Information Systems Vol. 17, No. 2, pp. 117–145, 1992
Printed in Great Britain. All rights reserved

0306-4379/92 $5.00 + 0.00
Copyright © 1992 Pergamon Press Ltd

ACCESS SUPPORT RELATIONS: AN INDEXING METHOD FOR OBJECT BASES†

ALFONS KEMPER[1] and GUIDO MOERKOTTE[2]
[1]Lehrstuhl für Informatik III, RWTH Aachen, D-5100 Aachen, Fed. Rep. Germany
[2]Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Fed. Rep. Germany

(Received 13 October 1989; in revised form 9 December 1991)

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 separate structures (dissociated from the object representation) to redundantly store those 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 and decompositions 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 performance analysis of access support relations is developed. This analytical cost model is, in particular, used to determine the best access support relation extension and decomposition with respect to specific database configuration and usage characteristics.

Key words: Indexing, access methods, access cost model, query optimization, object-oriented database systems

1. INTRODUCTION

Record-oriented database systems, e.g. those based on the pure relational or the CODASYL network model, are widely believed to be inappropriate for engineering applications. There is a variety of reasons for this assessment: no explicit support of behavior, data segmentation due to normalization, lacking support of molecular aggregation and generalization, etc.

Object-oriented database systems constitute a promising approach towards supporting technical application domains. Several object-oriented data models have been developed over the last few years. However, these systems are still not adequately optimized: they still have problems to keep up with the performance achieved by, for example, 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. Access support relations are not intended for optimizing the computation-intensive modification of, for example, CAD objects in main-memory. Rather, access support relations are geared towards support of associative search for objects on secondary storage. Despite the many computation-intensive engineering applications, such as CAD, our experience with mechanical engineering applications (e.g. Ref. [2]), suggests that there are numerous applications where the efficient retrieval of objects from secondary memory is essential. Examples include:

- *construction*: in the construction phase engineers frequently search for similar artifacts that have been designed in the past. Only the efficient retrieval of such objects from the database enables the engineers to exploit past construction work.

†This is a revised and extended version of "Access support in object bases" [1]

- *logistics control*: in logistics control applications it is often necessary to retrieve object descriptions of artifacts that were stored a long time ago. Surely, these objects reside on secondary storage and have to be retrieved into main memory for modification.
- *CAD*: even in computation-intensive CAD applications it is often necessary to load a particular subset of objects from secondary storage into the main memory. Indexing structures should support the localization of these objects in order to expedite the loading phase.
- *maintenance*: during machine maintenance it may be essential to retrieve stored descriptions of particular subcomponents of a complex machinery for e.g. "trouble shooting."

In the context of associative search one of the most performance-critical operations in relational databases 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 [3] building on links [4] 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. Ref. [5]) call this kind of object traversal also *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. In this respect access support relations constitute materializations of frequently traversed reference chains. In addition we have developed techniques that allow the materialization of function results in an object base [6].

The access support relations described in this paper constitute a generalization of two relational techniques: the *links* developed by Härder [4] and the binary *join indices* proposed by Valduriez [3]. Rather than relating only two relations (or object types) our technique allows to support access paths ranging over many types. Our indexing technique subsumes and extends several previously proposed strategies for access optimization 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 [8] are contained as a special case in our framework. Keßler and Dadam [9] reports on an indexing technique for hierarchical object structures, i.e. nested relations, which is related to our access support relations.

Our technique differs in three major aspects from the 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 particular application characteristics.

Also the (separate) replication of object values as proposed for the Extra object model [10] and for the PostGres model [11, 12] are subsumed by our technique.

The remainder of this paper is organized as follows. Section 2 introduces the Generic Object Model (GOM) [13]), which serves as the research vehicle for this work, and some simplified application examples to highlight the requirements on access support in object bases. Then, in Section 3 the access support relations are formally defined. In Section 4 we begin the analytical evaluation of our indexing technique by comparing the cardinalities of various representations of access support relations. Section 5 is devoted to estimating the performance enhancement in query processing on the basis of secondary page accesses. Section 6 addresses the maintenance of access support relations due to object updates. In each of the Sections 4–6 we illustrate the analytical cost model by some comparative results for characteristic application profiles. In Section 7 some sample evaluations of typical application mixes, i.e. database usage profiles consisting of updates and queries, are presented. Section 8 concludes this paper.

