Access Support in Object Bases

Alfons Kemper *Guido Moerkotte*

Universitat Karlsruhe
Fakultat fur Informatik
7500 Karlsruhe, W Germany
Netmail *kemper/moer*@ira uka de

Abstract

In this work *access support relations* are introduced as a means for optimizing query processing in object-oriented database systems The general idea is to maintain redundant separate structures (disassociated from the object representation) to store object references that are frequently traversed in database queries The proposed access support relation technique is no longer restricted to relate an object (tuple) to an atomic value (attribute value) as in conventional indexing Rather, access support relations relate objects with each other and can span over reference chains which may contain collection-valued components in order to support queries involving path expressions We present several alternative extensions of access support relations for a given path expression, the best of which has to be determined according to the application-specific database usage profile An analytical cost model for access support relations and their application is developed This analytical cost model is, in particular, used to determine the best access support relation extension and decomposition with respect to the specific database configuration and application profile

1 Introduction

Object-oriented database systems constitute a promising approach towards supporting technical application domains Several object-oriented data models have been developed over the last couple of years However, these systems are still not adequately optimized for applications which involve a lot of associative search for objects on secondary memory they still have problems to keep up with the performance achieved by, e g , relational DBMSs

Yet it is essential that the object-oriented systems will yield at least the same performance that relational systems achieve otherwise their acceptance in the engineering field is jeopardized even though they provide higher functionality than conventional DBMSs by, e g , incorporation of type extensibility and object-specific behavior within the model Engineers are generally not willing to trade performance for extra functionality and expressive power Therefore, we conjecture that the next couple of years will show an increased interest in optimization issues in the context of object-oriented DBMSs The contribution of this paper can be seen as one important piece in the mosaic of performance enhancement methods for object-oriented database applications the support of object access along reference chains

In relational database systems one of the most performance-critical operations is the *join* of two or more relations A lot of research effort has been spent on expediting the join, e g , access structures to support the join, the *sort-merge* join, and the *hash-join* algorithm were developed Recently, the binary join index structure [11] was designed as another optimization method for this operation

In object-oriented database systems with object references the join based on matching attribute values plays a less predominant role More important are object accesses along reference chains leading from one object instance to another Some authors, e g , [2], call this kind of object traversal *functional join* This work presents an indexing technique, called *access support relations*, which is designed to support the functional join along arbitrary long attribute chains where the chain may even contain collection-valued attributes

The access support relations described in this paper constitute a generalization of the binary join indices originally proposed for the relational model [11], and later extended for object models [3,12] Rather than relating only two relations (or object types) our technique allows to support access paths of arbitrary length Our indexing technique subsumes and extends several other previously

proposed strategies for optimizing access along attribute chains in object bases The index paths in GemStone [7] are restricted to chains that contain only single-valued attributes and their representation is limited to binary partitions of the access path Similarly, the object-oriented access techniques described for the Orion model [6,1] are extended in several dimensions in our framework

Our technique differs in three major aspects from the two aforementioned approaches

- access support relations allow collection-valued attributes within the attribute chain
- access support relations may be maintained in four different extensions The extension determines the amount of (reference) information that is kept in the index structure
- access support relations may be decomposed into arbitrary partitions This allows the database designer to choose the best extension and partition according to the application characteristics

Also the (separate) replication of object values as proposed for the Extra object model [9] and for the PostGres model [10,8] are subsumed by our technique

The remainder of this paper is organized as follows Section 2 introduces our Generic Object Model (*GOM*), which serves as the research vehicle for this work, and some simplified application example to highlight the requirements on object-oriented access support Then, in section 3 the access support relations are formally defined In section 4 we start the development of an analytical cost model for our indexing technique by estimating the cardinalities of various representations of access support relations Section 5 describes the utilization of access support relations in query evaluation and estimates the performance enhancement on the basis of secondary page accesses Section 6 is dedicated to presenting some sample results of operation mix costs for a few selected application characteristics Section 7 concludes this paper

2 The Object Model GOM

This research is based on an object-oriented model that unites the most salient features of many recently proposed models in one coherent framework the Generic Object Model *GOM* The interesting aspects of *GOM* concerning the access support relations are

object identity each object instance has an identity that remains invariant throughout its lifetime The object identifier (OID) is invisible for the database user, it is used by the system to reference objects This allows for shared subobjects because the same object may thus be associated with many database components Here, OIDs are denoted $\#i_0$, $\#i_1$,

type constructors the most basic type constructor is the tuple constructor which aggregates differently typed attributes to one object In addition, *GOM* has the two built-in collection type constructors set, denoted as {}, and list[1], denoted as <> *GOM* also provides for subtyping of tuple-structured types, however this is irrelevant for the present discussion

strong typing *GOM* is strongly typed, meaning that all database components, e g , attributes, set elements, etc, are constrained to a particular type This, in particular means that all path expressions are typed However, the constrained type constitutes only an upper bound, the actually referenced instance may be a subtype-instance thereof

object references assignment of an object to an attribute, a variable or insertion of an object into a set corresponds to maintaining a reference to the respective object Thus, object references are stored unidirectional, conforming to almost all published object models

2.1 Type Definitions

A *linear* path is an attribute chain that contains only attributes referring to a single object Single-object-valued attributes are only useful to model 1 1, or N 1 relationships In order to represent 1 M, or general N M relations one needs to incorporate collection-valued attributes, i e , attributes referring to a set or list instance To illustrate this let us define a vastly simplified database schema for modeling a *Company* composed of a set of *Divisions* Each *Division Manufactures* a set of *Products*, which themselves are composed of *BaseParts*

The schema is outlined below

```
type Company is {Division},
type Division is [Name  STRING,
                  Manufactures  ProdSET],
type ProdSET is {Product},
type Product is [Name  STRING,
                 Composition  BasePartSET],
type BasePartSET is {BasePart},
type BasePart is [Name  STRING,
                  Price  DECIMAL],
```

Additionally we assume the existence of a reference to a given company

```
var Mercedes  Company,
```

