as access path planning for efficient query processing and concurrent execution of transactions, special-purpose reasoners for temporal, spatial or other types of reasoning, as well as a rule management component which supports deductive inference and constraint checking.

Finally, the physical layer is responsible for the management of the disk-based data structures on which the knowledge base is stored, indices supported by the architecture, caching policies, etc. The functionality of the bottom part of this layer is assumed to be provided by a storage kernel such as the ones designed for object-oriented and nested relational databases (Carey et al., 1986; Paul et al., 1987; Biliris, 1992).

It has to be noted that this is a fairly abstract and high level design. Implementation is in a very primitive stage. The performance results reported were obtained by simulation.

The remainder of the paper focuses on the logical layer and the modules that implement it on top of the physical layer.

4 Storage Management

Despite the pace of hardware advances, the knowledge bases envisioned for the proposed KBMS will be too large to fit in main memory. Existing knowledge base building tools, such as expert system shells, do not provide for efficient secondary memory storage. When the size of a knowledge base grows beyond system limits, these tools rely on the underlying operating system which manages memory through paging for disk I/O. This means that disk I/O, generally considered the most serious bottleneck to a system's performance, is delegated to a system component that knows nothing about the structure and access patterns of the knowledge base. Not surprisingly, such implementations do not scale up to knowledge bases of size beyond $O(10^4)$ tokens. On the other hand, the coupling of an expert system shell to an existing (say, relational) DBMS for storing the knowledge base, is also a non-solution because conventional data models are not sufficiently rich to support the representational and inferential features of a KBMS, thus, rendering DBMS-supported optimization techniques ineffective.

Our objective in this section is to devise a suitable scheme for storing the knowledge base on a disk, taking into account the semantic features of a Telos knowledge base. Within our framework, the generation of a storage management scheme is viewed as a three step process. In the first step, called logical design, the knowledge base structure is mapped into a set of logical storage structures (such as a relational schema). The second step, called physical design, determines disk layout for the relations defined during logical design. The third step makes provisions for indices to be used for accessing stored information. In this paper, we limit the discussion to logical design and indexing.

4.1 Related Work

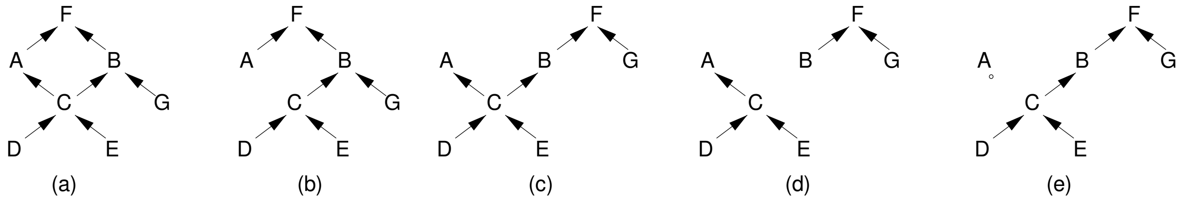
From the logical storage design approaches that have been proposed for object-oriented databases, two prominent approaches are the n-ary (direct) storage model (NSM) and the decomposition storage model (DSM) (Valduriez, Khoshafian and Copeland, 1986; Copeland and Khoshafian, 1985). The DSM has been proposed as an implementation technique for relational and complex object databases. According to the DSM, a separate relation is defined for each attribute of a class. As a result, class instances have values of their attributes distributed over many relations. Object identifiers (OIDs) serve as the connectives of different attribute values of the same object. A major advantage of the DSM is that it can be easily implemented and does not require storage reorganization because of schema evolution.[5] Disadvantages of the DSM include relatively high storage cost (usually double the cost of non-decomposed storage schemes) and inefficiency in executing operations which access many attributes of a single object. Because of these reasons, the DSM is definitely not appropriate for applications that tend to process objects in their entirety (e.g., CAD/CAM or software engineering). In the (n-ary) storage model (NSM), a separate relation is defined for each class in the knowledge base. If the total number of attributes for a class, including the attributes inherited from parent classes is n, then the relation corresponding to this class will have $n+1$ attributes. The extra attribute in the relation is needed for the object identifier. If the value of an attribute of a class 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. The advantage of this method is that it efficiently supports the access of a large number of attributes or subclasses of a given class. Its greatest disadvantage is that it is inflexible with respect to schema updates and has poor performance on sub-class updates.

In summary, both methods can deal with complex objects, while, in addition, the DSM offers support for generalization (or `isA`) hierarchies. However, neither method addresses issues of storage management for temporal knowledge, deductive rules and integrity constraints. In this section we propose an algorithm called, Controlled Decomposition Method (CDM), that combines the advantages of both the NSM and DSM and also takes into account temporal knowledge.

As far as access methods are concerned, the most promising solution for knowledge bases is the join index (Valduriez, 1987). The join index is used, as part of the data storage, by the DSM. Once again, however, this index cannot be adopted as it is because it offers no provisions for dealing with temporal attributes. The CDM extends this index to the Temporal Join Index (TJI) for this purpose.

[5] Schema evolution means adding to or deleting from the knowledge base classes or attributes.

Figure 3: ISA HIERARCHIES



4.2 The Controlled Decomposition Model

Generation of a logical design using the Controlled Decomposition Model (CDM) involves three steps. The first step transforms a given generalization hierarchy from a general graph to a forest (a set of trees). The second step generates relations corresponding to the forest. The final step creates hierarchical object identifiers for the tuples of the relations created in step 2.

Assume that the **isA** hierarchy associated with a knowledge base is represented as a directed acyclic graph $G = (V, E)$, where V is the set of classes and E is the set of **isA** relationships, declared or derived, between these classes. Furthermore, it is assumed that statistics for frequency of access (# of accesses over time) for every class $v \in V$, denoted $f(v)$, and frequency of access for every attribute a of a class v, denoted $g(v, a)$, are available. Such information can be obtained by, for instance, looking at the trace of transactions operating on a knowledge base over a period of time. We define the *most preferred parent* of a node v, with parents $v_1, v_2, \ldots, v_k$, as node v_i if $f(v_i) \geq f(v_j), 1 \leq j \leq k$. If more than one node satisfies this criterion, we arbitrarily pick one of them as the most preferred parent.

The CDM rule for transforming an **isA** hierarchy into a forest is then defined as follows:

C1. For every class $v \in V$ choose only one outgoing edge, the one leading to the most preferred parent. The attribute list of the class v is extended with the attributes inherited form the disconnected parents.

For example, decomposing the hierarchy shown in Figure 3 (a) involves comparing the values of $f(A)$ and $f(B)$. If $f(A) > f(B)$, the resulting hierarchy will be as shown in Figure 3 (b). It can also be the case that a class does not have a unique common ancestor (for example see Figure 3 (c)). In such a case, rule C1 is used to break a hierarchy into two. Figures 3 (d) and (e) show the possible outcome, depending on which of $f(B)$ and $f(A)$ is greater.

C2. For each class in this forest, a relation is created with attributes as determined by rule C4 below.

C3. A token is stored as a tuple in the relation corresponding to the most general class of which the token is an instance. Object identifiers are assigned to each tuple using a hierarchical scheme (see (Topaloglou, 1993)).

C4. A relation corresponding to a class will have all the local attributes of that class except in the following cases:

Figure 4: RELATIONS CREATED BY THE CDM

PERSON

P_ID		ENAM
P31	1..6	Alex
P03	4..8	Mary
P24	8..10	Mike
P25	6..9	Nick
P37	6..10	Karen
P28	5..10	John

EMPLOYEE (IsA PERSON)

E_ID		SAL
P31	1..6	{45 1..4, 55 5..6}
P24	8..10	{60 8..10}
P25	6..9	{55 6..9}
P28	5..10	{45 5..8, 60 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}
D6	9..10	{Res 9..10}
D7	9..10	{Dev 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
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
D6	O13	9..10
D7	O14	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, MA 9..10}	{TOR 1..10}
O12	4..10	{KC 6..10}	{Toronto 4..8 Paris 9..10}

C4a. If an attribute has a complex domain (i.e., its values are instances of another class), the relation corresponding to this class is decomposed by creating a separate relation for that attribute. Consider the class R and its attribute A that has class S as its domain. If r_i denotes the object identifier corresponding to a tuple in R and s_i denotes the object identifier corresponding to a tuple in S, then the decomposed relation RS is specified as:

$$RS = \{(r_i, s_i, t) | r_i.A \ [\texttt{at} \ \ t] = s_i\}$$

C4b. If an attribute A of a class R has a simple domain (e.g., `Integer, String, etc.`), and its access frequency $g(A, R)$ is greater than a threshold value g_T, then the relation corresponding to class R is decomposed such that the attribute A is stored in a separate relation RA. If r_i is an object identifier for a tuple in R, then the relation RA is specified as follows:

$$RA = \{(r_i, a_i, t) | r_i.A \ [\texttt{at} \ \ t] = a_i\}$$

C5. Every attribute A is stored as a 3-tuple $\langle a_i, h_i, b_i \rangle$, where a_i is the value of the attribute, h_i is the history time and b_i is the belief time.[6]

According to this logical schema representation of the knowledge base, the attributes of a class that are spread across several relations will have the same object identifier. The reader may have noticed that rules C2, C3 and C4a are adaptations from the DSM, whereas rule C4b is adopted from the NSM. Rule C5 accounts for temporal knowledge.

The relations which are generated by the CDM algorithm for a portion of the knowledge base of Figure 2

[6] In the discussion that follows we restrict ourselves to history time only.

are shown in Figure 4. The `EMPLOYEE` relation stores only the local attribute of the `Employee` class (e.g., `sal`) whereas the employee name is stored in the `PERSON` relation (rule C4). Since the attribute `dept` of the class `Employee` has a complex domain `Department`, a separate `EMP_DEPT` relation is created for it. The example only shows the history times associated with each attribute and instance-of relationship. In particular, each attribute value is stored with its corresponding history time interval. Values of the attribute at different history times are indicated by braces.

4.3 Performance Evaluation of the CDM

This section compares the CDM with the DSM and the NSM with respect to storage cost and reorganization cost incurred by changes in the schema.

4.3.1 Storage Cost

The following table summarizes the parameters which characterize a tree structured isA hierarchy $\mathcal{H}$. (We consider a tree structured hierarchy here, because, the rule C1 of the CDM transforms an arbitrary ISA hierarchy into a set of trees.)

Knowledge Base Parameters

C	number of classes in $\mathcal{H}$
d	average degree of decomposition in $\mathcal{H}$
l	average number of locally defined attributes per class
N	average number of instances per class
b	branching factor of the isA tree
h	height of the isA tree
i	size of an object identifier, attribute value and temporal identifier[7]

Let us now compute the required disk storage (rds) for each of the three models.

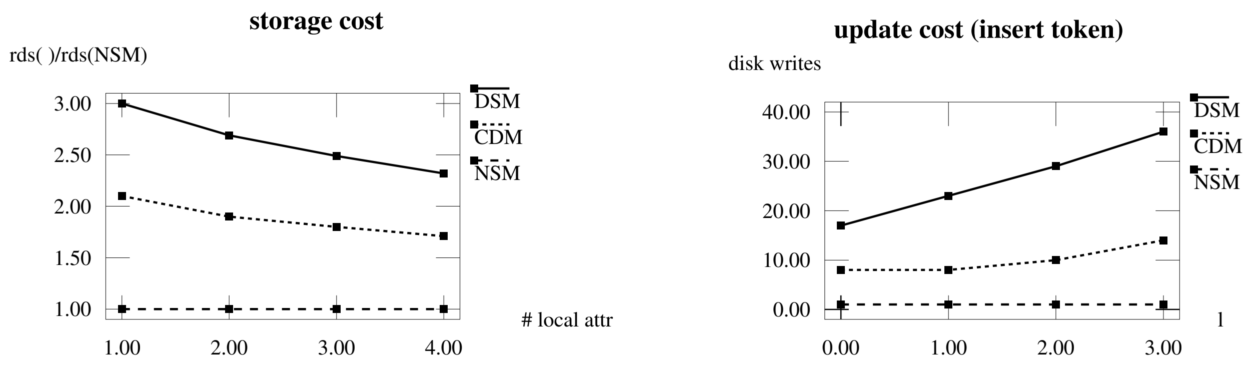
As mentioned earlier, NSM stores one relation with $n+1$ fields per class, where n is the sum of the number of locally defined attributes and the number of inherited attributes. The total number of relations is $\sum_{k=0}^{h} b^k$. A class at depth k has $k * l$ inherited attributes, giving $n = l + kl$. Therefore, the required disk storage is estimated by the equation:

$$\text{rds(NSM)} = \sum_{k=0}^{h} [(2*(l+kl)+1)i] b^k N \tag{1}$$

DSM stores one relation per attribute which amounts to $\sum_{k=0}^{h} b^k l$ relations. In each relation there is a tuple for each of the instances of that class. Therefore, the total number of DSM tuples is determined by the formula $N = \sum_{k=0}^{h} (b^k l)(1 + \sum_{j=k+1}^{h} b^j)N$. Each tuple has three fields. Therefore, the required disk storage can be estimated as follows:

[7] For simplicity, we assume that these sizes are equal and thus in Equation 1, for instance, we use the multiplier 2 to denote a value and a temporal identifier. A more general model that does not make this assumption is available elsewhere(Topaloglou, 1993).

Figure 5: (a) STORAGE AND (b) UPDATE COST



$$\text{rds(DSM)} = \sum_{k=0}^{h} 3 * (b^k l)(1 + \sum_{j=k+1}^{h} b^j)N \tag{2}$$

Note that equations 1 and 2 modify (Copeland and Khoshafian, 1985) formulas for NSM and DSM storage, taking into account attributes defined over `isA` hierarchies.

