Query Processing for Knowledge Bases Using Join Indices

Adel Shrufi and Thodoros Topaloglou
{shrufi,thodoros}@cs.toronto.edu
Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A4

Abstract

This paper addresses the problem of physical query processing for large object-oriented, temporal knowledge bases. The major tasks being investigated are how to generate the space of all possible execution plans for a given knowledge base query and how to traverse this space in order to choose an efficient execution plan. The results of this work include: (a) the formulation of a set of access level operations which depend on the underlying storage model and the development of a cost model for estimating their cost; (b) the exploration of various optimization heuristics for selecting efficient execution plans for temporal path queries which make use of the join index relations that are provided by the storage model; and (c) a performance study that shows the benefits of join index based query processing techniques for knowledge bases compared to the traditional tuple-oriented (characteristic of the AI-DB coupling systems) and bulk join query processing approaches in database systems.

1 Introduction

A critical factor to the success of knowledge base system technology, that deals with large-scale knowledge bases such as those referred in [LG89], [NFF+91], will be its ability to provide robust performance comparable to that of database systems. This is particularly crucial for large disk-resident knowledge bases where the goal is to provide access efficiency by minimizing the number of disk I/Os incurred during query execution. This paper addresses the problem of query processing and optimization for knowledge bases, a function of the knowledge base management system (KBMS) whose goal is to enhance performance by executing queries efficiently. This work is part of a more general approach towards the development of knowledge base management systems which we report in [MCP+95].

In general, query processing is considered a highly complex process requiring two primary phases which are applied in tandem. These are the syntactic and semantic optimization phase and the physical access planning phase which are handled by a semantic query optimizer and an access optimizer component, respectively. In a nutshell, the submitted query is specified in a non-procedural language describing the conditions that the answer set must satisfy. The query is first transformed, by the semantic optimizer, through syntactic and semantic simplifications to a more efficient representation. Then, the access optimizer transforms the query into an executable form, taking into account physical data structures, available access operations and quantitative information about the knowledge base content. It should also be noted that the optimal query execution plan (i.e. the one with the least cost) is generally difficult to attain due to the large size of search space, hence query optimizers strive to achieve near-optimal plans in the least amount of time possible.

Query processing for knowledge bases is even more complex for several reasons. First, knowledge bases usually offer highly expressive query languages consisting of temporal, spatial, deductive and object-oriented operators. Second, the rich knowledge base storage and index structures enlarge the access plan space thereby complicating the process of access planning. Furthermore, these structures, along with the novel indexing techniques for knowledge bases, require more sophisticated cost model formulas than the ones encountered in traditional systems.

In this paper we address the access planning phase for Telos knowledge bases [MBJK90] whose implementation is based on the controlled decomposition storage model [Top93]. Our earlier work on semantic query optimization in Telos is reported in [TIS92]. The query processing tasks that we concentrate on are how to generate the space of all possible execution plans and how to traverse this space in order to choose an efficient execution plan. The input of access planning consists of quantitative information about the knowledge base (e.g. number of instances per class, IsA hierarchies, etc.), information about its physical implementation and the input query which may have already been passed through the semantic query optimizer. During access planning, the obtained query is first transformed into a query graph which is successively transformed into a set of parameterized operator trees, and finally a set of execution trees. Each execution tree represents a possible access plan. Once we have the execution trees, we select the one with the least cost using the access optimizer.

The contributions of this paper are summarized as follows. First, we identify a set of access level operations for executing simple queries, and we show how execution plans for more complex queries are constructed out of them. Second, we develop the necessary cost formulae for estimating the disk I/O costs and size estimates of the defined access

To appear in the Fourth International Conference on Information and Knowledge Managemant (CIKM'95), Baltimore, MD, November 1995

operations and the intermediate processing results. Third, we tackle the problem of optimizing the access planning for queries with arbitrary number of path expressions. Finally, we show by conducting an experimental study, the benefits that join index based query processing techniques offer for knowledge bases implemented within a KBMS compared to the traditional tuple-oriented (AI) and bulk join based (database-like) query processing approaches.

The rest of this paper is organized as follows. Section 2 briefly outlines our data model and query language. Section 3 presents the storage model that underlies the Telos KBMS. Section 4 discusses the developed access language, cost model and employed optimizations. Section 5 provides a quantitative evaluation, and section 6 concludes with a summary and directions for future work.

2 Querying Knowledge Bases

2.1 The Telos Knowledge Model

Telos [MBJK90] offers a representational framework which includes structuring mechanisms analogous to those offered by semantic networks and semantic data models. It also provides an assertional sublanguage for the expression of deductive rules and integrity constraints. Two novel aspects of Telos are the treatment of attributes as first-class citizens and the provision of special representational and inferential capabilities for temporal knowledge. Structural and assertional knowledge is organized along *instantiation, generalization* and *aggregation* dimensions. Telos emphasizes the use of time for representing possible histories of generic events and activities, as well as the system's knowledge of these histories, by providing two temporal dimensions for *historical* and *belief* time.

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

In the following discussion, we assume that time is recorded using an attribute versioning scheme thus making attribute values represented by triples, $\langle v, t_h, t_b \rangle$, where v is the value that some attribute holds at historical time t_h according to the system's belief at time t_b. Both t_h and t_b are intervals consisting of a starting and an ending time point which are conventional dates. Time intervals are related in temporal assertions by using the Allen's [All83] thirteen interval relations. The temporal dimensions for instanceOf and IsA relationships are handled in a similar fashion.