[1] Lists are not further considered in this paper, though

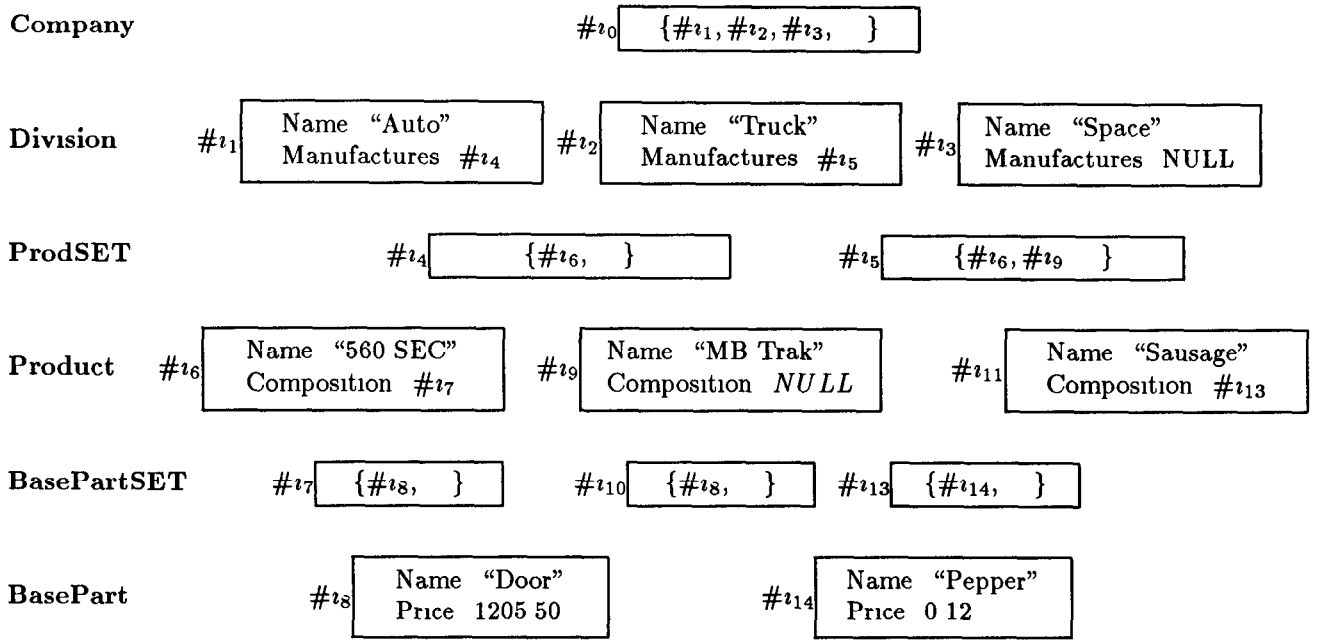

Figure 1 Database Extension With Non-Linear Paths

A sample extension of this schema is presented in Figure 1 Note that an object is represented as a triple $(\#i_j, v, t)$ where $\#i_j$ is the object identifier, v the object representation, and t the object's type References, e g , $\#i_1$ *Manufactures*, are maintained unidirectionally by storing the associated object's identifier, $\#i_4$, within the domain object $(\#i_1)$

Now let us illustrate some typical queries in an SQL-like syntax which access objects along references (possibly leading through sets)

Query 1. Which *Division* uses a *BasePart* named "Door"?

select *d* Name
from *d* in Mercedes,
b **in** *d* Manufactures Composition
where *b* Name = "Door"

"*d Manufactures Composition*" is a set-valued path expression with the following semantics

$$d\ Manufactures\ Composition = \bigcup_{m \in d\ Manufactures} m\ Composition$$

Query 2 Retrieve the *Name* of all the *BaseParts* used by the *Division* named "Auto"

select *d* Manufactures Composition Name
from *d* **in** Mercedes
where *d* Name = "*Auto*"

3 Access Support Relations

As mentioned earlier access paths are used to support query evaluation More precisely, access paths allow the fast selection of those members of an object collection which fulfill a given selection criteria based on object references along an attribute chain or path expression A path expression or attribute chain is defined as follows

Definition 3 1 *Let* $t_0, \quad , t_n$ *be (not necessarily distinct) types A path expression on* t_0 *is an expression* $t_0\ A_1 \quad A_n$ *iff for each* $1 \leq i \leq n$ *one of the following conditions holds*

- *Type* t_{i-1} *is defined as* **type** t_{i-1} **is** [$, A_i \ t_i, \ ]$, *i e , a tuple type containing at least the attribute* A_i *of type* t_i
- *Type* t_{i-1} *is defined as* **type** t_{i-1} **is** [$, A_i \ t'_i, \ ]$ *and the type* t'_i *is defined as* **type** t'_i **is** $\{t_i\}$ *In this case we speak of a set occurrence at* A_i *in the path* $t_0\ A_1 \quad A_n$

The type t_{i-1} *is called the domain of* A_i, *and* t_i *is called the range of* A_i

The second part of the definition is useful to support access paths through sets[2] If it does not apply to a given path the path is called *linear*

For simplicity we require each path expression to originate in some type t_0, alternatively we could have chosen a particular collection C of elements of type t_0 as the anchor of a path (leading to more difficult definitions and cost functions, though)

[2]Note, however, that we do not permit powersets

Since an access path can be seen as a relation we will use relation extensions to represent access paths The next definition maps a given path expression to the underlying access support relation declaration

Definition 3.2 *Let $t_0, \ldots, t_n$ be types, $t_0.A_1.\ldots.A_n$ be a path expression, and k the number of set occurrences in $t_0.A_1.\ldots.A_n$ Then the access support relation $E_{t_0.A_1.\ldots.A_n}$ is of arity $n+k+1$ and has the following form*

$$E_{t_0.A_1.\ldots.A_n}\ [S_0, \ldots, S_{n+k}]$$

The domain of the attribute S_0 is the set of identifiers (OIDs) of objects of type t_0 For $(1 \leq i \leq n)$ let $k(i)$ be the number of set occurrences before A_i, i e , set occurrences at A_j for $j < i$ Then the domain of the attribute $S_{i+k(i)}$ is the set of OIDs that identify objects of type