In the CDM, the degree of decomposition depends on the average number of attributes with complex domains and the access frequency of the attributes. The storage cost for CDM is the weighted sum of the costs for the NSM and DSM except for one difference. Unlike NSM, the attributes are not inherited, and in each relation corresponding to each class, there is a tuple corresponding to all its instances. Therefore, in Equation 1, the term kl is dropped when computing the storage cost, and a correction $cor(NSM)$ is applied to take into account the cost of storage of the instances (A more detailed analysis appears elsewhere (Topaloglou, 1993).) With this change, the expression for the storage cost of CDM is as follows:

$$\text{rds(CDM)} = (1-d) * (rds(NSM) + cor(NSM)) + d * rds(DSM) \tag{3}$$

The estimates of space costs suggested by the above formulae confirm the claim of (Copeland and Khoshafian, 1985) that DSM requires 2 to 4 times more data storage than NSM. As expected, CDM's storage costs fall between those of the other two schemes. The first graph in Figure 5 shows relative storage costs for a balanced-tree `isA` hierarchy with height 4 and branching factor 2 as the number of local attributes for each class is varied from 1 to 4. These results suggest that CDM requires on average 65% of the required space of DSM.

4.3.2 Update Costs

The most frequent types of updates in a knowledge base and their estimated cost functions are given below. The comparisons are also shown in Table 1.[8]

[8] Deletions in Telos are analyzed as transactions consisting of an update-value and an insertion part.

Table 1: SUMMARY OF UPDATE COSTS

update operation	NSM	DSM	CDM
insert attribute	not supported	new binary relation	rule C4
insert sub(super)class	not supported	$l+1$ new relations	rules C2, C4
insert token (in disk writes)	1	$1+2n$	$(1-d)+d(1+2n)$

1. Adding a new attribute to a class: This type of update can be realized incrementally by creating a new relation for this attribute.

2. Adding a subclass/superclass: First, the new class receives an identifier denoting its position in the `isA` hierarchy. Then, a new relation with $l+1$ fields is created; l is the number of the specialized attributes associated with the new class that have primitive attribute domains. Those with complex domains form additional relations as required by rule C4.

3. Inserting a new token: For the NSM, this operation requires a single tuple insertion which corresponds to one disk access. In the DSM, this operation requires $2n+1$ writes, where n is the number of attributes of the token. The cost in CDM is the weighted sum of the costs of the NSM and DSM models. Experimental results have shown that CDM's cost can be as low as 1/4 of DSM costs (Topaloglou, 1993).

4. Updating 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, as with the CDM, this requires on average three writes (Khoshafian and Copeland, 1986). Appropriate buffering policy may further minimize that cost. The graph of Figure 5(b) shows the cost for the token insertion case.

In summary, the CDM performs as well as the DSM on attribute value and schema updates, and performs better than the DSM on token updates. CDM's performance is worse than that of the NSM's for token updates. However, the NSM cannot handle schema updates at all. A summary of the comparison is presented in Table 1.

4.4 Access Methods

The access methods supported in the proposed KBMS architecture are based on three types of indices: the simple temporal index (STI), the temporal join index (TJI), and the record addressing index (RAI). STI and TJI are used to select object identifiers satisfying a certain condition. Given an object identifier, RAI determines the location of its components on disk. Due to space limitations, only a brief discussion of STI and TJI is presented here. The interested reader can find more details about the functionality and the performance of the above indices elsewhere (Topaloglou, 1993).

The STI and TJI are generalizations of non-temporal indices where, in contrast to the traditional $\langle key, value \rangle$ pair, each entry is a triplet $\langle key, value, time \rangle$, where $time$ is the history time during which the entry is true. With these indices it is possible to perform selection using keys and time simultaneously. Since the index entries are no longer points, it is assumed that they are implemented as spatial access methods (Guttman, 1984). Moreover, the indices are hierarchical in the sense that one index stores information for all the instances of a class (including inherited ones). This choice is consistent with a previous study (Kim, Kim and Dale, 1989) which shows that a hierarchical index always outperforms the simple class index.

The simple temporal index (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. An STI is defined for most frequently accessed attributes. The design of a more comprehensive set of criteria for index selection, along the lines studied in (Finkelstein, Schkolnick and Tiberio, 1988; Frank, Omiecinski and Navathe, 1992) remains an open problem.

The temporal join index (TJI) is used to describe a time-dependent relationship between the instances of two classes. In a TJI corresponding to the classes R and S each entry is a 3-tuple $\langle r, s, t \rangle$, where r and s are object identifiers for some instances of classes R and S, and t is the history time for which this relationship is true. For example, the relations `DEPT_COMP` and `EMP_DEPT` in Figure 4 represent such relationships. The searching key in a TJI is a pair of an object identifier and a temporal interval. For example, for the key $\langle$*OID, time*$\rangle$, a TJI will return the object identifiers of instances of the second class which are in a TJI tuple containing *OID* for a time interval overlapping with *time*. Moreover, TJI is a bi-directional index so that it can be accessed by the object identifier of the either of the classes. To make the two-way access efficient we assume that that two copies of a TJI are stored. Each copy is clustered on one of the object identifiers of the two classes. A series of TJIs can be used to form the temporal equivalent of the multi-join index (Bertino and Kim, 1989).

5 Query Processing

For the Telos KBMS, queries are specified through an `ASK` operation (Mylopoulos et al., 1990), that has the following structure:

```
ASK x1/S1, ..., xn/Sn : W
   ON t1
   AS OF t2
```

$x_1, \ldots, x_n$ are assumed to be target variables for which the query processor needs to determine values; $S_1, \ldots, S_n$ are either set expressions or classes which denote the range of the variables; t_1, t_2 are time interval constants such as dates, and `W` is a formula expressed in the Telos assertion language. The temporal subexpression in the query formula is interpreted as follows: *answer the query W with respect to all propositions whose history time is covered by or overlaps with t_1 according to what was believed by the knowledge base over the interval t_2*.

For example, the first query shown below retrieves all employees who had a salary increase of more than 5K since the beginning of 1988, according to the system's beliefs for the same time period. The second query retrieves all employees who worked for the database group of IBM in 1990 according to what the system currently believes. The first query is a temporal query and the second is a historical query (Snodgrass, 1987).

Q1. ASK e/Employee : Exist t_1, t_2/TimeInterval ($e[t_1]$.salary $\leq$ $e[t_2]$.salary $-$ 5000)
and (t_1 before t_2)
ON 1988
AS OF (1988..*)

Q2. ASK e/Employee : e.dept.comp.name = "IBM" and e.dept.name = "dbgroup"
ON 1990

As with database systems, such queries are expressed in a declarative fashion, in the sense that a query does not indicate how the data should be accessed from the knowledge base physical store. In general, a query is subjected to four phases of processing: parsing, optimization, code generation and execution (Selinger et al., 1979). The focus of this section is on the optimization phase.

Query optimization for knowledge bases is hard for several reasons. First, the representation formalism adopted for the knowledge bases is more expressive given a query language with temporal, spatial, class- and meta-class-related expressions. This requires us to develop new methods for the problem of syntactic and semantic simplification. Similarly, novel indexing techniques for knowledge bases, for instance ones dealing with temporal knowledge or deductive rules, render existing optimization techniques ineffective and require new ones in their place.

Our proposed query optimization method consists of two phases: *semantic query optimization* and *physical query optimization*. During semantic query optimization, a query is simplified according to the semantic features of the knowledge base, resulting in a query that is equivalent to the original but less expensive to evaluate. During physical query optimization, estimates are obtained for the cost of different access operations required to evaluate the query, resulting in a sequence of operations (the *access plan*) that will lead to minimum execution cost. This section addresses both semantic and physical query optimization.

5.1 Semantic Query Optimization

Semantic query optimization in knowledge bases exploits the semantic information due to structural, temporal and assertional properties. That allows for three different steps of semantic optimization: temporal simplification, syntactic simplification and semantic simplification.

The contributions of our method are the following. First, we propose *temporal simplification* in the context of knowledge bases. Second, we have reformulated the known techniques for *syntactic simplification* (Jarke and Koch, 1984; Chakravarthy, Grant and Minker, 1988) to exploit the properties of a knowledge base, such as generalization and aggregation. Finally, we have advanced the *semantic transformation* techniques by using theory resolution (Stickel, 1985) and specialized reasoners. In the following, we summarize the three steps in semantic optimization. More details can be found elsewhere (Topaloglou, Illarramendi and Sbattella, 1992).

5.1.1 Temporal Simplification

Temporal simplification attempts to identify those parts of a knowledge base that are relevant to a query from a temporal viewpoint (Jarke and Koubarakis, 1989). Temporal simplification involves the following three steps:

1. Check for inconsistent or redundant temporal constraints in the query expression;
2. Check whether there are available data for all the target variables of the query, for the *historical* and the *belief* period that the query refers to;
3. Select the part of the knowledge base which is involved in the answering of the query from a temporal viewpoint. At this step the deductive rules and the integrity constraints that are effective for the time periods specified in the query are selected.

The first step is formulated as a constraint satisfaction problem (CSP) on the temporal relations of the query formula. These relations are interval constraints which belong in the *pointsable* class and therefore the CSP is solved in polynomial time (Vilain, Kautz and van Beek, 1989).

The second step requires us to maintain meta-information which is able to answer the following schema query: *Does class C have instances in the KB at the historical (*resp. *belief) time of the query?* On a "no" answer for this, the original query receives "empty answer" and its processing is completed. The testing of the temporal condition in the schema query is formulated as a CSP with *integer order* constraints which is solved in polynomial time (Dechter, Meiri and Pearl, 1989).

For the last step, we need to index the rules and the constraints on their history and belief time. In the two dimensional space that history and belief time define, each rule or constraint is viewed as a rectangular area; We use an *R-tree* based spatial access method (Guttman, 1984) to assist the temporal selection with logarithmic complexity.

5.1.2 Syntactic Simplification

Syntactic simplification exploits the properties of the structural features of Telos (i.e., `isA`, `instanceOf` and `proposition` relationships). Also, subexpressions within the query expression that are always true, always false, or inconsistent are detected.

The syntactic simplification algorithm operates on a query graph representation of the syntactic aspects of the query, where the input query is submitted in prenex disjunctive normal form. *Nodes* in a query graph represent the constant and variable terms and *edges* represent the atomic formulae. The query transformations are carried out using graph rewriting operations. The transformation rules are divided into two groups: *completion rules* and *simplification rules*.

Completion rules add information to the query that may be used either for inconsistency detection or simplification. For instance, the isA transitivity rule adds the predicate `isA(C1,C3,t1*t3,t2*t4)` for any isA(C1,C2,t1,t2) ∧ isA(C2,C3,t3,t4) pattern that is encountered in the query expression.[9]

When the completion phase is executed, the simplification rules are applied to obtain a simplified representation of the query, to eliminate redundant information and to detect inconsistent queries. As an example, the query subexpression isA(C1,C2,t1,t2) ∧ isA(C2,C1,t3,t4) is replaced with False if t1*t3 and t2*t4 are not defined.

5.1.3 Semantic Transformation

The objectives of the semantic transformation step are the same as in the previous step, except that it uses deductive rules and integrity constraints. This step proceeds as follows: It takes as input the set of relevant deductive rules and integrity constraints, called the *rule base*, which is returned from the temporal simplification algorithm, and the query form which is returned from the syntactic simplification algorithm and applies the transformation to the query using theory resolution (Stickel, 1985).

There are some unique features of this algorithm. First, it uses the reduced size rule base that is produced by the temporal simplification and is specific to the query being optimized; consequently, it reduces the search space for the resolution based transformations. Second, it uses theory resolution that can accommodate the use of specialized reasoners for taxonomic, inequality and temporal sub-expressions of Telos queries. Theory resolution is, in general, more efficient than classical resolution because it decreases the length of refutations and the size of the search space. Finally, our semantic transformation algorithm has been shown to be sound (Topaloglou, Illarramendi and Sbattella, 1992).

5.2 Physical Query Optimization

The task of physical query optimizer is to take the simplified query as generated by the semantic optimization phase and generate an optimal execution strategy. The success of a query optimizer in discovering an optimal execution strategy depends critically on how well it can utilize the available physical data structure and the associated access methods. Therefore, statistics about the knowledge base need to be available, as well as a cost model that predicts the cost of using each access operation.

The contributions of this section are as follows. First we develop the necessary cost formulae for the estimate indicators which consist of the disk I/O costs and size estimates of intermediate results. Second, we identify a set of access level operations and associated costs for executing simple queries, and we generalize this result to more complex queries. Third, we tackle the problem of access planning in queries with arbitrary number of path expressions. Finally, we show through an experimental study that in the context of our KBMS, join-index based query processing achieves better performance than the nest-loop and sort-merge methods.

The physical query optimizer performs the following steps. Initially, a simplified query obtained from the

[9] $t_k = t_i * t_j$ denotes the common time interval between two intervals t_i, t_j in case they overlap, and it is undefined if they are disjoint.

semantic query optimizer, is transformed to a query graph which is successively transformed into a set of parameterized operator trees and a set of execution trees. Each execution tree represents a possible access plan. Once the execution trees are available, the one with the least cost is selected using the access optimizer. We will now explain each of these steps in more detail.

5.2.1 Query Graphs

Suppose the formula `W` of the example query (shown at the beginning of this section) has the form $F_1 \wedge F_2 \wedge \ldots \wedge F_n$. Each F_i is a path expression $x_i.A_1.A_2 \ldots A_k.a_i$ *op* v_i where x_i is a variable ranging over a target class C_i. Each A_j denotes a complex attribute[10] from class C_{j-1} to class C_j, a_i is a simple attribute defined at class C_k, v_i is an atomic value from the domain of attribute a_i and *op* is a restriction operator from the set $\{=, \leq, <, \geq, >\}$. Each A_i defines an attribute link between the classes C_{j-1} and C_j, except A_1, which defines a link between the class of x_i and C_1. For example, the path expressions in query Q2 are `e.dept.comp.name = "IBM"` and `e.dept.name = "dbgroup"`.

Let **C** be the set of all distinct classes for which attribute links occur in `W`. A query graph then is defined as the undirected graph whose set of nodes is **C** and its set of edges consists of all (C_{j-1}, C_j) pairs for which a distinct attribute A_j appears in `W`. The nodes of classes for which a primitive attribute restriction of the form a_i *op* v_i appears in `W` are annotated by a label $(a_i, op, v_i, \mathtt{t})$, where `t` is the historical time specified in the `ON` field of the query (`t` is substituted by the temporal constant `Now` if no historical time is specified). For example, the query graph for the query Q2 is as shown in Figure 6.