Figure 1 illustrates a variety of attribute types that are relevant to our discussion. `budget` and `sal` are examples of *simple* attributes, i.e., attributes whose domain is a primitive class (e.g., `Real`, `Integer`, ..). `dept` and `comp` are examples of *complex* attributes, i.e., attributes whose domain ranges over the instances of a class. Paths of complex attribute links, e.g., `dept.comp`, define nested attributes that are also called *path* attributes. The query processing techniques that we develop in this work emphasize on queries containing temporal path attributes in an object-based setting such as the one adopted by Telos.

2.2 Queries

In Telos, a knowledge base is queried with respect to a specified history and belief time interval. Queries are specified through the `ASK` operation [MBJK90] which has a general syntax shown as follows:

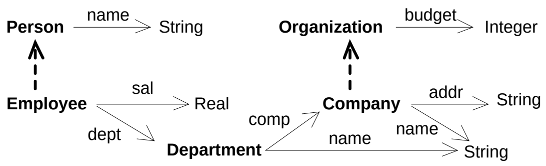

Figure 1: The `employee/departments` knowledge base

```
ASKx₁/S₁, ..., xₙ/Sₙ  : W
   ON t₁
   AS OF t₂
```

$x_1, ..., x_n$ are assumed to be target variables for which the query processor needs to provide values; $S_1, ..., S_n$ 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 all knowledge base propositions whose history time covered by or overlaps with t_1 according to what was believed by the knowledge base over the interval t_2.*

For example, the query shown below retrieves all employees who worked for the database group of IBM over the period starting at the beginning of 1990 and ending at the end of 1991 according to the system beliefs for the same time period. A variation of this query is to ask the same expression according to what the system currently believes just by omitting the `AS OF` part.

```
ASK e/Employee: e.dept.comp.name = "IBM" and
                e.dept.name = "dbgroup"
   ON 1990 ..  1991
   AS OF 1990 ..  1991
```

According to [Sno87], a query with both of the above time qualifications is termed a temporal, while one with only the first temporal qualification is called a historical query. In the following study, we focus our discussion on historical queries.

2.3 Query Graphs

The queries that we consider in this paper have the following general form:

`ASK` $x_1/C_1 \dots x_k/C_k$: $F_1 \wedge F_2 \wedge \dots \wedge F_n$ `ON t`

Each F_i is a path expression $x_i.A_1.A_2 \dots A_k.a_i$ *op* v_i where x_i is a variable ranging over a target class C_i, each A_j denotes a complex attribute form 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 example query above are `e.dept.comp.name = "IBM"` and `e.dept.name = "dbgroup"`.

Let $\mathbf{C}$ be the set of all distinct classes for which attribute links occur in a query expression `W`. A query graph then is defined as the undirected graph whose set of nodes is $\mathbf{C}$ and its set of edges consist 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 simple attribute restriction of the form a_i *op* v_i appears in `W` are annotated by a label $(a_i, op, v_i, \texttt{t})$, where `t` is the historical time specified in the `ON` field of the query or `alltime` otherwise. For example, the query graph for the example query is shown in Figure 3(a).

3 Storage Model

In order to put the discussion of query processing in perspective, it is important to shed some light on the underlying storage subsystem which is briefly described below.

Telos knowledge bases are stored using the Controlled Decomposition storage Model (CDM) [Top93]. CDM is a flexible combination of the n-ary (direct) storage model (NSM) and the decomposition storage model (DSM) [CK85], [VKC86]. NSM utilizes a single relation for each object class where DSM stores an object oriented schema by generating a binary relation for each attribute of a class. The most noticeable disadvantage that the above models have are the NSM's inflexibility to handle IsA relationships and schema updates and DSM's high storage cost. In CDM we have minimized these inefficiencies and we have also integrated historical time with the storage structures. In the next section, we outline the features of CDM emphasizing on those that are relevant to the development of this paper. A detailed description of CDM can be found in [Top93].

3.1 The Controlled Decomposition Model

The Controlled Decomposition algorithm operates on a description of the knowledge base structure and generates a logical schema and the associated index methods for it. The generation of a logical design using the Controlled Decomposition involves: a) transforming the IsA hierarchies from a general graph form to a forest (a set of trees), b) decomposing the trees into a set of relations, and c) assigning hierarchical identifiers to the class instances and storing them as tuples in the relations.

Let $G = (V, E)$ be the graph representing the `IsA` hierarchy associated with a knowledge base, with V is the set of classes and E is the set of `IsA` relationships, declared or derived, between these classes. Let also $f(C)$ be the number denoting the frequency of access of class C, for every class $C \in V$. Then the notion of *most preferred parent* for a class C which is a subclass of classes $C_1, C_2, \ldots, C_k$, is some class C_i such that $f(C_i) \geq f(C_j), 1 \leq j \leq k, i \neq j$. The steps below outline the transformation of an `IsA` hierarchy into a forest and then the mapping of the tree-structured class hierarchy into relations.