- *t_i, if A_i is a single-valued attribute*
- *t_i', if A_i is a set-valued attribute In this case the domain of $S_{i+k(i)+1}$ is the set of OIDs of type t_i*

If t_n is an atomic type then the domain of S_{n+k} is t_n, i e , values are directly stored in the access support relation If the underlying path expression is clear from context we will write E instead of $E_{t_0.A_1.\ldots.A_n}$

Let further m be defined as $m = n + k$

We distinguish several possibilities for the extension of such relations To define them for a given path expression $t_0.A_1.\ldots.A_n$ we need n temporary relations $E_0, \ldots, E_{n-1}$

Definition 3 3 *For each A_j $(1 \leq j \leq n)$ we construct the temporary relation E_{j-1} Depending on the domain of A_j the relation E_{j-1} is*

1 *binary, if A_j is a single-valued attribute In this case the relation E_{j-1} contains the tuples $(id(o_{j-1}), id(o_j))$ for every object o_{j-1} of type t_{j-1} and o_j of type t_j such that $o_{j-1}.A_j = o_j$ If t_j is an atomic type then $id(o_j)$ corresponds to the value $o_{j-1}.A_j$*
2 *ternary, if the attribute A_j is a set-valued attribute Then the relation E_{j-1} contains the tuples $(id(o_{j-1}), id(o_j'), id(o_j))$ for every object o_{j-1} of type t_{j-1}, o_j' of type t_j', and o_j of type t_j such that $o_{j-1}.A_j = o_j'$ and the set o_j' contains o_j In the special case that o_j' is an empty set the relation E_{j-1} contains the tuple $(id(o_{j-1}), id(o_j'), NULL)$*

Example: Recall the *Company* database extension of Figure 1 For the underlying schema we could declare the access support relation on the path expression *Division Manufactures Composition Name* This results in 3 temporary relations E_0, E_1, and E_2

E_0

$OID_{Division}$	$OID_{ProdSET}$	$OID_{Product}$
$\#i_2$	$\#i_5$	$\#i_9$
$\#i_1$	$\#i_4$	$\#i_6$

E_1

$OID_{Product}$	$OID_{BasePartSET}$	$OID_{BasePart}$
$\#i_{11}$	$\#i_{13}$	$\#i_{14}$
$\#i_6$	$\#i_7$	$\#i_8$

E_2

$OID_{BasePart}$	$VALUE_{Name}$
$\#i_{14}$	"Pepper"
$\#i_8$	"Door"

Let us now introduce different possible *extensions* of a given access support relation E For a given path expression $t_0.A_1.\ldots.A_n$ we distinguish four extensions

1 the *canonical* extension, denoted E_{can} contains only information about complete paths, i e , paths originating in t_0 and leading to t_n Therefore, it can only be used to evaluate queries that originate in an object of type t_0 and "go all the way" to t_n
2 the *left-complete* extension E_{left} contains all paths originating in t_0 but not necessarily leading to t_n, but possibly ending in a *NULL*
3 the *right-complete* extension E_{right}, analogously, contains paths leading to t_n, but possibly originating in some object o_j of type t_j which is not referenced by any object of type t_{j-1} via the A_j attribute
4 finally, the full extension E_{full} contains all partial paths, even if they do not originate in t_0 or do end in a *NULL*

Definition 3.4 (Extensions) *Let $\bowtie$ $(\overrightarrow{\bowtie}, \bowtie, \overleftarrow{\bowtie})$ denote the natural (outer, left outer, right outer) join on the last column of the first relation and the first column of the second relation Then the different extensions are obtained as follows*

$$\begin{aligned} E_{can} &= E_0 \bowtie \ldots \bowtie E_{n-1} \\ E_{full} &= E_0 \overrightarrow{\bowtie} \ldots \overrightarrow{\bowtie} E_{n-1} \\ E_{left} &= (\ldots(E_0 \overrightarrow{\bowtie} E_1) \overrightarrow{\bowtie} \ldots \overrightarrow{\bowtie} E_{n-1}) \\ E_{right} &= (E_0 \overleftarrow{\bowtie} (\ldots \overleftarrow{\bowtie} (E_{n-2} \overleftarrow{\bowtie} E_{n-1}) \ldots)) \end{aligned}$$

Example: For our example application the full extension contains also the incomplete paths, i e , those that lead to a NULL (e g , the first tuple in the extension shown in Figure 2) or those not originating in an object o_0 of type t_0 (the second tuple in E_{full} shown in Figure 2) Even partial paths not originating in t_0 and leading to a NULL are to be included The extension E_{can} would only contain the last tuple shown in E_{full} E_{right} would not

E_{full}

$OID_{Division}$	$OID_{ProdSET}$	$OID_{Product}$	$OID_{BasePartSET}$	$OID_{BasePart}$	$VALUE_{Name}$
$\#i_2$	$\#i_5$	$\#i_9$	NULL	NULL	NULL
NULL	NULL	$\#i_{11}$	$\#i_{13}$	$\#i_{14}$	"Pepper"
$\#i_1$	$\#i_4$	$\#i_6$	$\#i_7$	$\#i_8$	"Door"

Figure 2 A Sample Extension of E_{full}

contain the first tuple shown in Figure 2, whereas in E_{left} the second tuple would be omitted

Aside from different extensions of the access support relation also several decompositions are possible, which are discussed now Since not all of them are meaningful we define a decomposition as follows (Remember $m = n + k$)