5.2.2 Parameterized Operator Trees

Given a query graph, we first need to find an order in which the query graph nodes are to be processed, and second, to replace the conceptual entities appearing in the query graph with physical entities and operations for manipulating them. The first step is called the *execution ordering* generation step and its output is a set of *parameterized operator* trees (*PO* trees). This step is formulated as follows:

> **input:** A query graph QG
> **output:** A set of *PO* trees, each of which is denoted as, $J(\ldots J(J(C_1, C_2), C_3), \ldots C_n)$, and represents the different orderings of QG nodes, $C_1, C_2, \ldots C_n$ such that
>
> P1. C_1 is any node (class) in QG.
>
> P2. In QG, C_i is adjacent to one of the nodes in the sequence $C_1 \ldots C_{i-1}$.
>
> J is a parameterized operator which represents the intermediate results and is defined in more detail later.

[10] Recall that a simple attribute is an attribute whose values range over primitive domains, such as integers and strings. The values of a complex attribute range over non-primitive domains, for example, `Employee`. Also notice that the form of the query graph in the physical optimization is different from the one used in the syntactic simplification.

Another way to understand the *PO* tree is to view it as a left-deep tree representation of a possible traversal of the underlying query graph. More specifically, a *PO* tree is a binary tree in which every internal node has at least one leaf node from the set of nodes of the query graph. A non-leaf node of the tree represents the result obtained by applying the operation J to its children. As we will see in the sequel, this operation involves one of the standard join operations.

The rationale for focusing on left-deep trees, rather than the more general bushy trees, stems from three observations. First, the search space (that is, all possible access plans) in case of the generalized bushy trees becomes unmanageably large. Second, it has been argued that the space of left-deep trees covers the space containing the the execution strategies that incur the least cost, where cost is typically measured in terms of disk I/O required by the strategy (Steinbrunn, Moerkotte and Kemper, 1993). Third, in the controlled decomposition model adopted here, the left-deep trees better utilize the join index relations that are available.

The number of all possible *PO* trees for a query graph with n nodes is at most $n(n-1)$. Property P2 in conjunction with the connectivity of the query graph decreases the number of possible *PO* trees. As an example, in Figure 6, we show a possible *PO* tree for query Q2.

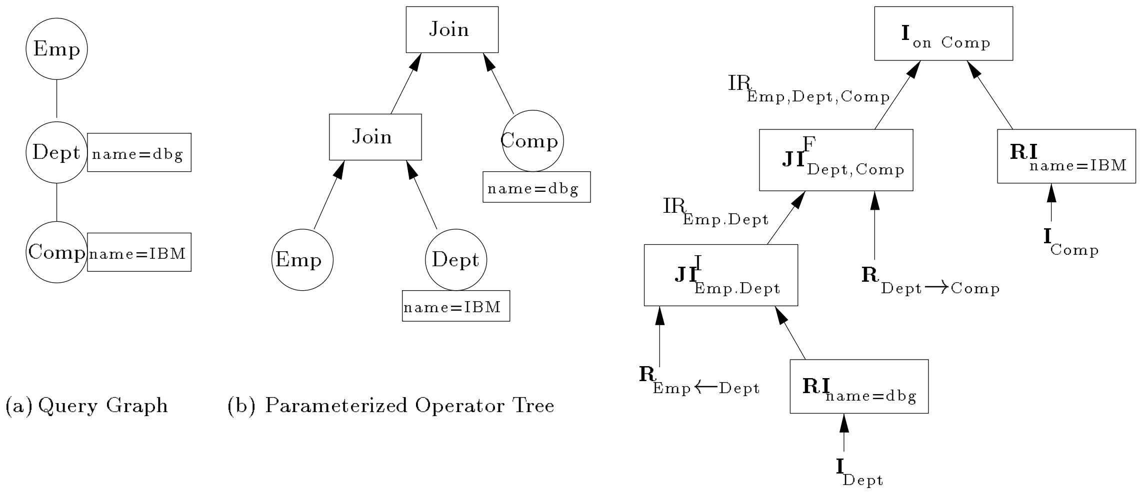
5.2.3 Execution Trees

An *execution tree* (ET) is a parameterized tree in which the exact steps for materializing the query result are specified. To this end, logical entities are replaced by physical storage relations as well as the low level operations that manipulate these relations. Recall that a logical class entity C_i is materialized in terms of a set of physical relations such that a physical relation R_{C_i} stores the subset of its primitive attributes. Furthermore, a physical join relation R_{C_i,C_j} materializes the join between classes C_i and C_j which is due to the complex attribute A of C_i. The final execution tree is expressed in terms of primitive operation being performed on these physical entities.

Before proceeding further, we will briefly present the primitive operations that can be used to access the data from the physical store. The following operations, which are also listed in Table 2, are available in our storage model:

- *Scan.* A scan operation, $S(R_C, P)$, accepts a predicate P and a physical relation R_C for class C as input and returns a list of OIDs for objects satisfying the predicate.
- *Index Retrieval.* An index retrieval, $RI(I_C, P)$, accepts as input a qualification predicate P on some attribute (possibly temporal) of a class C, and uses an index tree (B-tree for a non-temporal case, R-tree for the temporal case), denoted I_C, to return a list of OIDs for objects satisfying the predicate.
- *Forward Join Index.* An inverted join index operation, $JI^F(R_{C1\rightarrow C2}, L_{C1})$, accepts as input a list of object identifiers of class C_1 and uses the join index relation $R_{C_1\rightarrow C_2}$ to return the OIDs of class C_2 that are joined through the attribute link. If the list of OIDs of C_1 is not specified then the entire

Figure 6: AN EXAMPLE OF THE SUCCESSIVE STAGES IN THE PROCESSING OF QUERY Q2



(a) Query Graph (b) Parameterized Operator Tree (c) Execution Tree

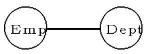
index relation is scanned.

- *Inverted Join Index.* A forward join index operation, $JI^I(R_{C1 \leftarrow C2}, L_{C2})$, is the inverse of the above operation except that now the OIDs of the domain class C_2 are used to look up the matching OIDs of class C_1. Here, the physical index relation $R_{C_1 \leftarrow C_2}$ is used instead.

- *Intersection.* An intersection operation, $I(L1, L2)$, takes as input two lists of object identifiers of the same class and returns the object identifiers that are common to both lists.

- *Object Fetching.* An object fetching operation, $F(R_C, L_C, P)$, takes a physical relation R_C for class C, a list L_C of OIDs and a predicate P as input. It then accesses the objects in the list and applies the predicate P thereby retaining only the ones satisfying P. Note that this operation is different from the scan operation in the sense that only a subset of objects are retrieved which may be dispersed over several pages.

In generating an execution tree from a *PO* tree, we substitute the nodes of the *PO* tree with the primitive operations along with the physical relations that they operate on. Each leaf node is labelled by one of the primitive operations (index retrieval or scan) and each non-leaf node is labelled by the join method (forward join index, inverted join index or intersection). The following two examples explain this process in detail.

Example: *Two-classes query.* The query graph for this query is: 

, and the *PO* tree is $J(Emp, Dept)$. Suppose also that there exists a restriction predicate `name="dbgroup"` on class *Dept*. Notice that this query is a sub-query of Q2. The following execution trees can be generated:

$ET1$: $JI^I(R_{Emp \leftarrow Dept}, RI(I_{Dept}, P_2))$

$ET2$: $F(R_{Dept}, JI^F(R_{Emp \rightarrow Dept}, \_)), name = \texttt{dbgroup})$

Table 2: COSTS OF BASIC OPERATIONS

Operation	**Cost**
Scan	$S(R_C, P) = \frac{\mid C \mid * SIZE(TUPLE_{R_C})}{B}$
Restriction Indexing	$RI(I_C, P) = h_{tree} + \sigma(P) * LP$
Inverted Join Index	$JI^I(R_{C1 \leftarrow C2}, L_{C2}) = \text{Yao}(\mid L_{C2} \mid, LP, V_{C,2} * r) + h_{tree_I}$
Forward Join Index	$JI^F(R_{C1 \rightarrow C2}, L_{C1}) = \text{Yao}(\mid L_{C1} \mid, LP, \mid C1 \mid * r) + h_{tree_F}$
Intersection	$I(L1, L2) = \mid L1 \mid \times \mid L2 \mid$
Object Fetching	$F(R_C, L_C, P) = \text{Yao}(\mid L_C \mid, M_{R_C}, \mid C \mid)$

Legend:

R_C	storage relation for class C	P	qualification predicate
σ	selectivity of a predicate	I_{C_a}	index relation for class C, attr. a
B	storage capacity of a page	L_C	list of OIDs of class C
Yao(K, M, N)	Yao's formula (Yao, 1977)	LP	Index pages at leaf level
$V_{C,i}$	unique instances of C in the domain of ith attribute	r_a	avg. number of historical values associated with each occurrence of attr. a
h_{tree}	height of an index tree	M_{R_C}	number of pages for storage relation R_C
$\mid C \mid$	cardinality of a class	$R_{C_1 \leftarrow C_2}$	join index relation

$ET3$: $I(JI^F(R_{Emp \rightarrow Dept}, \_\_), RI(I_{Dept}, name = \texttt{dbgroup}))$

Multiple classes query. Without any loss of generality we can assume the simple case of three classes. For convenience, we take this to be Q2. Figure 6(a) shows the query graph. There are four POs that are accepted by property P2:

$$J(J(Emp, Dept), Comp),\ J(J(Dept, Emp), Comp),\ J(J(Dept, Comp), Emp),\ J(J(Comp, Dept), Emp)$$

Let us now explore one of these PO trees, say $J(J(Emp, Dept), Comp)$ (Figure 6(b)). For these POs we can derive six ETs. As seen above, there are three possibilities for the $J(Emp, Dept)$ operation. Let $\texttt{IR}_{Emp,Dept}$ be the intermediate result of this operation. Finally, for the remaining $J(\texttt{IR}_{Emp,Dept}, Comp)$ step there are two execution possibilities:

$EP1$: $I(RI(I_{Comp}, name = \texttt{IBM}), JI^F(R_{Dept \rightarrow Comp}, \texttt{IR}))$

$EP2$: $I(\texttt{IR}, JI^I(R_{Dept \leftarrow Comp}, RI(Comp, name = \texttt{IBM})))$

Continuing this process for the rest of the PO trees, we can derive 24 possible execution trees. Figure 6(c) shows one of them ($ET1 + EP1$). The next section describes how to avoid generating the vast number of all possible ET trees.

5.2.4 The Selection of an Access Plan

As the number of classes in the query increases, it becomes prohibitively expensive to simply enumerate all the access plans in the strategy space. For a query with n classes, there are $n(n-1)$ parameterized trees generated, in the worst case. Unfortunately, the substitution of the primitive operations on the parameterized tree implies a $O(2^{n-1})$ size of the execution space that obviates the enumeration of all possible executions (Selinger et al., 1979).

In order to pick an execution tree with an optimal cost, we need a cost function that is used to compute the cost of access plan corresponding to each execution tree. Furthermore, we need a heuristic search algorithm to walk selectively the space of possible executions. The costs of the primitive operations that were shown in the previous section are given in Table 2. For our KBMS, we have adopted a heuristic search method based upon the enumeration of the most promising execution plan based on the selectivity of classes. Exploring randomized algorithms (Ioannidis and Kang, 1991) is a topic for future research.

5.2.5 Performance Analysis

In this section, our goal is to quantitatively evaluate the proposed join-index-based query processing methods. Specifically, this section compares the join index strategy to the sort-merge and nested-loop strategies. The sort-merge strategy represents the traditional database approach in computing large joins. The nested-loop strategy represents the traditional AI approach in query processing where processing is done in a tuple-oriented fashion (Bocca, 1986). Our key conclusion is that our choice of using the join index for evaluating joins performs better than the sort-merge and nested-loop methods.

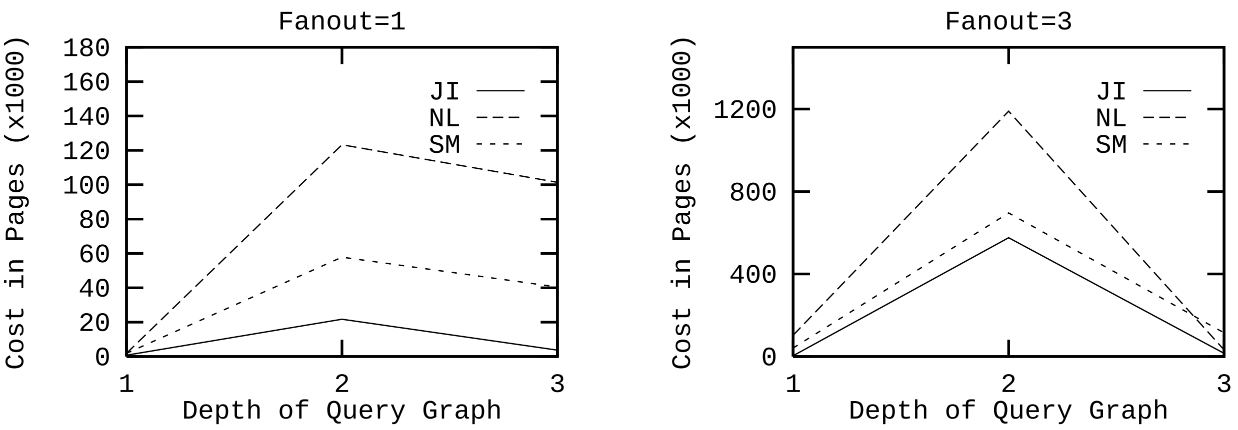
In our test knowledge base, the total number of classes is 40. We also varied the fanout, that connects each class with the others in the aggregation hierarchy, from one to three. The cardinality of classes at levels 0, 1, 2, and 3 are randomly selected from the intervals (20k..50k), (10k..20k), (5k..10k) and (1k..5k) respectively. Each class has two primitive attributes, one of which is indexed. Moreover, each attribute cardinality is randomly chosen from the range (5%..20%) of the domain class cardinality. Thus, these comprise the complex attributes in each class on which join indices are built. The fanout determines the length of aggregation hierarchies that are generated. The object size for each class is chosen at random between 200 and 300 bytes. It should be noted that we varied these parameters and the relative performance of the strategies remained unaffected. Also, the number of classes chosen was deemed reasonably large in view of statistics empirically observed in real knowledge bases (Chaudhri, 1995). The cardinalities of the classes and the primitive attributes were chosen randomly within reasonable ranges as in (Ioannidis, Ramakrishnan and Winger, 1993). This has thus ensured that the generation of data is not biased towards a particular strategy.