CD1 For every class $v \in V$ with more than one parent, choose one of the most preferred parents. Discard from the graph all edges connecting v to its parents except for that connecting v to its chosen most preferred parent. The attributes list of class v is extended with the attributes inherited form the disconnected parents.

CD2 The root class of each tree becomes a relation nucleus with fields that are determined by rule CD5.

CD3 Descending each tree, create relations for subclasses in order to store local attributes (i.e., ones not inherited) for each class.

CD4 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 [Top93]).

CD5 The relations corresponding to a class C are as follows:

`PERSON`

P_ID	ENAME

`EMPLOYEE`

E_ID	SAL

`DEPARTMENT`

D_ID	DNAME

`EMP_DEPT`

E_ID	D_ID	TIME

`DEPT_COMP`

D_ID	C_ID	TIME

`ORGANIZATION`

O_ID	BUDGET

`COMPANY`

C_ID	CNAME	CADDR

Figure 2: Relations created by the CDM

a Each attribute A_i of C that has a complex domain (i.e., its domain is another class, say C_i), is stored in a separate relation, called a *join index relation* and denoted by R_{C-C_i}. In fact, CDM stores also the history time of the joining pairs, thus forming a *temporal join index relation*. If $oid(C)$ denotes an object identifier corresponding to an instance of class C, then the temporal join relation is defined as:

$$R_{C-C_i} = \{\langle oid(C), oid(C_i), t\rangle | oid(C).A_i = oid(C_i) \text{ at } t\}$$

b For the attributes $B_1, \ldots. B_k$ of a class C that have simple domains (e.g., `Integer, String`, etc.), a single relation R_C is created, R_C has the following form:

$$R_C(OID, B1, \ldots, B_k) = \{\langle oid(C), v_{1_i}, \ldots v_{k_i}\rangle | \\ oid(C).B_j = v_{j_i}, 1 \leq j \leq k\}$$

CD6 If a simple attribute B in C, is a temporal attribute, then the B field values in an R_C relation are histories of the form $\{\langle v_1, t_1\rangle, \ldots, \langle v_l, t_l\rangle\}$. In this presentation of CDM, we support only historical time.

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. Also notice that, rules CD3, CD4 and CD5a are adaptations from the DSM, whereas rules CD2 and CD5b are adopted from the NSM. Rule CD6 accounts for temporal knowledge.

The storage relations which are generated by the CDM algorithm for a portion of the knowledge base of Figure 1 are shown in Figure 2. The `EMPLOYEE` relation stores only the local attribute of the `Employee` class (e.g., `sal`) while 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.

3.2 Access Methods

Time plays a key role in knowledge base queries and therefore its involvement in access methods definition is imperative. The existence of complex attributes, on the other hand, makes the join index relations [Val87] to be highly promising access methods for knowledge bases. These two observations were the key criteria in designing the access methods supported by the CDM storage manager.

The access methods that are considered by our query processor are: the *restriction temporal index* (used for resolving restriction predicates on simple attributes, hence the name) and the *temporal join index* (defined on non-atomic attributes). Both of them are generalizations of non-temporal indices where the *key* now becomes a pair $\langle value,\ time\rangle$ specifying also the historical time which the qualified tuples must satisfy. With these indices it is possible to perform selection using keys and time simultaneously. In addition, these indices are hierarchical in the sense that one index stores information for all the instances of a class (including inherited ones). Below we simply present their definitions, the interested reader can find more details about their functionality and performance in [Top93].

The restriction temporal index is used on attributes with primitive domains. Its search key is a pair of a domain value and a time segment i.e., $\langle$*value, time*$\rangle$ and the returned value is a list of qualified token identifiers. A restriction temporal index is defined for most frequently accessed attributes.

The temporal join index is used to describe a time-dependent relationship between the instances of two classes. In a temporal join index corresponding to the classes C_1 and C_2 each entry is a 3-tuple $\langle oid(C_1), oid(C_2), t\rangle$. For example, the relations $R_{\texttt{DEPT\_COMP}}$ and $R_{\texttt{EMP\_DEPT}}$ on Figure 2 represent such relationships. The searching key in a temporal join index is a pair of an object identifier and a temporal interval. For the key $\langle$*OID1, time*$\rangle$, the object identifiers of instances of the second class which are in a temporal join index tuple containing *OID1* for a time interval overlapping with *time* are returned. Moreover, temporal join index is a bi-directional index so that can be accessed by the object identifier of the either of the classes. To make this two-way access efficient we assume that that two copies of each temporal join index are stored, one clustered on OID1 (forward), $R_{C_1 \rightarrow C_2}$, and the other clustered on OID2 (inverted), $R_{C_1 \leftarrow C_2}$ (note that this distinction is indicated by the direction of the arrow).

4 Processing Queries

One of the characteristics of query processing is that it is highly dependent on the underlying storage model. Specifically, the success of a query optimizer in discovering an efficient execution strategy depends critically on how well it can utilize the available access structures. To this end, statistics about the knowledge base need to be available, as well as a cost model that predicts the cost of using each access method. Note that, in our case, due to the rich structure of the underlying knowledge representation scheme and the presence of hierarchies, estimate indicators that are used in the relational model such as selectivity of predicates, cardinality of domains and join selectivities need to be revised. Once again, time comes to play a role as well. For instance, the selectivity of a selection operation such as `employee.salary=50K ON 1989..1991` will be the selectivity of the operation as in [SAC+79] multiplied by a temporal selectivity factor. Thus, instead of simply referring to selectivity of predicates, we speak of temporal selectivity of predicates. This also implies that the cost model formulas must be revised to take this into account, as we show later on.