2. GOM AND ITS DECLARATIVE QUERY LANGUAGE

This research is based on an object-oriented model that unites the most salient features of many recently proposed models in one coherent framework. In this respect, the objective of GOM [13] can be seen as providing a syntactical framework of the essential object-oriented features identified in the "Manifesto" [14]—albeit the GOM model was developed much earlier. Independently —but with the same intention—Zdonik and Maier developed the so-called Reference Model in [15]. The features that GOM provides are relatively *generic* (and basic) such that the results derived for this particular data model can be applied to a variety of other object-oriented models. A list of object-oriented models, to which our proposed indexing method can be applied in a straightforward manner is compiled as follows: GemStone [16], O_2 [17], Orion [18], ObjectStore [19], EXTRA [5] and the Reference Model [15].

2.1. *Main concepts of GOM*

GOM provides the following object-oriented concepts:

object identity Each object instance has an identity that remains invariant throughout its lifetime. The object identifier 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.

values GOM has a built-in collection of elementary (value) types, such as *char*, *string*, *integer*, etc. Instances of these types do not possess an identity, rather their respective value serves as their identity.

type constructors The most basic type constructor is the *tuple* constructor—denoted as [ ]—which aggregates differently typed attributes to one object. In addition, GOM has the two built-in collection type constructors *set*, denoted as { }, and *list*, denoted as ⟨ ⟩.

subtyping A tuple-structured type *t* may be defined as the subtype of one other tuple-structured type *s* which means that *t* inherits all attributes of the supertype *s*.

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

instantiation Types can be instantiated to render a new object instance. All internal components of a newly instantiated tuple object are initially set to NULL, the undefined value. Set- and list-instances are initially set to the empty set or list.

uni-directional references Even though this is actually an implementation issue, because of its relevance to our indexing scheme we want to mention here, that GOM—like almost all other objects models (especially the ones mentioned above)—maintains references from one object to another only uni-directionally.

2.2. *Type definitions*

If s, t, $t_1, \ldots, t_n$ are types, and $a_1, \ldots, a_n$ are pairwise distinct attribute names then

$$\textbf{type } t \textbf{ supertype } s \textbf{ is}$$
$$[a_1 : t_1, \ldots, a_n : t_n]$$

is a tuple structured type definition.† In this case, the **supertype** *s*—which is optionally defined —must itself be a tuple-structured type. The type *t* is called a (direct) *subtype* of *s* and inherits all attributes of *s* (including those that *s* inherited from its supertype, if any).

Aside from tuple-structured types GOM provides built-in support for two *collection types*: *sets* and *lists* which are defined as follows:

$$\begin{array}{ll} \textbf{type } t \textbf{ is} & \textbf{type } t \textbf{ is} \\ \{s\} & \langle s \rangle \end{array}$$

†We presented only the structural parts of our object type definitions; of course, there are type-specific operations that can be defined by the database user.

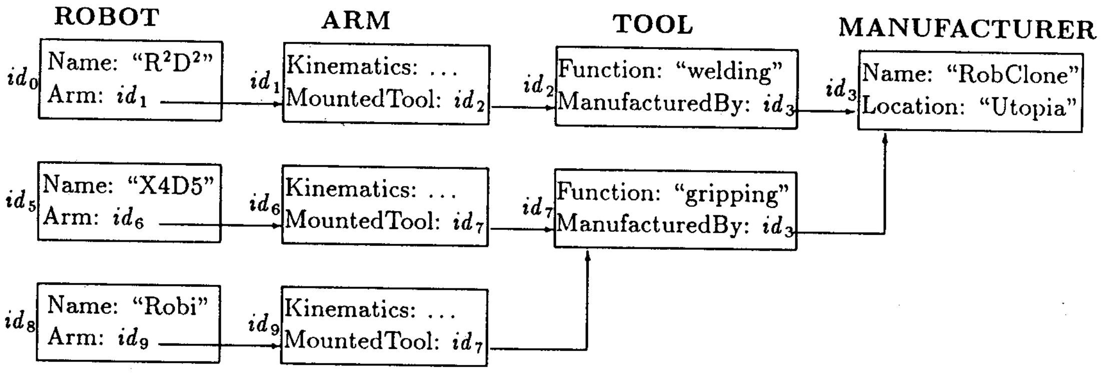

ROBOT
ARM
TOOL
MANUFACTURER
id0 Name: "R²D²" Arm: id1
id1 Kinematics: ... MountedTool: id2
id2 Function: "welding" ManufacturedBy: id3
id3 Name: "RobClone" Location: "Utopia"
id5 Name: "X4D5" Arm: id6
id6 Kinematics: ... MountedTool: id7
id7 Function: "gripping" ManufacturedBy: id3
id8 Name: "Robi" Arm: id9
id9 Kinematics: ... MountedTool: id7

Fig. 1. Database extension with linear paths.