Definition 3 5 (Decomposition) *Let E be an $(m+1)$-ary access support relation with attributes $S_0, \ldots, S_m$ Then the relations*

$$E^{0,i_1} \quad [S_0, \ldots, S_{i_1}] \quad \text{for } 0 < i_1 \leq m$$
$$E^{i_1,i_2} \quad [S_{i_1}, \ldots, S_{i_2}] \quad \text{for } i_1 < i_2 \leq m$$
$$E^{i_k,m} \quad [S_{i_k}, \ldots, S_m] \quad \text{for } i_k < m$$

are called a decomposition of E The individual relations $E^{i_j,i_{j+1}}$, called partitions, are materialized by projecting the corresponding attributes of E If every partition is a binary relation the decomposition is called binary The above decomposition is denoted by $(0, i_1, i_2, \ldots, i_k, m)$

Note that m and n are equal only in the case that there is no set occurrence along the path If there is any then $m > n$ Under the assumption that there is no set sharing, the set identifiers may be dropped from the access support relation This results in $m = n$ To simplify the analysis we will do so for the examples considered in the next section Note, however, that the analytical cost model captures the general case if one reads n as m

The last question discussed in this section concerns the usefulness of the above defined decompositions

Theorem 3 6 *Every decomposition of an access support relation is lossless*

The proof of this theorem is obvious since we decompose along multi-valued dependencies

3.1 Sharing of Access Support Relations

Consider the following two path expressions

$$P_1 \equiv \overbrace{\underbrace{t_0\ A_1 \quad A_i}_{t_i}\ A_{i+1} \quad A_{i+j}}^{t_{i+j}}\ A_{i+j+1} \quad A_n$$

$$P_2 \equiv \overbrace{\underbrace{s_0\ B_1 \quad B_l}_{t_i}\ A_{i+1} \quad A_{i+j}}^{t_{i+j}}\ C_1 \quad C_q$$

If $t_0\ A_1 \quad A_i$ and $s_0\ B_1 \quad B_l$ are path expressions both leading to objects of type t_i then part of the access support predicates may be shared

This, in general, is only possible when a *full* extension of at least one of the access support relations is maintained Let E_{full} be the full extension for the path P_1, and $\bar{E}_{full}$ the full extension of the access support relation for path P_2 Then the decomposition $(0, i, i+j, n)$ of E_{full} and $(0, l, l+j, r)$[3] of $\bar{E}_{full}$ share a common partition, i e , $E^{i,i+j}_{full} = \bar{E}^{l,l+j}_{full}$

Thus we obtain the following five partitions

$$E^{0,i}_{full}\ [OID_{t_0}, \ldots, OID_{t_i}] \quad \bar{E}^{0,l}_{full}\ [OID_{s_0}, \ldots, OID_{t_i}]$$

$$E^{i,i+j}_{full} = \bar{E}^{l,l+j}_{full}\ [OID_{t_i}, \ldots, OID_{t_{i+j}}]$$

$$E^{i+j,n}_{full}\ [OID_{t_{i+j}}, \ldots, OID_{t_n}]\ \bar{E}^{l+j,r}_{full}\ [OID_{t_{i+j}}, \ldots, OID_{s_r}]$$

The five partitions may then, individually, be further decomposed

In general, this sharing is only possible for full extensions Exceptions are

- if both paths P_1 and P_2 originate in t_0, i e , $i = l = 0$ and $t_0 = s_0$ Then the sharing is also possible for left-complete extensions
- if both paths lead to t_n, i e , their right-most part is identical, then the corresponding partition of the right-complete extensions may be shared

This should indicate that there may exist a higher level of organization, i e , an access support relations manager which controls (and constrains) the possible extensions and decompositions

[3] the length of path P_2 is $r = l + j + q$

4 Analytical Cost Model: Cardinality of Access Relations

In this section we develop the basis of our analytical cost model a model of the application profile and formulas for the cardinalities of access support relations under different extensions and decompositions Later on, the cost model is used to derive the best physical database design, i e , to find the best extension and decomposition of a given path expression according to the predetermined operation mix

4.1 Preliminaries

Before giving the sizes of the relations we introduce some parameters that model the characteristics of an application These are listed in Figure 3

application-specific parameters	
parameter	semantics (and derivation/default)
n	length of access path
c_i	total number of objects of type t_i
d_i	the number of objects of type t_i for which the attribute A_{i+1} is not *NULL*
fan_i	the number of references emanating on the average from the attribute A_{i+1} of an object o_i of type t_i
$shar_i$	the average number of objects of type t_i that reference the same object in t_{i+1} If no value for $shar_i$ is determined by the user, it is derived as $shar_i = min(1, (d_i * fan_i)/c_{i+1})$
$size_i$	average size of objects of type t_i
system-specific parameters	
PageSize	net size of pages, which is set to 4056
OIDsize	size of object identifiers, default is 8
PPsize	size of page pointer, default is 4
B^+_{fan}	fan out of the B^+ tree, which is derived as $\lfloor PageSize/(PPsize + OIDsize) \rfloor$

Figure 3 System and Application Parameters

4.1.1 Some Derived Quantities

The number of objects in t_i which are referenced by at least one object in t_{i-1} is denoted as e_i

$$e_i = \left\lceil \frac{d_{i-1} * fan_{i-1}}{shar_{i-1}} \right\rceil$$

The probability P_{A_i} that an object o_i of type t_i has a defined A_{i+1} attribute value is

$$P_{A_i} = \frac{d_i}{c_i}$$

The probability P_{H_i} that a particular object o_i of type t_i is "hit" by a reference emanating from some object of type t_{i-1} is

$$P_{H_i} = \frac{e_i}{c_i}$$

Let us now derive the probability that, for some object o_i of type t_i none of the fan_i references of the attribute o_i A_{i+1} hits a particular object $o_{i+1} \in t_{i+1}$, which belongs to the e_{i+1} referenced objects

This value is deduced by using the number of fan_i-element subsets of the e_{i+1} objects of type t_{i+1} This number is given as the binomial coefficient

$$\binom{e_{i+1}}{fan_i} = \frac{e_{i+1}!}{fan_i!(e_{i+1} - fan_i)!}$$

Then, the probability that the particular object o_{i+1} is not hit is given as

$$\frac{\binom{e_{i+1}-1}{fan_i}}{\binom{e_{i+1}}{fan_i}} = \frac{e_{i+1} - fan_i}{e_{i+1}} = 1 - \frac{fan_i}{e_{i+1}}$$