A solution for the physical query processing operation has to consider the the following tasks:

1. Maintaining statistical information about the knowledge base, including both tokens and classes, that can be used for evaluating the cost of an access plan.

2. Identifying the search space of all potentially useful access plans.

3. Estimating the cost of each access plan.

4. Selecting an efficient access plan among all the alternatives.

In the following sections, we address each of the above issues, in detail, explore the applicability of existing approaches, and finally, we propose our own solution.

4.1 The Cost Model

The cost model developed for the KBMS query processing consists of two primary components. The first (this section) pertains to the statistical profile of the knowledge base and the necessary formulas for estimating the size of intermediate results during query evaluation. Such formulas include those that estimate the domain cardinality of set-valued attributes, object-join selectivity (defined here as the degree of reference sharing) and temporal selectivity of predicates. The second component (next section) of the cost model pertains to the cost of performing an access operation. This is useful in computing the overall cost of an access plan, and hence in choosing one with the least cost. The primary cost considered here is disk I/O which is the bottleneck in large database and knowledge base systems.

Data Parameters. The following list of parameters is used in the formulas that define the cost model:

$\lvert C\rvert$	cardinality of a class C
$V_{C,a}$	number of unique class C instances in the domain of attribute a
H	length of the history of the knowledge base
T_q	history time at which the query was posed
g	average length (lifetime) of a temporal value
σ	selectivity of an operation
r_a	average number of historical time values associated with each occurrence of attribute a
m_{a_C}	multivaluedness factor for attribute a of class C.

Note that m_{a_C} counts multiplicity of attribute values over time. Given this parameter, it is possible to compute the domain cardinality of the set-valued attribute a (denoted by $|a|$) ranging over instances of class C as follows:

$$|a| = \binom{V_{C,a}}{m_{a_C}}$$

Effectively, the above formula computes the number of unique sets of size m that can be selected from $V_{C,a}$ unique values. Using the above parameters we can compute the degree of reference sharing and selectivity as shown below.

Degree of Reference Sharing. If a is an attribute of a class C_1 such that the domain of the attribute a is the class C_2, then the degree of reference sharing between classes C_1 and C_2, denoted by $DRS_{C_2 \rightarrow C_1}$ is defined as the average number of instances of class C_1 that reference a single class C_2 instance. The degree of reference sharing can be computed as:

$$DRS_{C_2 \rightarrow C_1} = \frac{\binom{V_{C_2,a} - 1}{m_{a_{C1}} - 1}}{\binom{V_{C_2,a}}{m_{a_{C1}}}} |C_1|$$

Note that the numerator term in the above formula gives the number of different sets (of size m) that a unique instance of C_2 participates in, thereby enabling us to compute its uniform probability of being selected.

Temporal Selectivity. The selectivity σ of a non-temporal qualification on a temporal data domain is estimated as in [SAC+79]. For a temporal qualification, the selectivity σ_T is obtained by multiplying σ by a *temporal selectivity* factor τ. The value of τ depends on the distribution of the temporal domain and the query time (T_q). [Top93] analyzes the temporal selectivity of three different types of temporal domains: (1) point values without duration, (2) uniform distribution of intervals of the same size g, and (3) right-ended intervals. The value of τ equals to $\frac{T_q}{H}$, $\frac{g+T_q}{1+g}(V*r+1)$, and $\frac{T_2}{H}$ respectively for the aforementioned temporal domains.

Index Parameters. The following parameters are used to characterize the performance of indices. The R and B subscripts of these parameters are used to distinguish R-trees and B$^+$-trees.

F	average number of entries per index node
E	average number of entries per leaf node
h	height of the index tree
u	average node occupancy
B	effective storage capacity of a page
$size(x)$	size of an entry x in bytes

If N is the number of keys, then the height (h) of an index tree (B- or R-tree) is given by the equation: $h = 1 + \lceil \log_F(\lceil N/E \rceil) \rceil$.

Cost of Indices. Given the above cost model, the cost in page accesses of an index lookup on single or multiple key queries is equal to the height of the tree plus a fraction of its leaf pages, i.e.,

$$cost(\text{tree\_index(P)}) = h_{tree} + \sigma(P) \times LP_{tree}$$

where P is the qualification predicate, $\sigma(P)$ its selectivity and $LP = \lceil N/E \rceil$ the number of leaf nodes. If the index is temporal the subscript $_{tree}$ is substituted by R, otherwise by B to denote R-tree or B-tree index, respectively. Note that an R-tree type index is used for the implementation of the temporal join index, both in the forward and the backward direction. The restriction index is implemented an R-tree or B-tree depending on whether its indexing domain is a temporal or a regular one.

Cost of Data Fetching. Finally, in order to estimate the number of pages accessed as a result of fetching data objects, we utilize Yao's formula [Yao77]. This formula has the generic form of Yao(K, M, N) where K is the number of data items to be fetched which are randomly placed among N objects, and M is the total number of pages. Then, the number of pages accessed is estimated as follows:

$$\text{Yao}(K, M, N) = M \times \left(1 - \frac{C_K^{N-N/M}}{C_K^N}\right)$$

where $C_y^x = \frac{x!}{y(x-y)!}$.