Here, s has to be a tuple-structured (i.e. complex) object type or an atomic type. At the present we do not deal with nested set types with respect to our indexing structures. Since the access support on ordered collections, i.e. lists, is analogous to sets we will not elaborate on list-structured types in the remainder of this paper.

2.3. *(Engineering) example application*

Let us first sketch an engineering application that heavily utilizes tuple-structured type s: modeling robots. The following schema constitutes an outline of such a—drastically simplified—robot model:

type ROBOT **is** [Name: STRING, Arm: ARM];
type ARM **is** [Kinematics: . . . , MountedTool: TOOL];
type TOOL **is** [Function: STRING, ManufacturedBy: MANUFACTURER];
type MANUFACTURER **is** [Name: STRING, Location: STRING];

As can be deduced from the schema, a *ROBOT* has a *Name* and an *Arm* attribute, the latter itself referring to a composite object of type *ARM*. An *ARM* instance is described by its *Kinematics*† and a *MountedTool*, an attribute referring to an instance of type *TOOL*. A *TOOL* is modeled by a string-valued attribute *Function* and the attribute *ManufacturedBy* which associates a *MANUFACTURER* object, which itself contains atomic attributes *Name* and *Location*, with the *TOOL* instance.

An extension of such a schema for just three *ROBOT* instances identified by id_0, id_5 and id_8 is graphically depicted in Fig. 1. An object instance is a triple (id, v, t) where id denotes the system-wide unique object identifier, v the object representation (everything inside the box in our graphical sketch), and t the type of the object. As indicated in Fig. 1 references are *uni-directional*, i.e. they are maintained in one direction only. As mentioned above, this conforms to (almost) all proposed object models

2.4. *The query language*

For our object model we developed a QUEL-like [21] query language along the lines of the EXCESS object query language that was designed as the declarative query language for the EXTRA object model [5]. Currently, our optimizer [22] supports only the declarative QUEL-like query language. In the future we intend to support other declarative query languages as well as the optimization of procedurally specified database access.

Let x_i be variables, T_i set typed expressions or type names, and S a selection predicate. Then, a query has the following form:

range $x_1 : T_1, \ldots, x_n : T_n$
retrieve x_i
where $S(x_i, \ldots, x_n)$

†This complex attribute is not further elaborated here. For more details see Ref. [20].

The selection predicate S in variables $x_1, \ldots, x_n$ may consist of path expressions, comparison operators, set operators, boolean connectors and may also contain a (nested) **retrieve** statement. Our current implementation of the GOM query language facilitates single-target queries only.

A query in such an object-oriented system would retrieve objects on the basis of attribute values of others associated objects along a reference chain, i.e. a path expression. A typical example is:

Example 2.1. Find the *Robots* which use a *Tool* manufactured in "*Utopia*". This query can be formulated in our QUEL-like language as follows:

```
range     r: ROBOT
retrieve  r
where     r.Arm.MountedTool.ManufacturedBy.Location = "Utopia"
```

The path expression used in this example can be outlined as follows (the underbraces indicate the type of the respective sub-path):

$$P \equiv \underbrace{\underbrace{\underbrace{ROBOT.MountedTool}_{\text{TOOL}}.ManufacturedBy}_{\text{MANUFACTURER}}.Location}_{\text{STRING}}$$

The cost to evaluate this query can be estimated as (under the assumption that each *TOOL* is mounted on some *ROBOT* and each *MANUFACTURER* produces some *TOOL*):

$$\#(ROBOT) + \#(TOOL) + \#(MANUFACTURER)$$

where $\#(t)$ denotes the cardinality of the extension of type t. This cost is induced because—no matter how good the query evaluation algorithm performs—every instance of the respective type has to be visited at least once. ◇

2.5. *General paths* (*containing collection-valued attributes*)

Note that a linear path contains only attributes referring to a single object. Single-object-valued attrributes 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 database schema for modeling a *COMPANY* composed of a set of *DIVISION*s. Each *DIVISION Manufactures* a set of *PRODUCT*s, which themselves are composed of *BASE_PART*s.