The probability that o_{i+1} is not hit by any of the references emanating from a subset $\{o_i^1, o_i^2, \quad , o_i^k\}$ of objects of type t_i, all of whose A_i attributes are defined, is

$$\left(1 - \frac{fan_i}{e_{i+1}}\right)^k$$

For $0 \le i < j \le n$ we now define $RefBy(i, j, k)$, which denotes the number of objects in t_j which lie on at least one (partial) path emanating from a k-element subset of t_i

$$RefBy(i,j,k) = \begin{cases} e_{i+1} * \left(1 - \left(1 - \frac{fan_i}{e_{i+1}}\right)^k\right) & j = i+1 \\ e_j * \left(1 - \left(1 - \frac{fan_{j-1}}{e_j}\right)^{E(i,j,k)}\right) & \text{else} \end{cases}$$

where the exponent $E(i,j,k) = RefBy(i, j-1, k) * P_{A_{j-1}}$

Further the probability, denoted $P_{RefBy}(i,j)$, that a path between anyone object in t_i and a particular object o_j in t_j exists for $0 \le i < j \le n$, is derived as

$$P_{RefBy}(i,j) = \begin{cases} 1 & i = j \\ \frac{RefBy(i,j,d_i)}{c_j} & \text{else} \end{cases}$$

Let $Ref(i,j,k)$ denote the number of objects of type t_i which have a path leading to some element of a k-element subset of objects of type t_j for $0 \le i < j \le n$ This value can be approximated as

$$Ref(i,j,k) = \begin{cases} d_i * \left(1 - \left(1 - \frac{shar_i}{d_i}\right)^k\right) & j = i+1 \\ d_i * \left(1 - \left(1 - \frac{shar_i}{d_i}\right)^{E'(i,j,k)}\right) & \text{else} \end{cases}$$

where the exponent $E'(i,j,k) = Ref(i+1,j,k) * P_{H_{i+1}}$

Let $P_{Ref}(i,j)$ be the probability that a given object in t_i has at least one path leading to any one object in t_j Then

$$P_{Ref}(i,j) = \begin{cases} 1 & i = j \\ \frac{Ref(i,j,c_j)}{c_i} & \text{else} \end{cases}$$

The number of paths between the objects in t_i and the objects in t_j can be estimated by

$$path(i,j) = d_i * fan_i * \prod_{l=i+1}^{j-1} (P_{A_l} * fan_l)$$

4.2 Cardinalities of Access Support Relations

We can now deduce closed formulas for the number of tuples in the access support relations

Let us first introduce two more probabilistic values Let $P_{lb}(i,j)$ denote[4] the probability that a particular object of type t_j is not "hit" by any path emanating from some object in t_i for $0 \le i < j \le n$

$$P_{lb}(i,j) = \begin{cases} 1 - P_{RefBy}(i,j) & i < j \\ 1 & \text{else} \end{cases}$$

Analogously, let $P_{rb}(i,j)$ denote[5] the probability that a particular object of type t_i contains no emanating path to some object in t_j for $0 \le i < j \le n$

$$P_{rb}(i,j) = \begin{cases} 1 - P_{Ref}(i,j) & i < j \\ 1 & \text{else} \end{cases}$$

Let $\#E_X^{i,j}$ denote the cardinality of the access relation partition $E_X^{i,j}$ for the general decomposition (,i,j,) under the extension X, i e , $X \in \{can, full, left, right\}$

$$\#E_{can}^{i,j} = P_{RefBy}(0,i) * path(i,j) * P_{ref}(j,n)$$

$$\#E_{full}^{i,j} = \sum_{k=1}^{j-i} \sum_{l=i}^{j-k} P_{lb}(max(i,l-1),l) * path(l,l+k) * P_{rb}(l+k, min(j,l+k+1))$$

$$\#E_{left}^{i,j} = \sum_{k=1}^{j-i} P_{RefBy}(0,i) * path(i,i+k) * P_{rb}(i+k, min(j,i+k+1))$$

$$\#E_{right}^{i,j} = \sum_{k=1}^{j-i} P_{lb}(max(i,j-k-1),j-k) * path(j-k,j) * P_{ref}(j,n)$$

[4] *lb* left-bound

[5] *rb* right-bound

4.3 Storage Representation of Access Support Relations

Following the proposal by Valduriez [11] for join indices an access support relation (partition) $E_X^{i,j}$ is stored in two redundant B^+ trees, one being keyed (clustered) on the first attribute, i e , OIDs of objects of type t_i, and the second B^+ tree being clustered on the last attribute, i e , OIDs of t_j objects In this way we can achieve a fast look-up of all tuples (partial paths) originating in some object o_i of type t_i and all (partial) paths leading to some object o_j of type t_j Particularly, in this way the semi-join of access support relation partitions is efficiently performed in both directions The right-to-left semi-join, e g ,

$$(\quad \ltimes (E_X^{i',i} \ltimes (E_X^{i,j} \ltimes E_X^{j,j'}) \ltimes \quad) \quad)$$

is performed for evaluating a backward query, the left-to-right semi-join to evaluate a forward query (cf section 5)

4.4 Storage Costs for Access Support Relations

The size of a tuple in the access support relation $E_X^{i,j}$ in bytes is

$$ats^{i,j} = OIDsize * (j - i + 1)$$

The number of tuples in access relation $E_X^{i,j}$ per page

$$atpp^{i,j} = \left\lfloor \frac{PageSize}{ats^{i,j}} \right\rfloor$$

The size of the access relation $E_X^{i,j}$ in bytes

$$as_X^{i,j} = \#E_X^{i,j} * ats^{i,j}$$

The approximate number of pages needed to store the access relation $E_X^{i,j}$

$$ap_X^{i,j} = \left\lceil \frac{\#E_X^{i,j}}{atpp^{i,j}} \right\rceil$$

Note that this value has to be multiplied by a factor of 2 due to the redundant maintenance of access support relations