For the above knowledge base, random queries were generated with path expressions that had a given fanout and depth of query graph (QG). The fanout controls the number of complex attributes emanating from a given node in QG or, alternatively, the number of path expressions in the query qualification. The depth of the query graph, on the other hand, controls the length of the path expressions for each predicate. In addition, some of the nodes, that are chosen randomly, have a restriction predicate on a simple attribute.

For each join strategy, we keep track of the number of page I/Os incurred as we process each class (node) in the access plan. For the join-index, we make use of the cost model presented earlier to estimate the cost of performing the join. For nested-loop and sort-merge we use known formulae which are shown below. In these formulae, the parameter M_c denotes the number of data pages needed to store the objects of class c,

Figure 7: COST OF JOIN EXECUTION OVER PATH EXPRESSIONS

(a) (b)



and Mem denotes the size of the main memory cache in pages.

Nested-Loop (NL): $\frac{max(M_{c_1}, M_{c_2})}{Mem-1} min(M_{c_1}, M_{c_2}) + max(M_{c_1}, M_{c_2})$

Sort-Merge (SM): $M_{c_1} log M_{c_1} + M_{c_2} log M_{c_2} + M_{c_1} + M_{c_2}$

The results for fanout values of 1 and 3 are shown in Figure 7. For each graph, the fanout is fixed and then the depth is varied. Clearly, the join-index query strategy outperforms both the nested-loop and sort-merge in all configurations of the query graph. Note also that the nested-loop sometimes does better than sort-merge as in a knowledge base setup joins are not large. As we increase the number of path expressions and the depth, the cost may actually decrease since the addition of new predicates restricts the number of qualifying instances and in turn the size of intermediate results.

The superiority of the join-index strategy stems from its ability to focus only on the relevant OIDs as the algorithm traverses attributes moving from one class to another during the resolution of the query. On the contrary, the other two methods experience drastic deterioration whenever a class, with large cardinality, is encountered along the path.

These results experimentally establish that the join index based query processing is indeed a better approach for evaluating queries in a KBMS setting.

6 Concurrency Control

This section focuses on *locking-based* algorithms for knowledge bases, as these have been most successful in ensuring serializability for relational and object-oriented databases. The best known locking algorithm, *two phase locking* (2PL) (Eswaran et al., 1976), works along the following lines. Associated with each data item is a distinct "lock". A transaction must acquire a lock on a data item before accessing it. While a transaction holds a lock on a data item no other transaction may access it.[11] A transaction cannot acquire any additional locks once it has started releasing locks (hence the name "two phase" locking).

Transactions in a knowledge base system often access large number of data items. In such situations, 2PL implies that a transaction will have to hold each lock until it finishes acquiring all the locks that it will ever need, thereby locking most of the knowledge base for other users. Hence, concurrency is significantly reduced when running such "global" transactions. For this reason, our research has been directed towards the development of new methods that only hold a small number of locks at any one time, even for global transactions.

Interestingly, knowledge bases generally possess much richer internal structure than that of traditional databases (e.g., generalization and aggregation hierarchies, deductive rules, temporal dimensions defined in terms of history or belief time, etc.). Information about this structure can be potentially useful in allowing release of locks before a transaction has acquired all the locks that it needs. Indeed, concurrency control algorithm does exist for databases that have a directed acyclic graph structure (and is accordingly called *DAG policy* (Silberschatz and Kedem, 1980; Yannakakis, 1982)). Under the DAG policy, a transaction may begin execution by locking any item. Subsequently, it can lock an item if it has locked all the predecessors of that item in the past and is currently holding a lock on at least one of those predecessors. Moreover, under the DAG policy, a transaction may only lock an item once. The DAG policy exploits the assumption that there are no cycles in the underlying structure and the structure does not undergo any change. Unfortunately, such a policy cannot be adopted for knowledge bases without modifications. The structure of a knowledge base is likely to contain cycles (e.g., the inference graph generated for a collection of recursive rules) and will undergo change (e.g., when rules are added or deleted).

In summary, neither 2PL nor DAG policies are, by themselves, appropriate for knowledge bases. 2PL is too conservative, thereby causing reduced concurrency, while the DAG policy does not provide sufficient functionality. Accordingly, we are proposing a new graph-based policy, the *Dynamic Directed Graph policy* (DDG) that can handle cycles and updates in the knowledge base and also allows release of locks before a transaction commits, thereby promising better performance than 2PL.

6.1 The Dynamic Directed Graph Policy

For purposes of concurrency control, a knowledge base is a directed graph $G(V, E)$, where V is a set of nodes v_i (e.g., Employee in Figure 1), and E is a set of edges which are ordered pairs (v_i, v_j) of nodes (e.g., (Manager,Employee) in Figure 1). This graph includes as a subgraph the class schema mentioned in section 4 and also represents structural information about tokens and cross-references among deductive rules. There can be several alternative way to generate such a graph for a given knowledge base. In the performance experiments that we report later, the graph was created by representing each class and each token by a node and introducing an edge between two nodes if there is a semantic relationship (for example, partof or isA)

[11] In a simple generalization of this model, the transactions may hold *shared* and *exclusive* locks on data items.

between the two.

We first define some properties of directed graphs that are necessary for specifying our algorithm. A *root* of a directed graph is a node that does not have any predecessors. A directed graph is *rooted* if it has a unique root and there is a path from the root to every other node in the graph. A directed graph is *connected*, if the underlying undirected graph is connected. A *strongly connected component* (SCC) G_i of a directed graph G is a maximal set of nodes such that for each $A, B \in G_i$, there is a path from A to B. An SCC is *non-trivial* if it has more than one node. An *entry point* of an SCC, G_i, is a node B such that there is an edge (B, A) in G, A is in G_i, but B is not in G_i. Thus, if a node is an SCC by itself, its entry points are simply its predecessors.

The dominator D of a set of nodes W is a node such that for each node $A \in W$, either every path from the root to A passes through D or D lies on the same strongly connected component as A. Thus, in a rooted graph, the root dominates all the nodes in the graph including itself. All nodes on a strongly connected component dominate each other.

The DDG policy has three types of rules. *Preprocessing rules* convert an arbitrary graph to a rooted and connected graph. *Locking rules* specify how each transaction should acquire locks. *Maintenance rules* specify additional operations that must be executed by transactions to keep the structure rooted and connected. The rest of the discussion in this section focuses on locking rules. A detailed description of the DDG algorithm appears elsewhere (Chaudhri, 1995).

A transaction may lock a node in *shared* or *exclusive* mode (Bernstein, Hadzilacos and Goodman, 1987). Two transactions may simultaneously lock a node only if both lock it in shared mode. (The version of the DDG policy that supports both shared and exclusive locks is called the DDG-SX policy, but for the sake of brevity in the present paper, we continue to use the DDG policy to denote the DDG-SX policy.) The locking rules for the DDG policy are as follows:

L1. Before a transaction T performs any `INSERT, DELETE` or `WRITE` operation on a node A (or an edge (A, B)), T has to lock A (both A and B) in exclusive mode. Before T performs a `READ` operation on a node A (an edge (A, B)), it has to lock A (both A and B) in either mode.

L2. A node that is being inserted can be locked at any time.

L3. Each node can be locked by T at most once.

L4. The first lock obtained by T can be on any node. If the first node locked by T belongs to a non-trivial SCC, all nodes on that SCC are locked together in the first step.

Subsequently,

L5. All nodes on an SCC are locked together if:

L5a. All entry points of that SCC in the *present state* of G have been locked by T in past, and T is now holding a lock on at least one of them, and

L5b. For every node A on this SCC that is a successor of an entry point, and every path $A_1, \ldots, A_p, A, p \geq 1$, in the present state of the underlying undirected graph of G, such that T has locked A_1 (in any mode), and $A_2, \ldots, A_p$ in shared mode, T has not unlocked any of $A_1, \ldots, A_p$ so far.

Theorem 6.1 *The DDG-SX policy produces only serializable schedules (Chaudhri, 1995; Chaudhri and Hadzilacos, 1995).*

In general, the DDG policy does not permit concurrency within cycles (see rule L4 above). This suggests that if a knowledge base contains very large cycles which need to be locked as one node, concurrency will be reduced. We have a version of the DDG policy (called, DDG$'$ policy) that does permit concurrency within cycles (Chaudhri, Hadzilacos and Mylopoulos, 1992). We adopted the above version, because the transactions in knowledge bases tend to access all the nodes on a cycle together, and therefore, the cycles are a natural unit of locking.

In order for a transaction to be able to satisfy locking rules L3a and L3b for all the nodes that it needs to lock, it has to begin by locking the dominator of all the nodes that it is going to access. This is not a contradiction to locking rule L1, which just states that to lock the first node (which would usually be the dominator of the set of nodes that the transaction expects to lock), no other condition needs to be satisfied.

6.2 Implementation of the DDG Policy

There are two main issues in the implementation of the DDG policy. First, to enforce the rules of the locking policy, we need to compute and maintain information about several graph properties. Second, we need a mechanism to decide the order in which the locks should be acquired and released.

To enforce the locking rules, we need information on the dominator relationships and the strongly connected components within the knowledge base graph. In our implementation, the dominator tree of the knowledge base is computed at compile time using a bit vector algorithm (Lengauer and Tarjan, 1979). Using this information, the dominator of the set of nodes in the transaction can be computed in time linear in the length of a transaction using the nearest common ancestor algorithm (Schieber and Vishkin, 1988). The dominator information is maintained using and incremental algorithm (Carroll, 1988). The information on strongly connected components is computed at compile time in time $O(m)$ (Aho, Hopcroft and Ullman, 1987), where m is the number of edges in the graph. We developed a new algorithm for incrementally maintaining information on strongly connected components as the knowledge base evolves (Chaudhri, 1995) as algorithm for this purpose was not available.

Let us describe the order in which a transaction acquires and releases locks. A transaction always begins by locking the dominator of all the nodes that it might access. The dominator is computed on the assumption that a transaction may access all the descendants of the first node on which it requests a lock. Subsequently, every time a lock is requested, the locking conditions are checked, and if not enough predecessors are locked (rule L5a), lock requests for them are issued recursively. Before a node A can be unlocked by a transaction T, following conditions must be satisfied:

U1. A is no longer needed by T, and

U2. Releasing the lock on node A does not prevent the locking of any of its successors at a later stage in the execution of T (as required by rule L5a), and

U3. For every path $A, A_1, \ldots, A_p, B$ in the present state of the underlying undirected graph, such that A is locked (in any mode), $A_1, \ldots, A_p$ are locked in shared mode, T intends to lock B in future, T must not unlock any of $A, A_1, \ldots, A_p$ (by locking rule L5b).

To implement U1, we require T to send a message to the lock manager when it has finished processing a node.

To implement U2, we have to know how many of the descendants might be later locked by T. Moreover, of all the predecessors of a node A, only one has to be kept locked until T locks A. Therefore, we distinguish one of the predecessors that needs to be locked until all the successors have been locked, and associate with it the number of successors that are yet to be locked. Once the number of successors yet to be locked for a node becomes zero, U2 is satisfied for this node.

To implement U3, we check all the undirected paths from A to all the nodes that T may lock in future. U3 needs to be checked only when T acquires an exclusive lock or when T locks a node none of whose descendants will be locked by it in future. The check can be made more efficient by observing that if U3 is not satisfied for a node A, it is also not satisfied for descendants of A, and therefore, the paths passing through them need not be checked.

These data structures are integrated into the lock manager by maintaining an *unlock table*. This table is indexed on the node identifier and transaction identifier. An *unlock* record has three fields that were described above: *neededByTransaction, onePredecessor* and *count*. These entries are created when the transaction begins execution and are incrementally updated as the transaction progresses and as changes in the underlying structure of the graph occur.

6.3 Performance Results

The DDG policy has been implemented in the DeNet (Livny, 1986) simulation environment. Our performance model is similar to that presented in (Agrawal, Carey and Livny, 1987) and has four components: a *source*, which generates transactions, a *transaction manager*, which models the execution behavior of transactions, a concurrency control manager, which implements the details of a particular algorithm; and a resource manager, which models the CPU and I/O resources of the database. More details about the model and the simulations are available elsewhere (Chaudhri et al., 1994).

The primary performance metric adopted for the simulation is the response time of transactions in each class. We employ a batch means method for the statistical data analysis of our results, and run each simulation long enough to obtain sufficiently tight confidence intervals (90% confidence level, within 5% of the mean) (Law and Kelton, 1991).

Performance of the DDG policy was studied on a knowledge base under development for industrial process control. The objects represented in the knowledge base (boilers, valves, preheaters, alarms, etc.) are organized into a collection of classes, each with its own subclasses, instances and semantic relationships to other classes.

There are five kinds of relationships in this knowledge base. The `isA` relationship captures the class-subclass relationship, the `instanceOf` relationship represents the instances of a class, the `linkedTo` relationship stores how the components are linked to each other in the power plant, the `partOf` relationship indicates the part-subpart relationship. And finally, the `Equipment` relationship associates an equipment with each alarm.

For our experiments, we view this knowledge base as a directed graph. Each class and each instance is represented by a node. There are 2821 nodes in this graph. There is an edge between two nodes if they have some semantic relationship. For example, there is an edge from node A to node B, if the object represented by B is a part of the object represented by node A.