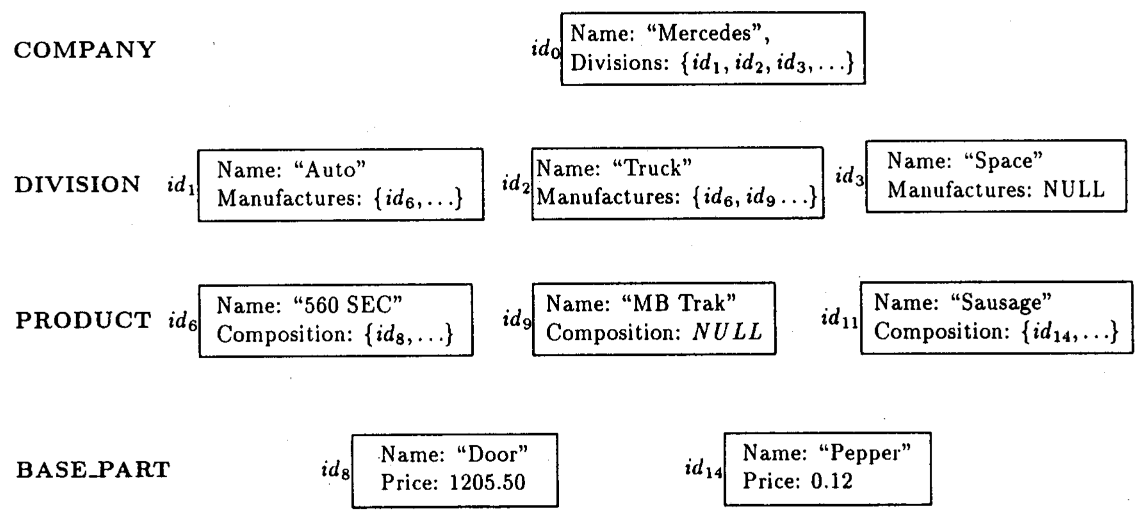

COMPANY
id_0 Name: "Mercedes", Divisions: $\{id_1, id_2, id_3, \ldots\}$
DIVISION
id_1 Name: "Auto" Manufactures: $\{id_6, \ldots\}$
id_2 Name: "Truck" Manufactures: $\{id_6, id_9 \ldots\}$
id_3 Name: "Space" Manufactures: NULL
PRODUCT
id_6 Name: "560 SEC" Composition: $\{id_8, \ldots\}$
id_9 Name: "MB Trak" Composition: $NULL$
id_{11} Name: "Sausage" Composition: $\{id_{14}, \ldots\}$
BASE_PART
id_8 Name: "Door" Price: 1205.50
id_{14} Name: "Pepper" Price: 0.12

Fig. 2. Database extension with non-linear paths.

The schema is outlined below:

```
type COMPANY is [Name: STRING, Divisions: {DIVISION}];
type DIVISION is [Name: STRING, Manufactures: {PRODUCT}];
type PRODUCT is [Name: STRING, Composition: {BASE_PART}];
type BASE_PART is [Name: STRING, Price: DECIMAL];
```

A sample extension of this schema is presented in Fig. 2.

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

Example 2.2. Which *DIVISION* of Mercedes uses a *BASE_PART* named "Door"?

```
range    c: COMPANY, d: DIVISION
retrieve d
where    c.Name = "Mercedes" and
         d in c.Divisions and
         "Door" in d.Manufactures.Composition.Name
```

Example 2.3. Retrieve all the *BASE_PARTs* used by Mercedes.

```
range    c: COMPANY, b: BASE_PART
retrieve b
where    c.Name = "Mercedes" and
         b in c.Divisions.Manufactures.Composition
```

3. ACCESS SUPPORT RELATIONS

3.1. Auxiliary definitions

A path expression has the form

$$o.A_1.\cdots.A_n$$

where o is a tuple structured object containing the attribute A_1 and $o.A_1.\cdots.A_i$ refers to an object or a set of objects, all of which have an attribute A_{i+1}. Thus, the result of the path expression is the set R_n, which is recursively defined as follows:

$$R_0 := \{o\}$$

$$R_i := \bigcup_{v \in R_{i-1}} v.A_i \quad \text{for } 1 \leqslant i \leqslant n.$$

Thus, R_n is a set of OIDs of objects of type t_n or a set of atomic values of type t_n if t_n is an atomic data type, such as *INT*.

It is also possible that the path expression originates in a collection C of tuple-structured objects, i.e. $C.A_1.\cdots.A_n$. Then the definition of the set R_0 has to be revised to: $R_0 := C$.