4.2 Primitive Access Operations

There are six primitive access operators that are supported by CDM. These are: class scanning, index-based retrieval based on a temporal or not-temporal qualification condition, forward or inverted join index access, intersection of same class temporary answer sets (also referred as explicit joins), and finally object fetching.

In what follows, we briefly explain these operations and give their respective costs and the output size estimates. In order to reduce the complexity of the cost formulas, we had to adopt some generally-accepted assumptions [IK91]. Specifically, intermediate results are always materialized before they are fed as input to subsequent operations. Secondly, we assume minimal buffering of data or index pages between operations. These assumptions eliminate the complex problems that arise with the incorporation of buffering in the cost formulas. The six primitive operations are defined as follows: [1]

- *Scan.* A scan operation takes 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.

  cost: $S(R_C, P) = \frac{|C| * SIZE(TUPLE_{R_C})}{B}$

  output size: $L_C = \sigma(P) * \mid C \mid$

- *Index Retrieval.* An index retrieval takes as input a qualification predicate P on some attribute (possibly temporal) of a class C, and uses an index tree (B-tree for the non-temporal case, R-tree for the temporal case), denoted I_C, to return a list of OIDs for objects satisfying the predicate.

  cost: $RI(I_C, P) = h_{tree} + \sigma(P) * LP$

  output size: $L_C = \sigma(P) * |C|$

- *Forward Join Index.* A forward join index operation takes as input a list of object identifiers of class C_1 and uses the join index relation $R_{C_1 \to C_2}$ to return the OIDs of class C_2 that are joined through the attribute link. In the list of C_1 OIDs is not specified then the entire index relation is scanned.

  cost:
  $JI^F(R_{C1 \to C2}, L_{C1}) = \text{Yao}(\mid L_{C1} \mid, LP, \mid C1 \mid *r) + h_F$

  output size:
  $L_{C2} = Yao((L_{C1} * m_{C1 \to C2}), V_{C2}, (N_{C1} * m_{C1 \to C2}))$

- *Inverted Join Index.* This is the counterpart operation 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.

  cost:
  $JI^I(R_{C1 \leftarrow C2}, L_{C2}) = \text{Yao}(\mid L_{C2} \mid, LP, V_{C2} * r) + h_I$

  output size: $L_{C1} = L_{C2} * DRS_{C2 \to C1}$

- *Intersection.* An intersection operation takes as input two lists of object identifiers of the same class and returns the object identifiers that are common to both the lists.

  cost: $I(L1, L2) = \mid L1 \mid \times \mid L2 \mid$

  output size: $\mid L3 \mid = \frac{|L1| * |L2|}{N_c}$

- *Object Fetching.* This operation takes a physical relation R_C for class C, a list 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 satisfying ones. 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.

  cost: $F(R_C, L_C, P) = \text{Yao}(\mid L_C \mid, M_{R_C}, \mid C \mid)$

  output size: $Yao(|L_C|, M_C, |C|)$

[1] L_C denotes a list of OIDs of class C (if the subscript is a number then it discriminates lists of the same class)

4.3 Access Planning

The task of an access planner is to take a query graph and generate an efficient execution strategy. The major tasks involved are how to generate the space of all possible execution plans and how to traverse this space in order to choose an efficient execution plan. A query graph can be transformed into a set of parameterized operator trees each of which may have one or more execution trees associated with it. Each execution tree is expressed in terms of the primitive access operations, hence represents a possible access plan of executing the query. The set of all parameterized operator trees and their associated execution trees represent the whole strategy space. Moreover, since we already saw that each access operation has an associated cost formula, this enables us to compute the cost of an execution tree. The ultimate task thus becomes the selection of the execution tree that has the least cost. Unfortunately, it is often hard to come up with the efficient execution tree; especially when the strategy space is too large to search (as is the case for complex queries), and hence it is customary to settle for a near-optimal execution tree.

4.3.1 Parameterized Operator Tree

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 formalized as follows:

input: A query graph QG
output: A set of PO trees, $J(\ldots J(J(C_1, C_2), C_3), \ldots C_n)$, representing 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. C_i is connected by an edge in QG to one of the nodes in the subsequence $C_1 \ldots C_{i-1}$ (i.e. C_i has a parent or a child node in the subsequence).

J is the parameterized operator.

A PO tree is a left-deep tree representation of a possible way to traverse the query graph. More specifically, a PO tree is a binary tree in which every internal node has at least one leaf node from the node set of the query graph. A non-leaf node of the tree represents a join operator whose input edges are base classes and/or intermediate result and output edge is the the result obtained by applying the parameterized operation J to its children. As we will see in the sequel, this operation resolves into a specific join operation.

The rationale for focusing on left-deep trees, rather than the more generalized bushy trees, stems from three reasons: First, the search space in the latter becomes unreasonably large, second, it has been argued in the literature [SMK93] that the left-deep trees space contain the most feasible execution strategies, and third, in our case, left-deep trees utilize better the available storage structure. The number of all possible PO trees for a query graph with n nodes is $n(n-1)$. In fact, for our queries this is a worst case bound. Property P2 in conjunction with the connectivity of the query graph limits decreases the numbers of possible PO trees. As an example, in Figure 3, we show two possible PO trees for query Q2.