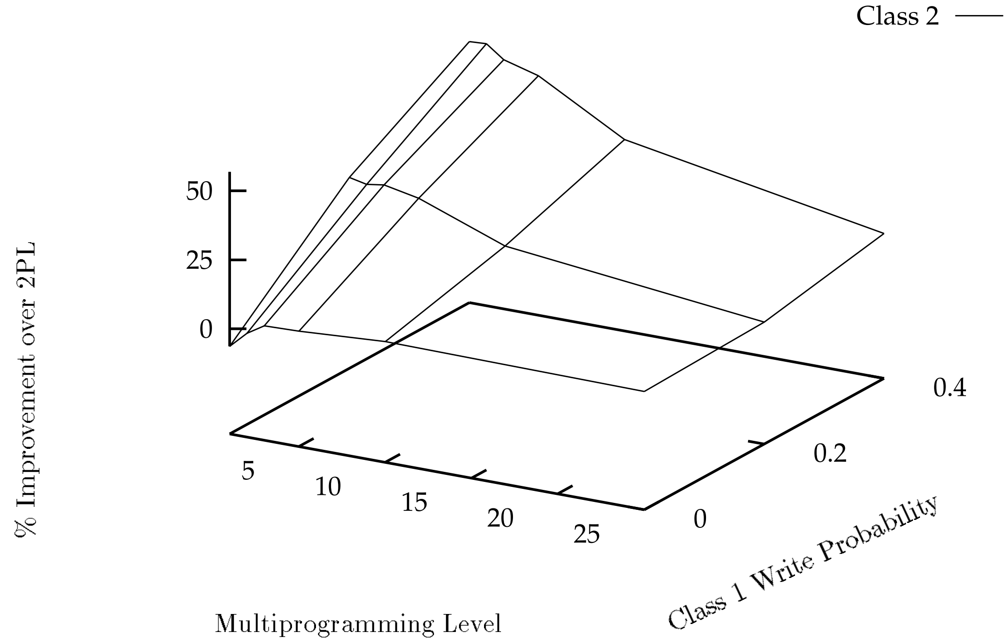
The graph corresponding to this knowledge base has cycles and undergoes changes. Moreover, the knowledge base receives two types of transactions. The first type consists of short transactions, which look-up or update an attribute value and occasionally change the structural relationships in the knowledge base (such as isA, partOf, etc.). The second class consists of long transactions which search the knowledge base along one of its structural relationships. The proportion of the transactions of the first type was determined to be 73% with the remaining 27% being of the second type.

Calibration of the simulation required determination of the relative running costs of the 2PL and DDG policies. For this, we divided running costs into three components: setup, locking and commit cost. For the DDG policy, the setup cost includes preprocessing of the transactions and generating information that will be used in prereleasing locks. It also includes the cost of creating entries in the lock table and the transaction table. For 2PL, no preprocessing of the transactions is required but entries in the transaction table and the lock table are created at setup time. The locking cost includes the CPU time required between the time a lock request was made and the time it was finally granted. If the transaction gets blocked, information on the processing used so far is maintained and, when it gets unblocked later, it is added to any further processing used. For the DDG policy, the cost of locking also includes the cost of checking for lock pre-release, cost of releasing locks and cost of unblocking of transactions in each call to the lock manager. The cost of commit includes the processing required to release all the locks and to finally commit the transaction. In general, the cost of commit for 2PL is higher as compared to the DDG policy. This is because under the DDG policy, a transaction would have already released several of its locks prior to commit whereas for the 2PL policy all the locks are released at commit time.

All simulation measurements were done on a DECStation 5000 (model 132). The values of overheads are subject to fluctuations. To maintain the consistency of results across different simulation runs, the values of overheads measured from the implementation were given as parameters to the simulation.

These parameters, along with the knowledge base and its associated transactions, were used as input to the simulation model. Preliminary experiments showed that for the parameter values of this application, the system response degrades du to a large number of long transactions and that the number of short transactions

Figure 8: COMPARISON OF DDG AND 2PL – PERCENTAGE IMPROVEMENTS



is not the key influencing factor. Therefore, we used a load control strategy in which the number of long transactions active at any time is controlled, whereas short transactions are processed as soon as they enter the system. In Figure 8, we plot the percentage improvement in response time of Class 2 transactions obtained by using the DDG policy as compared to 2PL. If $R(j)_{2PL}$ and $R(j)_{DDG}$ are the mean response times of transactions in class j, then the percentage improvement of the DDG policy over 2PL is computed as $100 \times (R(j)_{2PL} - R(j)_{DDG})/R(j)_{2PL}$.

The results for Class 2 transactions indicate that when Class 1 transactions are read only, the performance of the DDG policy is comparable to 2PL. When Class 1 transactions also perform some updates, the DDG policy can improve considerably (of the order of 50%) the response time of Class 2 transactions that are running concurrently. The results for Class 1 transactions are not shown here. We found that at low update probabilities there was slight degradation (about 10%) in Class 1 response time by using the DDG policy and at high update probabilities there was no significant difference between the two algorithms. These results make sense, because when there are only shared locks in a transaction, the DDG policy cannot allow any release of locks before its locked point, but incurs extra overhead, and therefore, leads to a slight degradation in response time. On the other hand, if the Class 1 transactions are update intensive, they release locks before their locked point, and the extra overhead is more than offset by the increased concurrency obtained due to lock pre-release leading to a net improvement in Class 2 response time.

In Figure 8, we can observe that the improvements are smaller at high multiprogramming levels. At low multiprogramming levels (for example, at MPL=1), any release of locks by a transaction before its locked point contributes to the improvement in response time of concurrently executing Class 2 transactions. At higher multiprogramming levels, this may not be necessarily true, because the lock released by a Class 1 transaction could be acquired by another Class 1 transaction giving no benefit to Class 2 transactions. As a result, we see greater improvements in the Class 2 response time at low multiprogramming levels of Class 1 as compared to improvements at high multiprogramming levels.

The overall system response time can be computed as the weighted sum of the individual class response times where class throughputs are used as weights. In the APACS workload the throughput of Class 2 (25 transactions/second) is much higher than the throughput of Class 1 (approximately 0.4 transactions/second), and therefore, the behavior of Class 2 transactions dominates the overall response time. On computing the improvements in the overall response time, we found that improvements were very close to the corresponding values for Class 2 response time as shown in Figure 8.

In view of the relative behavior of the two algorithms, we designed an adaptive scheme which can switch between 2PL and the DDG policy depending on the load conditions. Such a scheme uses 2PL at low write probabilities and the DDG policy at high write probabilities, thus giving the best of the two algorithms. Since simultaneously using the two algorithms may lead to non-serializable schedules the adaptive scheme uses a transition phase while switching from one algorithm to the other. While switching from 2PL to the DDG policy, transactions locked according to the DDG policy are forced to behave as two-phase locked transaction by delaying the release of lock until locked point. While switching from the DDG to 2PL, the adaptive scheme waits until all active transactions running under the DDG policy finish execution. This results in an algorithm which is a hybrid of 2PL and the DDG policy and performs consistently better than a concurrency control algorithm that only uses 2PL.

Of course, an important consideration in the choice between the two algorithms is the relative complexity of the two algorithms. The DDG policy is more complex than 2PL, and therefore, its implementation requires a greater effort. For example, in our simulation, 2PL required approximately 1000 lines of code and the DDG policy required 3000 lines of code. Our simulations incorporate the cost of extra complexity by measurements of runtime overhead of the two algorithms and show that there is a net improvements in response times. Thus, the final choice between the DDG policy and 2PL has to be made by the implementor who has to evaluate whether the improvements showed above are worth the added complexity of the DDG policy.

Even though our experiments are based on a knowledge base of rather small size, we can predict the relative performance of the two algorithms on larger knowledge bases. It has been shown that the performance results of a 2PL algorithm on a knowledge base of size of D entities with N active transactions are indicative of its performance on another knowledge base of size bD entities with bN active transactions, where $b > 0$ (Tay, 1987). In a similar fashion we expect that our results are representative of the relative performance of 2PL and the DDG policy on knowledge bases that are larger and have greater number of active transactions. Another important aspect in scaling up of these results to larger knowledge bases is the run-time cost for incremental maintenance of graph properties, which include, strongly connected components and dominator relationships. The run-time cost of incremental algorithms is highly dependent on the nature of changes in the graph. For example, if the insertions and deletions to the graph cause only local changes in a graph property, the run-time cost is minimal. On the other hand, if the changes are such that they lead to a change in a substantial part of the solution, the computational cost can be excessive. It is difficult to comment on this aspect in the absence of a specific knowledge base and its workload. In general, if the database is highly structured (Chaudhri, 1995) the locality of changes is almost assured, and the incremental algorithms, and therefore, the results presented in this section will scale up to larger problems

The present section has presented an overview of the results based on the second author's doctoral dissertation (Chaudhri, 1995) which have also been published as conference length papers (Chaudhri, Hadzilacos and Mylopoulos, 1992; Chaudhri et al., 1994; Chaudhri and Hadzilacos, 1995; Chaudhri and Mylopoulos, 1995).

7 Integrity Constraint and Rule Management

Integrity constraints specify the valid states of a knowledge base (*static* constraints) as well as the allowable knowledge base state transitions (*dynamic* constraints). Integrity constraints are used to express general, complex semantic relationships that cannot be built-into the data structures used to represent knowledge. Such relationships may refer to state transitions or histories (Plexousakis, 1993a).

As an example, consider the following integrity constraints on the knowledge base of Section 2. (These constraints are expressed in an equivalent form without using the meta-predicate `Holds`.)

`IC1:` $\forall p/\texttt{ConfPaper}\ \forall x/\texttt{Author}\ \forall r/\texttt{Referee}\ \forall t_1,t_2,t_3,t_4/\texttt{TimeInterval}$

$[author(p,x,t_1,t_2) \wedge referee(p,r,t_3,t_4) \wedge during(t_3,t_1) \wedge during(t_4,t_2)] \Rightarrow [r \neq x]$

`IC2:` $\forall c/\texttt{Conference}\ \forall p/\texttt{ConfPaper}\ \forall a/\texttt{Author}\ \forall d/\texttt{Department}\ \forall t_1,t_2/\texttt{TimeInterval}$

$(submitted\_to(p,c,t_1,t_2) \wedge organized\_by(c,d,t_1,t_2) \wedge author(p,a,t_1,t_2) \wedge works\_in(a,d,t_1,t_2)\ \Rightarrow \texttt{False})$

`IC3:` $\forall p/\texttt{Employee})(\forall s,s'/\texttt{Integer}\ \forall t_1,t_2,t_3/\texttt{TimeInterval}$

$(salary(p,s,t_1,t_2) \wedge salary(p,s',t_3,t_2) \wedge \texttt{before}(t_1,t_3) \Rightarrow (s \leq s'))$

`IC4:` $\forall p,c,l/\texttt{Proposition}\ \forall t,t'/\texttt{TimeInterval}\ (\texttt{prop}(p,c,l,c,t) \wedge \texttt{instanceOf}(p,\texttt{Class},t,t') \Rightarrow$

$(\forall T,T'/\texttt{TimeInterval}\ (\texttt{overlaps}(t,T) \wedge \texttt{overlaps}(t',T') \Rightarrow\ \texttt{instanceOf}(p,\texttt{Class},T,T')))$

Constraints `IC1` and `IC2` are *static*, expressing the properties that "*no author of a paper can be its referee*" and "*an author cannot submit a paper to a conference organized by the department she works in*" respectively. Constraint `IC3` enforces the property that "*an employee's salary can never decrease*". This constraint expresses a *transitional* property as it refers to more than one state of the domain being modeled. Constraints referring to multiple domain states are called *dynamic*. The last of the above formulae is an example of a dynamic epistemic (meta-) constraint expressing the property that "*the system cannot stop believing a class definition*". Constraints of this type refer to multiple knowledge base states in addition to multiple domain states.

The above types of constraints are significantly more general than functional dependencies, type constraints and other types of constraints traditionally supported in relational or object-oriented databases.

In particular, these constraints contain semantic information in the form of aggregation and generalization relationships, and represent temporal knowledge.

The presence of deductive rules in a knowledge base is an additional impediment to the problem of constraint enforcement because implicitly derived knowledge may affect the properties specified by the constraints. For example, consider the following deductive rules:

`DR1:` $\forall u/$`UnivAffiliate` $\forall d/$`Department` $\forall s, s'/$`String` $\forall t_1, t_2/$`TimeInterval`
$$(address(u, s, t_1, t_2) \;\wedge\; D\_addr(d, s', t_1, t_2) \;\wedge\; (s = s') \Rightarrow works\_in(u, d, t_1))$$
`DR2:` $\forall d/$`Department` $\forall u/$`University` $\forall s/$`String` $\forall t_1, t_2/$`TimeInterval`
$$(univ(d, u, t_1, t_2) \wedge location(u, s, t_1, t_2) \Rightarrow D\_addr(d, s, t_1))$$

`DR1` and `DR2` express the rules that "*A university affiliate works in the department that has the same address as she does*" and "*A university department's address is the same as the university's location*" respectively. Looking at the expressions of rule `DR1` and constraint `IC2`, it can be easily seen that an update in any of the literals in the body of `DR1` may change the truth value of its conclusion literal which occurs in `IC2` and thus violate `IC2`. In the same fashion, facts implicitly derived using `DR2` may trigger the evaluation of `DR1` and, as before, violate constraint `IC2`.

The key issue in our research has been to devise an efficient method for *constraint checking*. Constraint checking consists of verifying that all constraints remain satisfied in the state resulting from an update to the knowledge base. Constraint checking constitutes a major performance bottleneck and most commercially available database systems only guarantee very limited types of integrity constraint checking, if at all. *Incremental* integrity checking methods are based on the premise that constraints are known to be satisfied prior to an update. Accordingly, only a subset of the constraints needs to be verified after the update, namely those that are affected by it. Moreover, incremental integrity checking can be further optimized by specializing integrity constraints with respect to the anticipated types of updates and by performing simplifications on the specialized forms. In the presence of deductive rules an integrity checking method must account for implicit updates induced by the interaction of explicit updates and deductive rules. In other words, it has to be verified, that no implicitly generated insertion or deletion may violated any of the constraints. The method we propose in this section is an incremental compile-time simplification method that accounts for implicit updates as well as temporal knowledge.