Formally, a path expression or attribute chain is defined as follows:

Definition 3.1. (Path expression) *Let $t_0, \ldots, t_n$ be (not necessarily distinct) types. A path expression on t_0 is an expression $t_0.A_1.\cdots.A_n$ iff for each $1 \leqslant i \leqslant n$ one of the following conditions holds:*

- *The type t_{i-1} is defined as* **type** t_{i-1} **is** $[\ldots, A_i : t_i, \ldots]$, *i.e. t_{i-1} is a tuple with an attribute A_i of type t_i.*†
- *The type t_{i-1} is defined as* **type** t_{i-1} **is** $[\ldots, A_i : t_i', \ldots]$ *and the type t_i' is defined as* **type** t_i' **is** $\{t_i\}$, *i.e. t_i' is a set type whose elements are instances of t_i. In this case we speak of a set occurrence at A_i in the path $t_0.A_1.\cdots.A_n$.*

For simplicity of the presentation we assumed that the involved types are not being defined as a subtype of some other type. This—of course—is generally possible; it would only make the definition a bit more complex to read.

†This means that the attribute A_i can be associated with objects of type t_i or any subtype thereof.

The second part of the definition is useful to support access paths through sets.† If it does not apply for a given path the path is called *linear*. A path expression that contains at least one set-valued attribute is called *set-valued*.

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.

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. (Access support relation [ASR]) *Let $t_0, \ldots, t_n$ be types, $t_0.A_1.\cdots.A_n$ be a path expression. Then the access support relation $[\![t_0.A_1.\cdots.A_2]\!]$ is of arity $n+1$ and has the following form:*

$$[\![t_0.A_1.\cdots.A_n]\!] : [S_0, \ldots, S_n]$$

The domain of the attribute S_i is the set of identifiers (OIDs) of objects of type t_i for $(0 \leqslant i \leqslant n)$. If t_n is an atomic type then the domain of S_n is t_n, i.e. values are directly stored in the access support relation.

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

Definition 3.3. (Temporary binary relations) *For each i $(1 \leqslant i \leqslant n)$—that is, for each attribute in the path expression—we construct the temporary binary relation $[\![t_{i-1}.A_i]\!]$. The relation $[\![t_{i-1}.A_i]\!]$ contains the tuples $(id(o_{i-1}), id(o_i))$ for every object o_{i-1} of type t_{i-1} and o_i of type t_i such that*

- $o_{i-1}.A_i = o_i$ *if A_i is a single-valued attribute.*
- $o_i \in o_{i-1}.A_i$ *if A_i is a set-valued attribute.*

If t_n is an atomic type then $id\,(o_n)$ corresponds to the value $o_{n-1}.A_n$. Note, however, that only the last type t_n in a path expression can possibly be an atomic type.

Example 3.4. Let us re-consider the path expression of our schema *COMPANY* (again, we indicate the types of the subpaths by the underbraces):

$$P \equiv \underbrace{\underbrace{\underbrace{\underbrace{COMPANY.Divisions}_{\text{DIVISION}}.Manufactures}_{\text{PRODUCT}}.Composition}_{\text{BASE\_PART}}.Name}_{\text{STRING}}$$

For this path expression the temporary binary relations have extensions as shown in Table 1.

3.2. Extensions of access support relations

Let us now introduce different possible extensions of the ASR $[\![t_0.A_1.\cdots.A_n]\!]$. We distinguish four extensions:

1. The *canonical* extension, denoted $[\![t_0.A_1.\cdots.A_n]\!]_{can}$ contains only information about complete paths, i.e. paths originating in t_0 and leading (all the way) 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 $[\![t_0.A_1.\cdots.A_n]\!]_{left}$ 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 $[\![t_0.A_1.\cdots.A_n]\!]_{right}$, 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 $[\![t_0.A_1.\cdots.A_n]\!]_{full}$ contains all partial paths, even if they do not originate in t_0 or do end in a *NULL*.

†Note, however, that we do not permit powersets.

Table 1

[COMPANY.Divisions]	
$OID_{COMPANY}$	$OID_{DIVISION}$
id_0	id_1
id_0	id_2
id_0	id_3
...	...

[DIVISION.Manufactures]	
$OID_{DIVISION}$	$OID_{PRODUCT}$
id_1	id_6
id_2	id_6
id_2	id_9
...	...