4.3.2 Execution Tree

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 is materialized in terms of a set of physical relations at the storage such that: a physical relation R_C stores the subset of its primitive attributes, and a physical join relation R_{C,C_i} materializes the join between classes C and C_i which is due to the complex attribute A_i of C. The final execution tree is expressed in terms of primitive operation being performed on these physical entities.

In the process of 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 labeled by one of the primitive operations (index retrieval or scan) and each non-leaf node is labeled by the join method (forward join index, inverted join index or intersection). The following two examples explain this process in detail.

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

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

$ET1$: $JI^I(R_{Emp \leftarrow Dept}, RI(I_{Dept}, P_2))$
$ET2$: $F(R_{Dept}, JI^F(R_{Emp \rightarrow Dept}, \_)), name = \texttt{dbgroup})$
$ET3$: $I(JI^F(R_{Emp \rightarrow Dept}, \_), RI(I_{Dept}, name = \texttt{dbgroup}))$

Multiple classes query. Without any restriction of the generality we can assume the simple case of three classes, for convenience, we take this to be Q2. Figure 3(a) shows the query graph. As indicated earlier, the objectives of access planning are to find an execution ordering and then to decide on the primitive access operations. There are $n(n-1) = 6$ possible orderings in this case that follow the left-deep-tree strategy (i.e., $Emp - Dept - Comp$, $Comp - Emp - Dept$, $Dept - Comp - Emp$, $Emp - Comp - Dept$, $Dept - Emp - Comp$, $Comp - Dept - Emp$). However, only four of them are accepted by property P2, thus forming the following PO trees: $J(J(Emp, Dept), Comp)$, $J(J(Dept, Emp), Comp)$, $J(J(Dept, Comp), Emp)$ and $J(J(Comp, Dept), Emp)$. Let us now explore one of these trees, say $J(J(Emp, Dept), Comp)$ (Figure 3(b)). There are six ET trees derived here. As seen above, there are three possibilities for each join operation such as $J(Emp, Dept)$. Let $\texttt{IR}_{Emp,Dept}$ be the intermediate result of this operation, then there would be two execution possibilities for the remaining $J(\texttt{IR}_{Emp,Dept}, Comp)$ as follows:

$ET1$: $I(RI(I_{Comp}, name = \texttt{IBM}), JI^F(R_{Dept \rightarrow Comp}, \texttt{IR}))$
$ET2$: $I(\texttt{IR}, JI^I(R_{Dept \leftarrow Comp}, RI(Comp, name = \texttt{IBM})))$

Continuing this process for all four of the PO trees, we derive a total of 24 possible execution trees (i.e. six execution trees for each PO tree). Figure 3(c) shows the form of $ET1$. In next section we show how can avoid generating the vast number of all possible ET trees.

4.3.3 The selection of 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

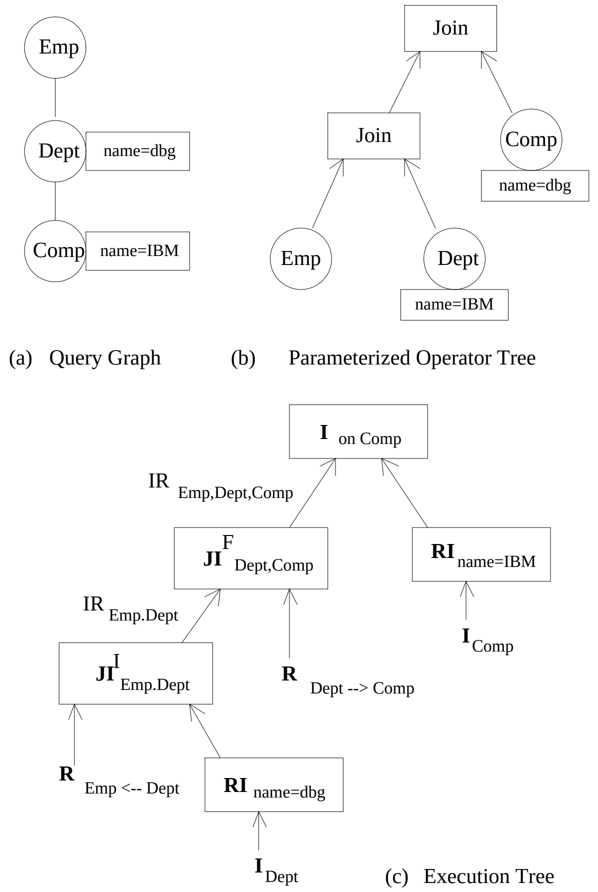

Figure 3: An example of the successive stages in the processing of Q2 query

worst case. Unfortunately, the substitution of the primitive operations on the parameterized tree implies a $O(2^{n-1})$ for the size of the execution space that practically prevents the enumeration of the possible execution strategies as done in [SAC+79].

In order to pick an execution tree that has an efficient cost, we utilize the cost formulas presented earlier to estimate the cost of each access plan corresponding to each execution tree. Furthermore, we need a heuristic search algorithm that is used to walk selectively the space of possible executions.