A last issue in constraint enforcement is that of *integrity recovery*, i.e. the undertaking of appropriate action for restoring the consistency of the knowledge base once it has been violated by some updating transaction. At present, we adopt a coarse-grained approach to integrity recovery, namely the rejection of any integrity violating transaction. A transaction is not committed until all constraints are found to be satisfied. [12]

[12] A finer grained approach could initiate a sequence of updates that change the KB so that constraints are satisfied in the resulting state.

7.1 Inadequacies of Existing Methods

A number of incremental constraint checking techniques for relational (e.g. (Nicolas, 1982), (Ceri and Widom, 1990),(Ling and Lee, 1992), (Gupta et al., 1994),), deductive (e.g. (Decker, 1986), (Bry, Decker and Manthey, 1988), (Kuchenhoff, 1991)), object-oriented (Jeusfeld and Jarke, 1991) and temporal databases (Chomicki, 1992), have appeared in the recent literature. A complementary approach, which modifies transactions prior to their execution to ensure knowledge base integrity, is studied in (Stonebraker, 1975) and (Wallace, 1991). Along the same lines, a transaction modification technique for temporal constraints has been proposed in (Lipeck, 1990), but does not account for implicit updates. A transaction modification method for temporal constraints and implicit updates appears in (Plexousakis, 1995). Transaction modification is less flexible than constraint simplification since each transaction has to be modified for each relevant constraint.

Most promising, as a basis for enforcing more expressive constraints such as the ones expressible in the assertion language of Telos, are the *compilation method* of (Bry, Decker and Manthey, 1988) and the *historical knowledge minimization* techniques of (Hulsmann and Saake, 1990) and (Chomicki, 1992). The former method, extended with the ability to deal with object identity, aggregation and classification, has been used in the integrity subsystem of the deductive object base ConceptBase (Jeusfeld and Jarke, 1991) (also based on a version of Telos). However, this method does not deal with temporal or dynamic constraints. Historical knowledge minimization techniques assume a formulation of constraints in temporal logic and attempt to minimize the historical information required in order to verify the constraints. They are applicable to relational databases only and assume a fixed set of constraints. Thus, neither of these methods by itself is sufficient to deal with the integrity constraints of our KBMS.

This section describes a novel method for compile-time simplification of temporal integrity constraints in a deductive object-oriented setting. The method uses a comprehensive compilation and simplification scheme that leads to an efficient implementation of constraint checking by allowing us to precompute implicit updates at compile time. Moreover, a number of optimization steps, including temporal simplification, are performed at compile time so that the resulting forms are easier to evaluate at update time. The compilation scheme allows for dynamic insertion or removal of integrity constraints and deductive rules without having to recompile the entire knowledge base. We describe the method in more detail in (Plexousakis, 1993a) .

7.2 A Constraint Enforcement Algorithm

Our constraint enforcement method operates in two phases: *compilation* and *evaluation*. Compilation is performed at schema definition time and leads to the organization of simplified forms of rules and constraints into a *dependence graph*, a structure that reflects their logical and temporal interdependence. The evaluation phase is performed every time there is an update to the knowledge base. This phase is responsible for enforcing the constraints and incrementally maintaining the dependence graph that was generated during compilation. We will first give some definitions and then describe the compilation and evaluation phases in more detail.

7.2.1 Formal Framework

Integrity constraints are expressed declaratively as rectified[13] closed wffs of the assertion language. An integrity constraint can have one of the following two forms:

$$\mathtt{I} \equiv \forall x_1/\mathtt{C}_1 \ldots \forall x_k/\mathtt{C}_k F, \text{ or}$$
$$\mathtt{I} \equiv \exists x_1/\mathtt{C}_1 \ldots \exists x_k/\mathtt{C}_k \; F$$

where F is any well-formed formula (wff) of the assertion language whose quantified subformulae are of the above forms and in which the variables $x_1, \ldots, x_k$ are free variables. Each $\mathtt{C}_i$ is a Telos class and the meaning of each restricted quantification is that the variable bound by the quantifier ranges over the extension of the class instead of the entire domain. Any constraint in this form is *range-restricted* (Decker, 1986).[14]

The typed quantifications $\forall x/\mathtt{C}\ F$ and $\exists x/\mathtt{C}\ F$ are short forms for the formulae:

$$\forall x\ \forall t\ \mathtt{instanceOf}(x, \mathtt{C}, t) \wedge \mathtt{instanceOf}(t, \mathtt{TimeInterval}, \mathtt{Alltime}) \Rightarrow F \qquad ,$$
$$\exists x\ \exists t\ \mathtt{instanceOf}(x, \mathtt{C}, t) \wedge \mathtt{instanceOf}(t, \mathtt{TimeInterval}, \mathtt{Alltime}) \wedge F$$

The introduction of temporal variables and their restricting literals is necessary since all atomic formulae of the assertion language have a temporal component.

Deductive rules are considered to be special cases of integrity constraints. Their general form is:

$$\mathtt{DR} \equiv \forall x_1/\mathtt{C}_1 \ldots \forall x_n/\mathtt{C}_n \ (F \Rightarrow A)$$

where F is subject to the same restrictions as above and A is an atom of the assertion language. In addition deductive rules are assumed to be *stratified*[15] (Ullman, 1988).

Let us now introduce some terminology that we will use in the rest of the section.

Definition 7.1 *An update is an instantiated literal. A positive literal is considered as an insertion, whereas a negative literal is considered as a deletion.*

Given the general form of constraints, it can be seen that a constraint is affected by an update only when a "tuple" is inserted into the extension of a literal occurring negatively in the constraint, or when a "tuple" is deleted from the extension of a literal occurring positively in the constraint. The definition of *relevance* found in (Jeusfeld and Jarke, 1991) is not sufficient in the presence of time. The following definition provides sufficient conditions for "relevance" of a constraint to an update, by considering the relationships of the time intervals participating in the literals of the constraint and the update.

Definition 7.2 *(*Affecting Update*) An update $U(\_,\_,t_1,t_2)$ is an affecting update for a constraint I with history and belief time intervals T and T' respectively, if and only if there exists a literal $L(\_,\_,\_,\_)$ in I such that L unifies with the complement of U and the intersections $t_1 * T$ and $t_2 * T'$ are non-empty.*

[13] A formula is rectified if no two quantifiers introduce the same variable (Bry, Decker and Manthey, 1988).
[14] This class of constraints is equivalent to both the *restricted quantification form* of (Bry, Decker and Manthey, 1988) and the *range form* of (Jeusfeld and Jarke, 1991).
[15] A set of rules is stratified if whenever a negated literal P occurs in the body of a rule with head Q, there is no derivation path for P in which Q occurs in the body of a rule.

Similar to the above definition, we define the notion of *concerned class* for a literal L. The role of a concerned class is to limit the search space for constraints affected by an update. This is possible because of the fine granularity — not found in relational databases — provided by aggregation. Determining concerned classes for literals in constraints takes into account the instantiation and specialization relationships in the knowledge base.

Definition 7.3 *(*Concerned Class*) A class $C(\_,\_,t_1,t_2)$ is a concerned class for a literal $L(\_,\_,T,T')$ if and only if inserting or deleting an instance of C affects the truth of L, the intersections $t_1 * T$ and $t_2 * T'$ are non-empty and C is the most specialized class with these properties.*

The notions of *dependence* and *direct dependence* are used to characterize the logical and temporal interdependence between rules and constraints and the generation of implicit updates. A *dependence graph* is a structure representing such a dependence relation for a set of deductive rules and integrity constraints.

Definition 7.4 *(*Direct Dependence*) A literal L directly depends on literal K if and only if there exists a rule of the form $\forall x_1/\mathtt{C}_1 \ldots \forall x_n/\mathtt{C}_n \; (F \Rightarrow A)$ such that, there exists a literal in the body F of the rule unifying with K with most general unifier θ and $A\theta = L$.*

Dependence is the transitive closure of *direct dependence.*

Definition 7.5 *(*Dependence*) A literal L depends on literal K if and only if it directly depends on K or depends on a literal M that directly depends on K.*

For example, literal *works_in* of constraint `IC2` directly depends on literal *D_addr* of rule `DR1` and (transitively) depends on literal *location* of rule `DR2`.

7.2.2 Compilation Phase

The compilation process employs a number of rules for *parameterized form generation* and *formula simplification.* The former type of rules generate a simplified parameterized form for each integrity constraint and deductive rule defined. Formula simplification rules apply a number of simplification steps including temporal simplifications to the original expressions in order to produce more easily evaluable forms. The final step in the compilation phase is *dependence graph generation.* This incremental process accepts as input the parameterized forms generated and represents them in the form of a dependence graph. In the following paragraphs, we describe the application of the above rules and the generation of the graph structure in more detail.

Parameterized Form Generation

For each literal l occurring in a constraint or the body of a deductive rule, the compilation process generates a *Parameterized Simplified Structure* (PSS). A PSS is a 6-tuple $\langle l, p, c, h, b, s\rangle$, where l is the literal, p is the list of instantiation variables [16] occurring in l, c is the concerned class of l, h is the history time and b is the belief time of the constraint or rule with respect to which the simplified form is generated. A PSS allows us to efficiently retrieve the constraints or rules affected by an update by indexing with respect to time, characteristic literal or class. The rules for the generation of parameterized forms are given below. Rules PF1a to PF1d are used to determine concerned classes for literals. Rules PF2 to PF5 are used to simplify the formulas and generate the parameterized forms.

PF1. For each literal appearing in an integrity constraint or deductive rule, compute the concerned class as follows:

PF1a. `Instantiation literals`: for literals of the form `instanceOf`(x, y, t_1, t_2), if y is instantiated, then y is the concerned class provided this class exists during t_1 and its existence is believed during t_2; otherwise, the built-in class `InstanceOf` is the concerned class.

PF1b. `Generalization literals`: for literals of the form `isA`(x, y, t_1, t_2) where both x and y stand for classes, the concerned class is the built-in class `isA`, since the truth of an `isA`-literal does not depend on the insertion/deletion of instances to/from the extensions of x and y.

PF1c. `Attribute literals`: for a literal of the form $att(x, y, t_1, t_2)$, where att is an attribute of the class x, if both x and y are un-instantiated then the concerned class of the literal is the unique attribute class q with components `from`$(q) = X$, `label`$(q) = att$, `to`$(q) = Y$ and `when`$(q) = T$, that is such that x is an instance of X for t_1, y is an instance of Y for t_1 and both these are believed during t_2. In other words, the most specialized concerned class is the attribute class that includes all instantiated attributes that relate objects x and y of types X and Y respectively, under the assumption that to each attribute literal of the assertion language, corresponds a unique proposition with the above properties.

PF1d. For a literal of the form `prop`(p, x, y, z, t), if the components x and z are equal, then the concerned class is the built-in class `Individual`; if not, the concerned class is the class `Attribute`. In case none of x and z are instantiated, the concerned class is the class `Proposition`. However, `prop` literals will not be considered in the generation of simplified forms, because it is assumed that for every proposition p with components x, y, z, t, there exist token propositions in the knowledge base defining the components.[17]

PF2. Drop the quantifiers binding the instantiation variables and set the parameter list to be the list of instantiation variables.

PF3. Constrain the temporal variables with respect to the history and belief times of the constraint using the `during` temporal predicate and conjoin them with the constraint or rule.

[16] Instantiation variables are universally quantified and are not governed by an existential quantifier.
[17] It is possible to express *referential integrity* in Telos as a meta-constraint in the assertion language.

Table 3: ABSORPTION RULES

$\phi \wedge \texttt{True} \equiv \phi$	$\phi \wedge \texttt{False} \equiv \texttt{False}$	$\phi \Rightarrow \texttt{True} \equiv \texttt{True}$	$\phi \Rightarrow \texttt{False} \equiv \neg\phi$
$\phi \vee \texttt{True} \equiv \texttt{True}$	$\phi \vee \texttt{False} \equiv \phi$	$\texttt{True} \Rightarrow \phi \equiv \phi$	$\texttt{False} \Rightarrow \phi \equiv \texttt{True}$
$\neg\texttt{True} \equiv \texttt{False}$	$\neg\texttt{False} \equiv \texttt{True}$	$\phi \Leftrightarrow \texttt{True} \equiv \phi$	$\phi \Leftrightarrow \texttt{False} \equiv \neg\phi$

PF4. Substitute the atom into (from) whose extension an insertion (deletion) takes place by the Boolean constant `True` (`False`) since after the update it is known that the fact expressed by the literal is true (false respectively).

PF5. Apply absorption rules (see table 3) until no further simplification is possible.

PF6. Apply temporal simplification rules (see table 7.2.2) until no further simplification is possible.

Example: The concerned class for literal *author* of constraint `IC1` is the attribute class defined by the proposition (`Paper`, *author*, `Author`, t), with t satisfying the properties of rule PF1c. Assume that the history and belief time intervals of the constraint are the intervals (`01/01/1988..*`) and (`21/01/1988..*`) respectively. Applying the rest of the rules to constraint `IC1` for an insertion $author(\texttt{P},\texttt{X},\texttt{T},\texttt{T}')$ yields:

$\forall r/\texttt{Referee}\ \forall t_3, t_4\ \texttt{TimeInterval}(referee(\texttt{P}, r, \texttt{t}_3, \texttt{t}_4) \wedge \texttt{during}(\texttt{t}_3, \texttt{T}) \wedge \texttt{during}(\texttt{t}_4, \texttt{T}') \wedge$
$\texttt{during}(\texttt{t}_3, (\texttt{01/01/1988..*})) \wedge \texttt{during}(\texttt{t}_4, (\texttt{02/01/1988..*})) \Rightarrow (r \neq \texttt{X}))$

At update time, if the constraint is affected by the update, the form that will have to be verified is
$\forall r/\texttt{Referee}\ (referee(\texttt{P}, r, \texttt{t}_3, \texttt{t}_4) \wedge \texttt{during}(\texttt{t}_3, \texttt{T}) \wedge \texttt{during}(\texttt{t}_4, \texttt{T}') \Rightarrow (r \neq \texttt{X}))$ $\diamond$

Let us briefly comment on the application of compilation to dynamic (epistemic) constraints. The expressions of dynamic constraints may contain atoms occurring more than once since the properties expressed refer to two or more states. In such a case the compilation process will generate one PSS for each literal occurrence. The forms will differ in their parameter lists, as well as in their simplified forms. The original constraint will be violated if any of its simplified forms is. However, in such a case, not all occurrences of the literal can be replaced by their truth values on the basis that both the update and the fact that constraints were satisfied before the updates are known. This would be possible only if it were known that the constraints were non-trivially satisfied in the previous state. A logical implication is trivially satisfied if its antecedent is false. This kind of knowledge requires the maintenance of meta-level information about the satisfaction of constraints. For the moment we will assume that no such knowledge is available and that a PSS is generated for each literal occurrence in an integrity constraint. The following example shows the application of the compilation process in the case of a dynamic constraint.