5 Query Processing and Update Costs

In this section we first evaluate the applicability and the costs of the different extensions and decompositions to query processing

5.1 Query Costs

To compare the query evaluation costs we consider abstract, representative query examples of the following two forms

Backward Queries In this query expression the objects $o \in C$ are retrieved, where C is a collection of t_i instances The resulting objects are selected based on the membership of some other object o_j of type t_j in the path expression $o\ A_{i+1} \quad A_j$

$$Q^{i,j}(bw) \equiv \textbf{select } o \textbf{ from } o \textbf{ in } C \quad /* \text{ set of } t_i \text{ instances } */ \textbf{ where } o_j \textbf{ in } o\ A_{i+1} \quad A_j$$

Forward Queries Forward queries retrieve objects of type t_j which can be reached via a path emanating from some given object o of type t_i

$$Q^{i,j}(fw) \equiv \textbf{select } o\ A_{i+1} \quad A_j \textbf{ from } o \textbf{ in } C \quad /* \text{ set of } t_i \text{ instances } */ \textbf{ where}$$

Let us now investigate the applicability of various extensions of an access support relation for the path $t_o\ A_1 \quad A_n$ The full extension can be used to support the evaluation of all path expressions of the form $o\ A_i \quad A_j$, i e , all sub-paths of the path expression $t_0\ A_1 \quad A_n$ On the other hand, the canonical extension can only be used if $i = 0$ and $j = n$ The left-complete extension can support the evaluation if $i = 0$, the right-complete extension is only applicable if $j = n$

Unfortunately, the space limitations do not allow us to derive the analytical formulas for estimating the costs of queries under different access support relations, see [4] for a more detailed treatment

Query Costs for an Example Application Figure 4 visualizes the cost of a backward query of the form $Q^{0,4}(bw)$ for the application-specific parameters shown below (the path under consideration is of length 4)

number of objects	c_0	c_1	c_2	c_3	c_4
	100	500	1000	5000	10000
# objects with defined A_{i+1} attr	d_0	d_1	d_2	d_3	d_4
	90	400	8000	2000	—
fan-out	f_0	f_1	f_2	f_3	f_4
	2	2	3	4	—
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$
	500	400	300	300	100

The access support relations were either decomposed into binary partitions (*bi*) or non-decomposed (*no dec*) As expected, the query costs for non-decomposed access relations are slightly lower than for binary decomposed relations For this application profile the performance gain is in the order of a factor of 100, for larger databases the performance gain is even more drastic (the performance gain grows proportional to the database size)

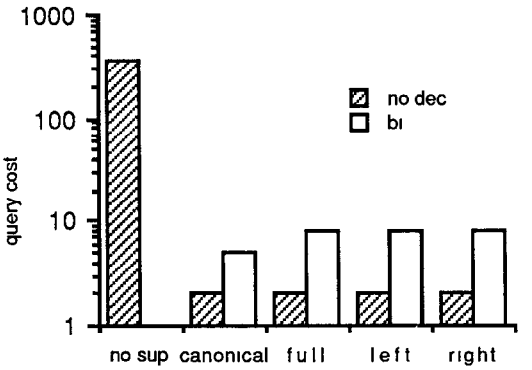

Figure 4 Query Costs for a Backward Query

5.2 Update Costs

For the different extension and decomposition possibilities we now consider the dynamic aspect of maintenance Of course, updates in the object base have to be reflected in the access relation extensions

We consider the insertion and deletion of an object into/from a set-valued attribute (single-valued attributes are a special case) Thus, we distinguish the following two abstract operations

$$ins^i \equiv \textbf{insert } o \textbf{ into } o_i\ A_{i+1}$$
$$del^i \equiv \textbf{delete } o \textbf{ from } o_i\ A_{i+1}$$

We assume that the object o_i is of type t_i, and o is of type t_{i+1} Note, that the costs for both update operations are essentially the same The cost formulas are again developed in [4] We consider only "pure" update costs, that is, the costs of the queries to locate the objects o_i and o is not included in our update costs Therefore, some cost functions (cf Figure 5 and 6) may actually decrease as the update probability increases, this happens when the pure update cost is lower than the query costs

6 Evaluation

In this chapter we demonstrate the cost estimates for a few selected application examples Before doing so, we need a model of a database load profile, called an *operation mix*

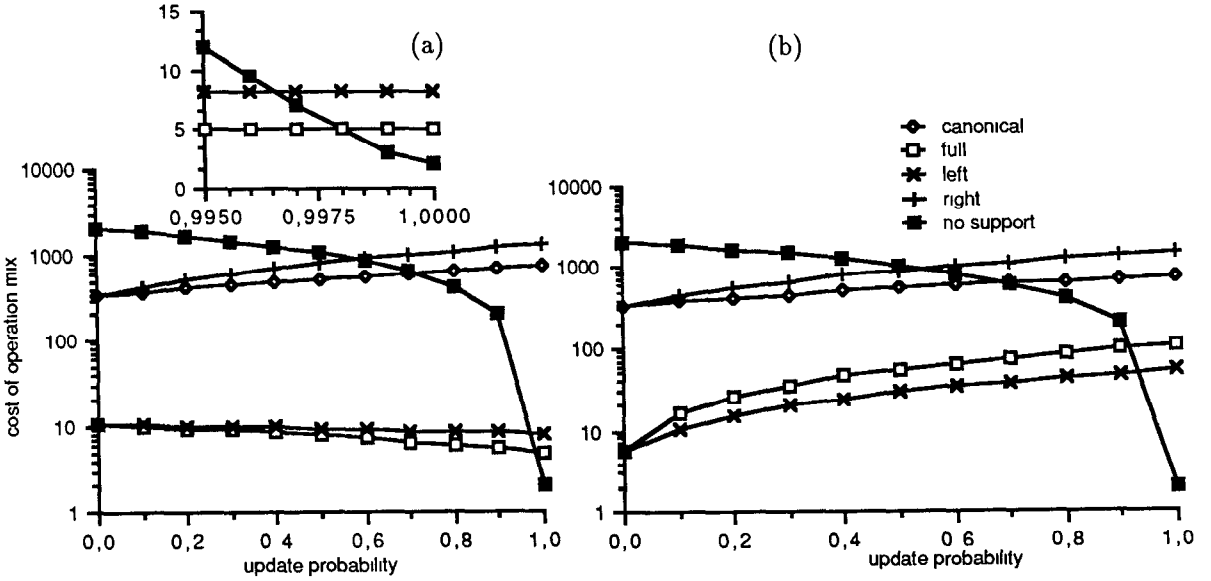

Figure 5 Cost of Operation Mix for two Decompositions (a) binary decomposition, (b) the decomposition (0,3,4)