As far as the heuristic search is concerned, we distinguish two possibilities. The first is a heuristic enumeration of the most promising execution plans based on the selectivity of classes. The second is to use a randomized searching algorithms to reach a near-optimal plan [IK91]. The latter requires some transformation rules that model the transition from one state (execution tree) to another. Here we show two such rules. Assume that A and B represent class subsequences in a PO tree for the query and C_i stands for a single class in it, and join method can be either JI^F or JI^I, then we have two rules as follows:

R1 Join Method Selection:

$$J(J_{method_i}(A,C),B) \rightarrow J(J_{method_j}(A,C),B)$$

R2 Left-hand-side Join Operand Selection:

$$J(C_1,\ldots J(C_{j-1},C_j)\ldots,J((C_{i-1},C_i))),B) \rightarrow$$
$$J(C_1\ldots J(C_{j-1},C_{i-1})\ldots J((C_j,C_i))),B)$$

Provided that both C_j, C_i are connected in QG to some node in the subsequence $C_1 \ldots C_{j-1}$.

In this study, we adopted the heuristic search approach whereby we order the classes based on the cardinality of their participating instances (after applying any restrictions). Thus, the classes that have the fewest instances participating in the query, and hence reduce the size of the intermediate result, will be chosen first. Exploring randomized algorithms is left for future work.

5 Performance Analysis

In this section, we evaluate the performance of the developed join-index-based query processing methods in the context knowledge base management system.

Our analysis is based on two assumptions that differentiate knowledge bases from databases. First, knowledge bases contain many and relatively small attribute domains where databases have fewer and larger attribute domains. Second, in knowledge bases, the attribute values have a temporal dimension. Both of these assumptions are technical rather than general. However, they reflect the key features of Telos-like knowledge bases. The second assumption, in specific, applies to temporal databases for which the first assumption is not a necessary condition.

Under these assumptions, we show that the use of join indices for the processing of path queries outperforms the established *sort merge* and *nested loop* methods for join computations. The methods that we compare with are chosen as representatives of certain approaches on query processing. In specific, 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 (AI-DB *coupling systems*). Our key conclusion is that neither a coupling system nor a traditional DBMS can guarantee good performance for a knowledge base. Instead, query processing techniques which reflect properties of a knowledge base must be sought.

The test knowledge base, used in the experiments, is generated based on the parameters described in table 1. The total number of classes in the knowledge base is 40. We use a uniform average fanout of 3 that connects each class with the others in the aggregation hierarchy. The aggregation hierarchy is four levels deep, and 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 has two primitive attributes where only of them is indexed. Also 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. Finally, a page size of 4K was used throughout. We note here that some of the parameters used here, such as the number of classes, are based on observations obtained from empirical studies of real knowledge bases in [Cha94].

For the above knowledge base, we generated random queries with path expressions in a way that we controlled the fanout and the depth of the query graph (QG). The fanout controls the number of complex attributes emanating from the some node in QG or alternatively the number of path expressions in the query qualification. The depth,

level	class *cardinality*	*attr*1	*attr*2
0	20k...50k	5%...20%	5%...10%
1	10k...20k	5%...20%	5%...10%
2	5k...10k	5%...20%	5%...10%
3	1k...5k	5%...20%	5%...10%

Table 1: Summary of parameters

on the other hand, controls the length of the path expressions in each predicate. In addition, the decision of whether a certain node in QG has a restriction predicate on a simple attribute was made randomly.

For each join strategy, we keep track of the number of page IOs 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* (**NL**) and *sort-merge* (**SM**) 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, and Mem denotes the size of the main memory cache in pages.

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

SM: $M_{c_1} log M_{c_1} + M_{c_2} log M_{c_2} + M_{c_1} + M_{c_2}$

The results for various values of these parameters are shown in Figure 4. Note that in 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 for the reason we mentioned earlier. Keep in mind that as we increase the number of path expressions and the depth, the cost may actually decrease since the addition of new predicates restrict 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 we traverse 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 processed along the path. This can be seen at depth 2 in figures 4(a) and (b) and at depth 3 in figure 4(c). The graph in figure 4(b) further corroborates this observation where the cost is plotted against the number of classes participating in the query graph.

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

6 Conclusions

In this paper we presented a comprehensive approach for dealing with path queries in structurally object-oriented temporal knowledge bases. The main contributions of our work are as follows: We first identified a set of access level operations for executing simple queries, and then we showed how execution plans for more complex queries are constructed out of them. Second, we developed cost formulae for estimating the disk I/O costs and size estimates of the defined access operations and the intermediate processing results. Third, we tackled the problem of optimizing the access planning in queries with arbitrary number of path expressions. Finally, we showed by conducting an experimental study,