Example: Assume that the history and belief time intervals of constraint `IC3` of our working example are T and T' respectively. The literal *salary* occurs twice in the expression of the constraint. Only one of the history time variables t_1 and t_3 will be instantiated in each of the two forms. It is known that the constraint is satisfied before an update to a *salary* literal occurs. Hence, according to the current beliefs of the system, either all employees have not had a change in salary, or for those that have had a salary change, this change was an increase. If no information exists about whether the satisfaction of the constraint prior to the update is strict or trivial, the following two forms can be generated by the compilation process:

$$\forall s/\texttt{Integer}\ \forall t_1/\texttt{TimeInterval}\ (salary(p,s,t_1,t_2)\ \wedge\ \texttt{during}(t_1,T)\ \wedge\ \texttt{during}(t_3,T)$$
$$\wedge\ \texttt{before}(t_1,t_3)\ \wedge\ \texttt{during}(t_2,T') \Rightarrow (s \leq s'))$$
$$\forall s'/\texttt{Integer}\ \forall t_3/\texttt{TimeInterval}\ (salary(p,s',t_3,t_2)\ \wedge\ \texttt{during}(t_1,T)\ \wedge\ \texttt{during}(t_3,T)$$
$$\wedge\ \texttt{before}(t_1,t_3)\ \wedge\ \texttt{during}(t_2,T') \Rightarrow (s \leq s'))$$

Were it known that `IC3` was non-trivially satisfied, only one simplified form would be generated, namely the form resulting from dropping all quantifiers from the above forms and replacing the *salary* literals by `True`. If however it was trivially satisfied before the update, i.e., at least one of the *salary* literals was false or the temporal constraint was violated, then the *salary* literals cannot be eliminated. ◇

Temporal Simplification

The objective of temporal simplification rules is to simplify a conjunction of temporal relationships into a single temporal relationship. In its full generality this task is intractable (Allen, 1983). In our method, however, we require that at least one of the temporal variables in each temporal relation should be instantiated, and with this condition the simplification can be performed efficiently. In fact only a table lookup is required.

Formally, the problem of temporal simplification is stated as follows: given a conjunction $\texttt{during}(t,i_1) \wedge \texttt{r}_1(t,i_2) \wedge \texttt{r}_2(i_1,i_2)$, where $\texttt{r}_1$ and $\texttt{r}_2$ are any of the 13 possible relationships (or their negation) between any two time intervals, and i_1, i_2 are known time intervals, find a temporal relationship $\texttt{r}$ and an interval i such that $\texttt{r}(t,i)$ is satisfied if and only if the original conjunction is satisfied. For some combinations of $\texttt{r}_1$ and $\texttt{r}_2$, $\texttt{r}$ is a disjunction of temporal relationships. In those cases, and for the sake of completeness, we do not replace the original expression by the equivalent disjunction. Table 4 shows the simplified forms obtainable from the various combinations of temporal relationships for $\texttt{r}_1$ and $\texttt{r}_2$. **F** denotes a logical contradiction and the cases where no simplification is possible without introducing disjunction, are denoted by the entry "**no simp.**".

Example: Consider the conjunction:

$$\texttt{during}(t, 01/88..09/88) \wedge \texttt{before}(t, 05/88..12/88) \wedge \texttt{overlaps}(01/88..09/88, 05/88..12/88).$$

Using Table 4, it can be simplified into $\texttt{during}(t, 01/88..05/88)$. ◇

Table 4: TEMPORAL SIMPLIFICATION TABLE

r1(t,i2) / r2(i1,i2)	b	d	o	m	s	f	e	a	c	ob	mb	sb	fb
before (b)	during i1	F	F	F	F	F	F	F	F	F	F	F	F
during (d)	F	during i1	F	F	F	F	F	F	F	F	F	F	F
overlaps (o)	during i1-i1*i2	during i1*i2	overlaps i1*i2	finishes i1-i1*i2	starts i1*i2	F	F	F	F	F	F	F	F
meets (m)	during i1	F	F	F	F	F	F	F	F	F	F	F	F
starts (s)	F	during i1	F	F	F	F	F	F	F	F	F	F	F
finishes (f)	F	during i1	F	F	F	finishes i1	F	F	F	F	F	F	F
equal (e)	F	during i1	F	F	F	F	F	F	F	F	F	F	F
after (a)	F	F	F	F	F	F	F	during i1	F	F	F	F	F
contains (c)	during (i1-,i2-)	during i2	no simp.	finishes (i1-,i2-)	starts i2	finishes i2	equal i2	during (i2+,i1+)	no simp.	no simp.	starts (i2+,i1+)	starts (i2-,i1+)	finishes (i1-,i2+)
overlapped by (ob)	F	during (i1-,i2+)	F	F	F	finishes (i1-,i2+)	F	during (i2+,i1+)	F	no simp.	starts (i2+,i1+)	F	finishes (i1-,i2+)
met by (mb)	F	F	F	F	F	F	F	during i1	F	F	F	F	F
started by (sb)	F	during i2	F	F	F	finishes i2	F	during i1-i2	F	no simp.	starts i1-i2	F	F
finished by (fb)	during (i1-,i2-)	during i2	no simp.	finishes (i1-,i2-)	starts i2	F	F	F	F	F	F	F	finishes i1

Legend

F: false
* : intersection operator
\- : difference operator
no simp. : no simplification possible
t- : left endpoint
t+ : right endpoint

Formal Properties of Simplification

The following properties have been proven for the simplification method described in the previous paragraphs. Detailed proofs can be found elsewhere (Plexousakis, 1995).

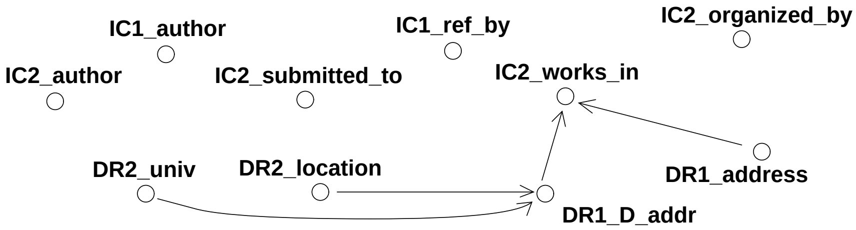
Theorem 7.1 *The simplification rules PF1-PF6 are sound. Temporal simplification (rule PF6) is also complete.*

The simplification method consists of a number of truth-preserving transformations that produce formulae which, if proven not to be satisfied in the resulting knowledge base state, imply that the original formulae are not satisfied. Moreover, no inconsistency can be introduced by any of the simplification steps. Temporal simplification is also complete in the sense that all possible temporal transformations are performed. No transformation takes place in those cases where the derived temporal relationship is a disjunction of temporal predicates.

Dependence Graph Organization

This section describes the organization of compiled forms of integrity constraints and deductive rules into a dependence graph. The graph is used for computing the effects (direct or indirect) of updates on integrity constraints.

Figure 9: A DEPENDENCE GRAPH



The notions of *dependence* and *direct dependence* have already been defined in Subsection 7.2.1. The dependence graph for a knowledge base KB with a set of integrity constraints I and a set of deductive rules R is defined as follows:

Definition 7.6 *Given $KB = (\_, I, R)$, the dependence graph for KB is a directed graph $G = (V, E)$ with the following properties. There is a node $v \in V$ corresponding to each parameterized simplified structure (PSS) of each deductive rule and integrity constraint. The set V is partitioned into two disjoint sets V_I and V_R where each $v \in V_I$ is a PSS corresponding to a literal appearing in an integrity constraint and each $v \in V_R$ is a literal appearing in a deductive rule. There is an edge $(u, v) \in E$ between nodes $u \in V_R$ and $v \in V$ if and only if v directly depends on u. The set E of edges is made up of edges between rule nodes (E_{RR}) and edges from rule to constraint nodes (E_{RC}).*

Example: Figure 9 shows the dependence graph for our working example. The edge from the PSS for literal `address` of rule `DR1` to the `works_in` literal of `IC2` denotes the direct dependence of `IC2` on constraint `DR1`. ◇

From the previous definition it can be seen that the dependence graph has a special structure: there are no edges outgoing from a node $v \in V_I$. There can be cycles among deductive rule nodes in the graph. This happens when R contains mutually recursive rules. There are no trivial cycles in the graph and it has the following property (Plexousakis, 1995):

Theorem 7.2 *For any Telos knowledge base, dependence graph construction yields a graph that may contain cycles of length at most equal to the number of deductive rules participating in the same recursive scheme.*

Furthermore, the graph is sparse for an average number α of literals per rule body or constraint greater than 2. The graph's sparsity will be exploited for the efficient maintenance of the graph's transitive closure. The dependence graph is constructed once when the knowledge base is compiled and is updated incrementally when new rules or constraints are inserted or deleted. The transitive closure of the graph is computed by a modification of the δ-*wavefront* algorithm (Qaddah, Henschen and Kim, 1991). The algorithm has been modified to apply to cyclic graphs and take advantage of the dependence graph properties. Evaluating the dependence graph's transitive closure amounts to computing the *potential implicit* updates caused by explicit updates on literals occurring in the bodies of deductive rules. The *actual implicit* updates are obtained during the evaluation phase. The time complexity for computing implicit updates caused by an explicit update matching some node in the graph is $O(|E|)$, and $O(|V_R| * |E|)$ for computing the transitive closure of the entire graph by solving $|V_R|$ single-source problems (Plexousakis, 1995). Experiments with randomly generated dependence graphs have shown that, on the average, the execution time of computing single-source implicit updates is sub-linear in $|E|$ (Plexousakis, 1993a).

7.2.3 Evaluation Phase

In this section we describe the evaluation phase of our algorithm. We first discuss how the dependence graph generated in the compilation phase is used to check the integrity constraints at the time of update. Then we describe how this graph is incrementally maintained as the set of integrity constraints and deductive rules changes.

Using Dependence Graphs for Constraint Checking

The dependence graph reflects both the logical and temporal interdependence of rules and constraints. To check if an update U affects an integrity constraint, we first locate all literals L_i in the dependence graph that unify with the update. The set of integrity constraints that may be violated are those which have at least one node on a path initiating at a literal L_i. As mentioned earlier, the dependence graph transitive closure can be precomputed at compile time. Hence, at update time reachability information does not need to be recomputed. It suffices to verify that the potential implicit updates are actual updates by instantiating the literals of the implicit updates and evaluating the bodies of rules in which they belong. For example, in Figure 9 an update on literal `univ` might cause a violation of constraint `IC2` since one of `IC2`'s literal's lies on a path with source from `univ`.

Incremental Maintenance of Dependence Graphs

As rules and constraints are inserted or deleted, we incrementally modify the dependence graph instead of reconstructing it from scratch. The dependence relationships between rules and constraints also change and have to be reflected in the graph's transitive closure that was computed during the compilation phase. In the rest of this section, we briefly describe how insertions and deletions of rules and constraints are handled. A more detailed description of the algorithms for incremental transitive closure maintenance and their complexity can be found elsewhere (Plexousakis, 1993a; Plexousakis, 1995).

Insertion of an integrity constraint is accepted only if the constraint is satisfied by the knowledge base. Then it is transformed into a set of parameterized forms, one for each of its literals. These forms are added as nodes to the dependence graph and in case that there exist rules already in the graph on which the constraint directly depends, edges from the rule nodes to the constraint nodes are added. The worst-case complexity of the dependence graph modification in case of a constraint insertion is $O(|V_R|)$, since the newly introduced nodes have to be connected with as many rule nodes as the number of rules whose conclusion literal matches the constraint literal. On the average, as experimental results with randomly generated graphs suggest, the cost is much smaller, since only a subset of the deductive rules will match the constraint literals. The deletion of a constraint cannot violate the integrity of the knowledge base. It suffices to remove all nodes corresponding to some simplified form of the constraint along with their incident edges. The worst-case complexity of the deletion process is $O(|E|)$.

When a new deductive rule is inserted, its direct dependence to existing rules or constraints is determined and represented in the graph. If there exist PSSs of constraints or rules with literals unifying with the rule's conclusion literal then the conclusions of the rule and any implicit updates must be derived and checked for possible constraint violations. If no violation of constraints arises then the rule is compiled and inserted in the dependence graph. If a literal of a rule/constraint unifies with the rule's conclusion then appropriate edges are added as described in the previous section. This process has a worst-case complexity of $O(|V_R| * |E|)$. Similarly, if an already compiled rule is to be deleted and there exist rules or constraints with literals matching the rule's negated conclusion, then the literals deducible with this rule are treated as normal deletions. If they do not cause integrity violation, the parameterized forms of the rule must be deleted along with all their incident edges. Rule deletion requires worst-case time of $O(|V_R| * |E|)$. An analytical model giving more precise characterizations of the cost of updates of rules and constraints can be found elsewhere (Plexousakis, 1995).

In addition to updating the dependence graph, we also need to incrementally compute its transitive closure. Incremental transitive closure algorithms available in literature can deal only with directed acyclic graphs (Ibaraki and Katoh, 1983; Italiano, 1988). In our research we have developed an algorithm that incrementally computes transitive closure for general graphs (Plexousakis, 1995). Our preliminary experiments with randomly generated sparse cyclic graphs have shown that this algorithm can efficiently update the transitive closure of a dependence graph. In the experiments carried out, the average cost for on-line transitive closure maintenance of sparse dependence graph was as low as $0.1 * |E|$ for the case of randomly generated sparse cyclic graphs.