6.1 Modeling an Operation Mix

In our analytical cost model an operation mix M is described as a triple

$$M = (Q_{mix}, U_{mix}, P_{up})$$

Here, Q_{mix} is a set of weighted queries of the form

$$Q_{mix} = \{(w_1, q_1), \quad , (w_p, q_p)\}$$

where for $(1 \leq i \leq p)$ the q_i are queries and w_i are weights, i e , w_i constitutes the probability that among the listed queries in Q_{mix} q_i is performed It follows that $\sum_{i=1}^{p} w_i = 1$ has to hold

Analogously, the update mix U_{mix} is described Finally, the value P_{up} determines the update probability, i e , the probability that a given database operation turns out to be an update

6.2 Update Mix under Binary and Non-Binary Decomposition

This example is based on the same application profile as introduced in section 5 1 Let us derive the costs for a pre-determined operation profile

$$Q_{mix}=\{(1/2, Q^{0,4}(bw)), (1/4, Q^{0,3}(bw)), (1/4, Q^{1,2}(fw))\}$$

$$U_{mix} = \{(1/2, ins^2), (1/2, ins^3)\}$$

This means that, when a query is performed, the first one is chosen with probability 0 5, and either of the remaining is selected with probability 0 25 The update operations are selected with equal probability

Figure 5 a shows the (normalized) costs under binary decomposition for different update probabilities P_{up} ranging between 0 0 1 0 It can be seen that for an update probability less than 0 3 the left-complete extension and the full extension incur about the same cost The break even point between no support and full extension is at an update probability of 0 998 as shown in the upper left-hand plot[6]

The experiment was run again for the (0, 3, 4) decomposition of the access support relations The result is shown in Figure 5 b In this case the left-complete extension is generally superior to the other extensions Comparing Figures 5 a and 5 b we conclude that the binary decomposition for full extension is better than the decomposition (0, 3, 4) (left-complete extension) for update probabilities exceeding 0 1

6.3 Comparison: Left-Complete vs Full Extension

Let us now consider the following, larger database profile with a path expression of length 5

number of objects	c_0	c_1	c_2	c_3	c_4	c_5
	1000	1000	5000	10^4	10^5	10^5
#obj with def A_{i+1}	d_0	d_1	d_2	d_3	d_4	d_5
	100	1000	3000	8000	10^5	—
fan-out	f_0	f_1	f_2	f_3	f_4	f_5
	2	2	3	4	10	—
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$	$size_5$
	600	500	400	300	300	100

[6]Note, that some cost functions decrease as the update probability increases because the query costs that may be needed to perform an update is not included in the update costs

For this application characterization the normalized costs for a database operation mix consisting of the following queries and updates was computed

$$Q_{mix}=\{(1/3,Q^{0,5}(bw)),(1/3,Q^{0,4}(bw)),(1/3,Q^{0,5}(fw))\}$$
$$U_{mix}=\{(1/3,ins^3),(1/3,ins^0,(1/3,ins^4)\}$$

In Figure 6 the costs for the operation mix under left-complete and full extension of the access relations are plotted for two different decompositions (1) binary decomposition $(0,1,2,3,4,5)$ and (2) the decomposition $(0,3,4,5)$ It turns out that up to an update probability

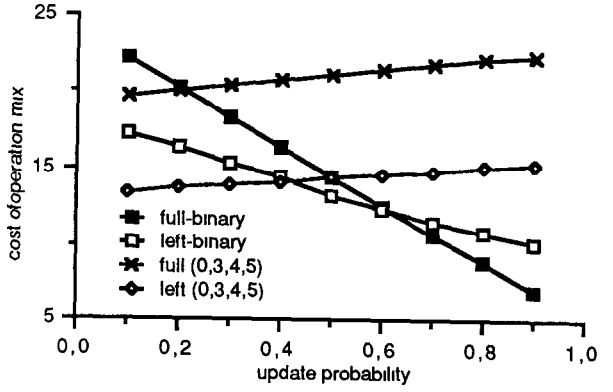

Figure 6 Operation Mix for Full and Left-Complete Access Relations

of 0 4 the left-complete, decomposition $(0,3,4,5)$ is optimal Then, for an update probability $0\ 4 \leq P_{up} \leq 0\ 6$ the left-complete, binary decomposition is superior Finally, for $P_{up} \geq 0\ 6$ the full extension under binary decomposition is the optimal choice

6.4 Comparison: Right-Complete vs Full Extension

In this experiment the following application profile is being used

number of objects	c_0	c_1	c_2	c_3	c_4	c_5
	10^5	10^5	50000	10^4	1000	1000
#obj with def A_{i+1}	d_0	d_1	d_2	d_3	d_4	d_5
	10^5	10^4	30000	10^4	100	100
fan-out	f_0	f_1	f_2	f_3	f_4	f_5
	1	10	20	4	1	—
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$	$size_5$
	600	500	400	300	200	700

For this application characterization the normalized costs for a database operation mix consisting of the following queries and updates was computed

$$Q_{mix} = \{(1/2,Q^{0,5}(bw)),(1/4,Q^{1,5}(bw)),(1/4,Q^{2,5}(bw))\}$$
$$U_{mix} = \{(1,ins^3))\}$$