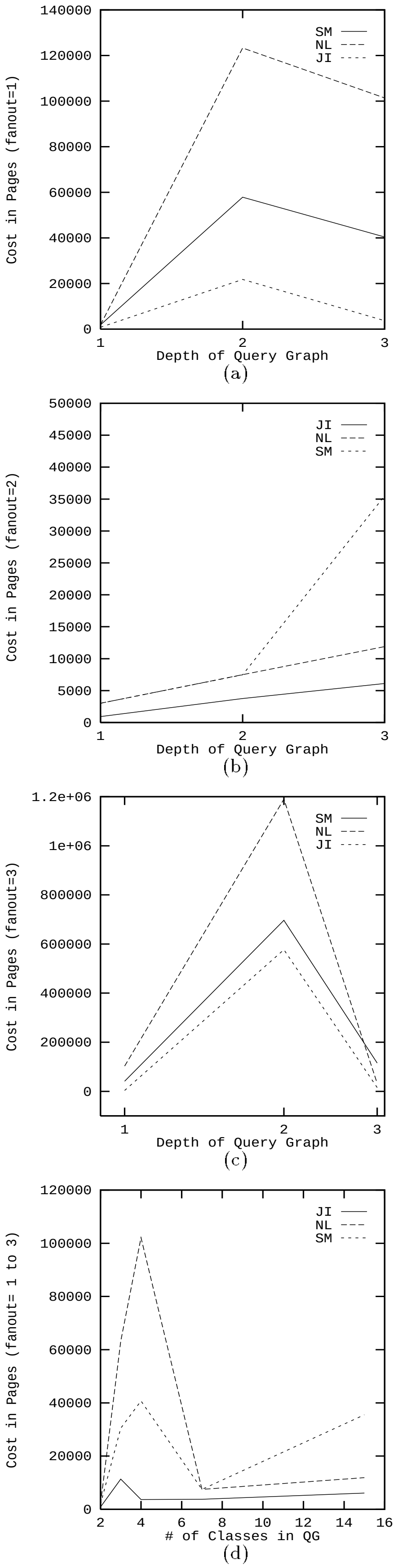

Figure 4: Cost of join operations on path queries

that in the context of our knowledge bases, join-index based query processing achieves better performance than the nest-loop and sort-merge methods.

The significance of the last point is that neither a coupling system nor a traditional DBMS can guarantee good performance for a knowledge base. Instead, query processing techniques which reflect properties of a knowledge base must be sought. This conclusion although is shown in the context of the Telos KBMS has more general scope as further studies have shown [Cha94].

Our work relates to earlier work on path queries for object oriented databases [BK89]. However, the indices proposed there, nested index and path index, are not applicable in our case because of the time versioning of attributes in a knowledge base and the frequency of updates. In fact, our results show that in the presence of time indices consisting of paths of join indices offer a better option [Top93]. The advantages of join indices for supporting efficient navigation in object-oriented databases are also shown in [XH94] where the concept of join index hierarchies is introduced.

As for future work, there are many interesting directions to pursue, including: Extending the cost model so it is capable of handling wider range of queries such as recursive queries. A second direction is exploring techniques such as randomized algorithms to search for the efficient execution plan, and finally applying machine learning techniques to train the access planner with previous experience as it walks the space of possible executions.

Acknowledgments

The results reported in this paper are based on research conducted within the project titled "A Telos Knowledge Base Management System", led by Professor John Mylopoulos and funded by the Province of Ontario through the Information Technology Research Center and the Department of Computer Science of the University of Toronto.

The authors are grateful to Professor John Mylopoulos for his guidance and support. Special thanks are due to our colleagues Dr. Vinay Chaudhri (SRI) and Dimitri Plexousakis (Kansas State University) for the fruitful discussions that we had during the course of this work.

References

[All83] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[BK89] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, 1989.

[Cha94] Vinay K Chaudhri. *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation.* PhD thesis, Department of Computer Science, University of Toronto, 1994.

[CK85] G.P. Copeland and S.N. Khoshafian. A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279, 1985.

[IK91] Y. E. Ioannidis and Y. C. Kang. 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, 1991.

[LG89] Douglas B. Lenat and R.V. Guha. *Building Large Knowledge-based Systems: Representation and inference in the Cyc Project.* Addison-Wesley, 1989.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: A Language for Representing Knowledge About Information Systems. *ACM Transactions on Office Information Systems*, 8(4):325–362, 1990.

[MCP+95] J. Mylopoulos, V. Chaudhri, D. Plexousakis, A. Shrufi, and T. Topaloglou. Building Knowledge Base Management Systems. Technical report, *The VLDB Journal* (to appear), 1995.

[NFF+91] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56, 1991.

[SAC+79] G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM/SIGMOD Conference on Management of Data*, pages 23–34, 1979.

[SMK93] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing Join Orders. Technical Report MIP-9307, Fakultät für Mathematik and Informatik, September 1993.

[Sno87] Richard Snodgrass. The Temporal Query Language TQuel. *ACM Transcactions on Database Systems*, 12(2):247–298, June 1987.

[TIS92] T. Topaloglou, A. Illarramendi, and L. Sbattella. Query Optimization for KBMSs: Temporal, Syntactic and Semantic Transformation. In *Proceedings of the International Conference on Data Engineering*, pages 310–319, 1992.

[Top93] Thodoros Topaloglou. Storage Management for Knowledge Bases. In *Proceedings of Second International Conference on Information and Knowledge Management (CIKM'93)*, 1993.

[Val87] Patrick Valduriez. Join Indices. *ACM Transaction on Database Systems*, 12(2):218–246, June 1987.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proceedings of the 12th VLDB Conference*, pages 101–109, Kyoto, Japan, 1986.

[XH94] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the 20th VLDB Conference*, pages 522–533, 1994.

[Yao77] S.B. Yao. Approximating Block Accesses in Database Organizations. *Communications of ACM*, 20(4):260–261, 1977.