8 Concluding Remarks

The proposed architecture for a knowledge base management system addresses several performance issues in a novel way:

- The physical data structures on which the knowledge base is stored are derived through the controlled decomposition method and include a novel temporal indexing technique;
- The query optimization includes semantic optimization in the presence of temporal knowledge as well physical optimization based on the cost models for our new storage and indexing schemes;
- Our concurrency control algorithms are extensions of existing algorithms for DAG databases, intended to take full advantage of the rich structure of a knowledge base; moreover, we have proven the correctness of our concurrency control policy, and have established both implementation and performance results;

- Our assertion compilation methods combine and extend previous results on compiling and simplifying static and dynamic temporal assertions. Soundness and completeness of the simplification method have been proven and preliminary performance results have been established.

Clearly, the design and performance analysis of the proposed architecture is not complete. In particular, work is in progress on the physical design of the KBMS (Shrufi, 1994), exploring the use of existing database storage kernels. A thorough experimental performance analysis is planned to validate the cost function of our storage and query model. The study of semantic criteria for reducing the search space when an access is planned is an issue that requires further research. The use of machine learning techniques in order to train the query optimizer with past experience is one possible direction for further exploration. Record clustering and buffer management are other directions of research that could lead to performance improvements.

We are generalizing our concurrency control algorithm so that it can distinguish between different semantic relationships, and to include multiple granularities of locking. Further down the road, we expect that the issues of fault tolerance such as recovery will become more important.

As far as rule management is concerned, a hybrid theorem prover for simplified constraints needs to be devised, possibly by combining existing special-purpose reasoners. Moreover, issues such as the efficient storage and access of the dependence graph and storage and indexing of rules and constraints are currently under investigation. The performance of the compilation method needs to be assessed and compared to methods that interleave compilation and evaluation, e.g. (Kuchenhoff, 1991). A dual approach to constraint enforcement, based on compiling constraints into transaction specifications, is a topic of current research (Plexousakis and Mylopoulos, 1995). Finally, a more fine grained approach to integrity violation needs to be devised, possibly adopting ideas of finite constraint satisfiability (Bry, Decker and Manthey, 1988).

In addition, we are working towards benchmarks for knowledge based systems, so that we can have a standard method to evaluate the algorithms developed for such systems.

On the basis of these results, we believe that a KBMS technology which offers the representational and inferential mechanisms of state-of-the-art knowledge representation schemes, while at the same time addresses efficiently database issues such as storage management, query processing and concurrency control is viable.

Acknowledgments

Our work has significantly benefited from the contributions of several people. Special thanks are due to our colleagues Lawrence Chung, Prof. Vassos Hadzilacos, Igor Jurisica, Manolis Koubarakis (now at UMIST) Bryan Kramer, David Lauzon, Brian Nixon, Thomas Rose (now at FAW, Ulm), Prof. Ken Sevcik and Huaiqing Wang; also to visitors Prof. A. Illarramendi (Universidad del Pais Vasco, Spain), Prof. Yannis Ioannidis (University of Wisconsin), Prof. Matthias Jarke (Technical University of Aachen), L. Sbattella (Politechnico di Milano, Italy) and Prof. Yannis Vassiliou (Technical University of Athens).

References

Agrawal, R., Carey, M. J., and Livny, M. (1987). Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1987). *Data Structures and Algorithms*. Addison-Wesley Publishing Company.

Allen, J. (1983). Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843.

Attardi, G. and Simi, M. (1981). Semantics of Inheritance and Attributions in the Description System OMEGA. Technical Report S-81-16, Universita Di Pisa. Also MIT AI memo 642, 1981.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Welssley Publishing Company.

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

Biliris, A. (1992). The Performance of Three Database Storage Structures for Managing Large Objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Bocca, J. (1986). On Evaluation Strategy of EDUCE. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 368–378.

Brodie, M., Mylopoulos, J., and Schmidt, J., editors (1984). *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. Springer Verlag.

Bry, F., Decker, H., and Manthey, R. (1988). A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *1st Int. Conference on Extending Data Base Technology*, pages 488–505, Venice, Italy.

Buchanan, B. G. and Wilkins, D. C., editors (1993). *Readings in Knowledge Acquisition and Learning*. Morgan Kaufmann Publishers, Inc., San Mateo, CA.

Carey, M., DeWitt, D., Richardson, J., and Shekita, E. (1986). Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 91–100.

Carroll, M. D. (1988). Data Flow Analysis via Dominator and Attribute Updates. Technical Report LCSR-TR-111, Rutgers University.

Ceri, S. and Widom, J. (1990). Deriving Production Rules for Constraint Maintenance. In *Proceedings of the 16th Int. Conference in Very Large Databases*, pages 566–577.

Chakravarthy, V., Grant, J., and Minker, J. (1988). Foundations of Semantic Query Optimization for Deductive Databases. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan-Kaufmann.

Chaudhri, V. K. (1995). *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation*. PhD thesis, University of Toronto, Toronto.

Chaudhri, V. K. and Hadzilacos, V. (1995). Safe Locking Policies for Dynamic Databases. In *Fourteenth ACM Symposium on Principles of Database Systems*.

Chaudhri, V. K., Hadzilacos, V., and Mylopoulos, J. (1992). Concurrency Control for Knowledge Bases. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 762–773.

Chaudhri, V. K., Hadzilacos, V., Mylopoulos, J., and Sevcik, K. (1994). Quantitative Evaluation of a Transaction Facility for a Knowledge Base Management System. In *Proceedings of the Third International Conference on Knowledge Management*, pages 122–131, Gaithersberg, MD.

Chaudhri, V. K. and Mylopoulos, J. (1995). Efficient Algorithms and Performance Results for Multi-User Knowledge Bases. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, page To Appear, Montreal.

Chomicki, J. (1992). History-less Checking of Dynamic Integrity Constraints. In *8th Int. Conference on Data Engineering*, pages 557–564, Phoenix,AZ.

Copeland, G. and Khoshafian, S. (1985). A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279.

Dechter, R., Meiri, I., and Pearl, J. (1989). Temporal Constraint Networks. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning*, pages 83–93.

Decker, H. (1986). Integrity Enforcement in Deductive Databases. In *Expert Database Systems, 1st Int. Conference* , pages 271–285.

Eswaran, K., Gray, J. N., Lorie, R. A., and Traiger, I. L. (1976). The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(9):624–633.

Findler, N., editor (1979). *Associative Networks*. Academic Press.

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

Frank, M., Omiecinski, E., and Navathe, S. (1992). Adaptive and Automated Index Selection in RDBMS. In *Proceedings of International Conference on Extending Database Technology*, pages 277–292.

Frenkel, K. A. (1991). The Human Genome Project and Informatics. *Communications of the ACM*, 34(11):41–51.

Gupta, A., Sagiv, Y., Ullman, J., and Widom, J. (1994). Constraint Checking with Partial Information. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* , pages 45–55.

Guttman, A. (1984). R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57.

Hull, R. and King, R. (1987). Semantic Database Modeling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3).

Hulsmann, K. and Saake, G. (1990). Representation of the Historical Information Necessary for Temporal Integrity Monitoring. In *2nd Int. Conference on Extending Data Base Technology*, pages 378–392, Venice, Italy.

Ibaraki, T. and Katoh, N. (1983). On-Line Computation of Transitive Closures of Graphs. *Information Processing Letters*, 16(3):95–97.

Ioannidis, Y., Ramakrishnan, R., and Winger, L. (1993). Transitivie Closure Algorithms based on Graph Traversal. *ACM Transactions on Database Systems*, 18(3):512–576.

Ioannidis, Y. E. and Kang, Y. C. (1991). Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 168–177.

Ishikawa, H., Suzuki, F., Kozakura, F., Makinouchi, A., Miyagishima, M., Izumida, Y., Aoshima, M., and Yamane, Y. (1993). The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System . *ACM Transactions on Database Systems*, 18(1):1–50.

Italiano, G. (1988). Finding Paths and Deleting Edges in Directed Acyclic Graphs. *Information Processing Letters*, 28(1):5–11.

Jarke, M. and Koch, J. (1984). Query Optimization in Database Systems. *Computing Surveys*, 16(2):111–143.

Jarke, M. and Koubarakis, M. (1989). Query Optimization in KBMS: Overview, Research Issues, and Concepts for a Telos Implementation. Technical Report KRR-TR-89-6, Dept. of Computer Science, University of Toronto.

Jeusfeld, M. and Jarke, M. (1991). From Relational to Object-Oriented Integrity Simplification. In *2nd Int. Conference on Deductive and Object-Oriented Databases*, pages 460–477, Munich, Germany.

Khoshafian, S. and Copeland, G. (1986). Object identity. In *Proceedings of OOPSLA-86*, pages 406–416, Portland, Oregon.

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

Kuchenhoff, V. (1991). On the Efficient Computation of the Difference Between Consecutive Database States. In *2nd International Conference on Deductive and Object-Oriented Databases*, pages 478–502, Munich, Germany.

Law, A. M. and Kelton, W. D. (1991). *Simulation Modeling and Analysis*. McGraw-Hill New York, NY.

Lengauer, T. and Tarjan, R. E. (1979). A Fast Dominator Algorithm for Finding Dominators in a Flow Graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141.

Ling, T. and Lee, S. (1992). Integrity Checking for Transactions in Relational Databases. In *International Computer Science Conference*, pages 245–251.

Lipeck, U. (1990). Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science*, 76:115–142.

Livny, M. (1986). DeNeT User's Guide (Version 1.5). Technical report, University of Wisconsin.

Lockemann, P. C., Nagel, H.-H., and Walter, I. M. (1991). Databases for Knowledge Bases: empirical study of a knowledge base management system for a semantic network. *Data & Knowledge Engineering*, 7:115–154.

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

Mylopoulos, J., Kramer, B., Wang, H., Benjamin, M., Chou, Q. B., and Mensah, S. (1992). Expert System Applications in Process Control. In *Proceedings of the International Symposium on Artificial Intelligence in Materials Processing Applications*, Edmonton.

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

Nicolas, J.-M. (1982). Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, 18:227–253.

Paul, H.-B., Schek, H.-J., Scholl, M., Weikum, G., and Deppish, U. (1987). Architecture and Implementation of a Darmstadt Database Kernel System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 196–207, San Francisco, CA.

Plexousakis, D. (1993a). Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 146–157, Dublin, Ireland. Morgan Kaufmann.

Plexousakis, D. (1993b). Semantical and Ontological Consideration in Telos: a Language for Knowledge Representation. *Computational Intelligence*, 9(1):41–72.

Plexousakis, D. (1995). *On the Efficient Enforcement of Integrity Constraints in Temporal Deductive Knowledge Bases*. PhD thesis, Department of Computer Science, University of Toronto. Forthcoming.

Plexousakis, D. and Mylopoulos, J. (1995). Accommodating Integrity Constraints During Database Design. Submitted for Publication.

Qaddah, G., Henschen, L., and Kim, J. (1991). Efficient Algorithms for the Instantiated Transitive Closure Queries. *IEEE Transactions on Software Engineering*, 17(3):296–309.

Schieber, B. and Vishkin, U. (1988). On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM Journal of Computing*, 17(6):1253–1262.

Selinger, G., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. (1979). Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34.

Shrufi, A. (1994). Performance of Clustering Policies in Object Bases. In *Proceedings of the Third Conference on Information and Knowledge Management*, pages 80–87.

Silberschatz, A. and Kedem, Z. M. (1980). Consistency in Hierarchical Database Systems. *Journal of the Association for Computing Machinery*, 27(1):72–80.

Snodgrass, R. (1987). The Temporal Query Language TQuel. *ACM Transcactions on Database Systems*, 12(2):247–298.

Stanley, M. (1986). CML: A Knowledge Representation Language with Application to Requirements Modeling. Technical report, University of Toronto, Toronto.

Steinbrunn, M., Moerkotte, G., and Kemper, A. (1993). Optimizing Join Orders. Technical Report MIP-9307, Universität Passsu, Fakulät für Mathematik and Informatik.

Stickel, M. (1985). Automated Deduction by Theory Resolution. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 455–458, Los Angeles, CA.

Stonebraker, M. (1975). Implementation of Integrity Constraints and Views by Query Modification. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 65–78.

Stonebraker, M. and Dozier, J. (1991). An Overview of the SEQUOIA 2000 Project. Technical Report SEQUOIA-TR-91/5, University of California, Berkeley.

Tay, Y. C. (1987). *Locking Performance in Centralized Databases*. Academic Press, London.

Topaloglou, T. (1993). Storage Management for Knowledge Bases. In *Proceedings of Second International Conference on Information and Knowledge Management (CIKM'93)*.

Topaloglou, T., Illarramendi, A., and Sbattella, L. (1992). Query Optimization for KBMSs: Temporal, Syntactic and Semantic Transformation. In *Proceedings of the International Conference on Data Engineering*, pages 310–319.

Ullman, J. (1988). *Principles of Data Base and Knowledge Base Systems*, volume 1. Addison Wesley.

Valduriez, P. (1987). Join Indices. *ACM Transaction on Database Systems*, 12(2):218–246.

Valduriez, P., Khoshafian, S., and Copeland, G. (1986). Implementation Techniques of Complex Objects. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 101–109, Kyoto, Japan.

Vilain, M., Kautz, H., and van Beek, P. (1989). Constraint Propagation Algorithms for Temporal Reasoning: a Revised Report. In Weld, D. and de Kleer, J., editors, *Readings in Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kaufmann.

Wallace, M. (1991). Compiling Integrity Checking into Update Procedures. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 903–908.

Yannakakis, M. (1982). A Theory of Safe Locking Policies in Database Systems. *Journal of the Association for Computing Machinery*, 29(3):718–740.

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

Zdonik, S. B. and Maier, D., editors (1989). *Readings in Object-Oriented Databases.* Morgan Kaufmann Publishers, Inc., San Mateo, CA.