Figure 7 visualizes the costs for the operation mix under the following decompositions of the right-complete and full extension

1 the binary decomposition $(0,1,2,3,4,5)$
2 the decomposition $(0,3,5)$

It turns out that the latter decomposition is always superior For very low update probabilities less than 0 005 the right-complete extension is better than the full extension under this particular decomposition This break-even point is shown in the upper plot of Figure 7

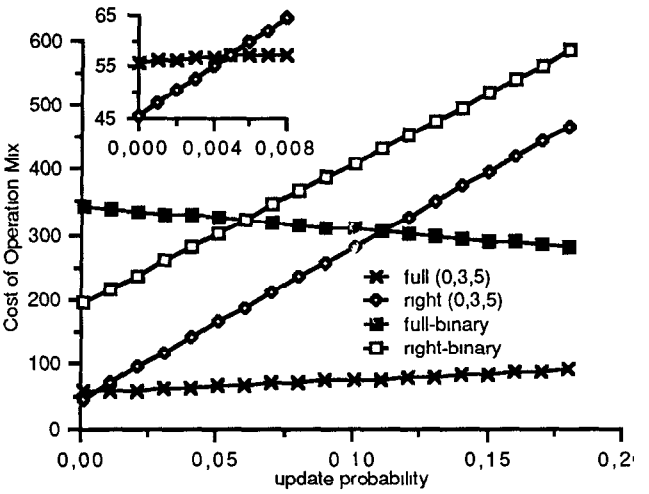

Figure 7 Isolating Right-Complete and Full Extension

7 Conclusion and Future Work

In this work we have tackled a major problem in optimizing object-oriented DBMS the evaluation of path expressions We have described the framework for a whole class of optimization methods, which we call *access support relation* The primary idea is to materialize such path expressions and store them separate from the object (data) representation The access support relation concept subsumes and extends several previously published proposals for access support in object-oriented database processing

Access support relations provide the physical database designer with design choices in two dimensions

1 one can choose among four extensions of the access support relation (canonical, full, left-, and right-complete extension)

2 for a fixed extension one can choose among all possible decompositions of an access support relation

It is not possible, to provide a generally valid forecast for the optimal design choice this is highly application dependent Therefore, it is essential that a complete analytical cost model has been developed which takes as input the application-specific parameters, such as number of objects, object size, fan-out, number of not-NULL attributes, etc Based on the application characteristics the analytical model can be used to compute for all (feasible) design choices the expected costs (based on secondary page accesses) of pre-determined database usage profiles, i e , envisaged operation mixes From this, the best suited access support relation extension and decomposition can be selected

From our cost evaluations for a few (sometimes contrived) application profiles it follows that an object oriented database system that allows associative access should provide the full range of options along both dimensions extensions and decompositions

The cost model is fully implemented Presently, it is being used to validate the access support relation concept So far, we have used the cost model to determine operation costs for some application characteristics that we deemed typical as non-standard database applications However, in a "real" database application one should periodically verify that the once envisioned usage profile actually remains valid under operation Therefore, the cost model is intended to be integrated into our object-oriented DBMS in order to verify a given physical database design, or even to automate the task of physical database design Thus, for a recorded database usage pattern the system could (semi-) automatically adjust the physical database design

So far, the access support relation manager has been implemented, we are currently working on the query optimizer that transforms queries with path expressions in order to take full advantage of any existing access support relations As much of the query evaluation should be performed using the access support relations, rather than searching in the stored object representation For this purpose we are currently developing a rule-based query optimizer [5]

Acknowledgements

Peter Lockemann and Klaus Radermacher read a preliminary draft of this paper and gave valuable comments Matthias Zimmermann helped to create the graphics in this paper

References

[1] E Bertino and W Kim Indexing techniques for queries on nested objects *IEEE Trans Knowledge and Data Engineering*, 1(2) 196–214, Jun 1989

[2] M J Carey, D J DeWitt, and S L Vandenberg A data model and query language for EXODUS In *Proc of the ACM SIGMOD Conf on Management of Data*, pages 413–423, Chicago, Il , Jun 1988

[3] G Copeland and S Khoshafian A decomposition storage model In *Proc of the ACM SIGMOD Conf on Management of Data*, pages 268–279, Austin, TX, May 1985

[4] A Kemper and G Moerkotte *Access Support in Object Bases* Internal Report 17/89, Fakultat für Informatik, Universitat Karlsruhe, D-7500 Karlsruhe, Oct 1989

[5] A Kemper and G Moerkotte Advanced query optimization in object bases using access support relations Manuscript (submitted for publication), 1990

[6] W Kim, K C Kim, and A Dale Indexing techniques for object-oriented databases In W Kim and F H Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394, Addison Wesley, Reading, MA, 1989

[7] D Maier and J Stein Indexing in an object-oriented DBMS In K R Dittrich and U Dayal, editors, *Proc IEEE Intl Workshop on Object-Oriented Database Systems, Asilomar, Pacific Grove, CA*, pages 171–182, IEEE Computer Society Press, Sep 1986

[8] T K Sellis Intelligent caching and indexing techniques for relational database systems *Information Systems*, 13(2) 175–186, 1988

[9] E J Shekita and M J Carey Performance enhancement through replication in an object-oriented DBMS In *Proc of the ACM SIGMOD Conf on Management of Data*, pages 325–336, Portland, OR, May 1989

[10] M Stonebraker, J Anton, and E Hanson Extending a database system with procedures *ACM Trans Database Systems*, 12(3) 350–376, Sep 1987

[11] P Valduriez Join indices *ACM Trans Database Syst* , 12(2) 218–246, Jun 87

[12] P Valduriez, S Khoshafian, and G Copeland Implementation techniques of complex objects In *Proc of The Conf on Very Large Data Bases (VLDB)*, pages 101–110, Kyoto, Japan, Aug 1986

[13] S B Yao Approximating block accesses in database organizations *Communications of the ACM*, 20(4), Apr 77