[PRODUCT.Composition]	
$OID_{PRODUCT}$	OID_{BASE_PART}
id_6	id_8
id_{11}	id_{14}
...	...

[BASE_PART.Name]	
OID_{BASE_PART}	*STRING*
id_8	"Door"
id_{14}	"Pepper"
...	...

Definition 3.5. (Extensions) *Let* $\bowtie$ ($⟗$, $⟕$, $⟖$) *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*:

$$[\![t_0.A_1.\cdots.A_n]\!]_{can}:=[\![t_0.A_1]\!]\bowtie\cdots\bowtie[\![t_{n-1}.A_n]\!]$$

$$[\![t_0.A_1.\cdots.A_n]\!]_{full}:=[\![t_0.A_1]\!]⟗\cdots⟗[\![t_{n-1}.A_n]\!]$$

$$[\![t_0.A_1.\cdots.A_n]\!]_{left}:=(\cdots([\![t_0.A_1]\!]⟕[\![t_1.A_2]\!]\cdots⟕[\![t_{n-1}.A_n]\!]$$

$$[\![t_0.A_1.\cdots.A_n]\!]_{right}:=([\![t_0.A_1]\!]⟖\cdots([\![t_{n-2}.A_{n-1}]\!]⟖[\![t_{n-1}.A_n]\!])\cdots) \qquad \square$$

Example 3.6. For the path of Example 3.4 the full extension, whch is denoted as

$$[\![COMPANY.Divisions.Manufactures.Composition.Name]\!]_{full}$$

looks as shown in Table 2. This extension contains all paths and subpaths corresponding to the underlying path expression. The first two tuples actually constitute complete path which would be present in the canonical extension as well; however the last three paths would be omitted in the canonical extension. In the left-complete extension the first four tuples would be present, whereas the last one would be omitted since it does not originate in *COMPANY*. Analogously, the right-complete extension would contain the first two and the last tuple and omit the third and fourth tuple since they do not "go all the way through" to a *STRING* representing the *Name* of some *BASE_PART*.

The next definition states under what conditions an existing access support relation can be utilized to evaluate a path expression that originates in an object (or a set of objects) of type s.

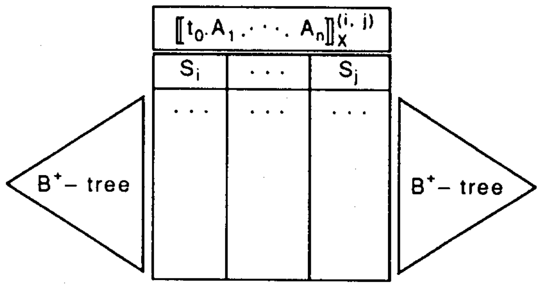
Definition 3.7. (Applicability) *An access support relation* $[\![t_0.A_1.\cdots.A_n]\!]_X$ *under extension* X *is applicable for a path* $s.A_i.\cdots.A_j$ *with* $s \leqslant t_{i-1}$ *under the followihg condition, depending on the extension* X:

$$Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j)=\begin{cases}X=full \wedge 1\leqslant i\leqslant j\leqslant n\\ X=left \wedge 1=i\leqslant j\leqslant n\\ X=right \wedge 1\leqslant i\leqslant j=n\\ X=can \wedge 1=i\leqslant j=n\end{cases}$$

Here $s \leqslant t_{i-1}$ *denotes that type* s *has to be identical to type* t_{i-1} *or a subtype thereof.*

Table 2

[COMPANY.Divisions.Manufactures.Composition.Name]$_{full}$				
$OID_{COMPANY}$	$OID_{DIVISION}$	$OID_{PRODUCT}$	OID_{BASE_PART}	*STRING*
id_0	id_1	id_6	id_8	"Door"
id_0	id_2	id_6	id_8	"Door"
id_0	id_2	id_9	—	—
id_0	id_3	—	—	—
—	—	id_{11}	id_{14}	"Pepper"
...	...	...	...	...



Scheme 1

3.3. Decomposition and storage structure of ASRs

Aside from extensions, we also facilitate the decomposition of access support relations. The following formally defines valid decompositions:

Definition 3.8. (Decomposition) *Let $[\![t_0.A_1.\cdots.A_n]\!]_X$ be an $(n+1)$-ary access support relation with attributes $S_0,\ldots,S_n$ under extension X, for $X \in \{can, full, left, right\}$. Then the relations*

$$\begin{array}{ll} [\![t_0.A_1.\cdots.A_n]\!]_X^{(0,i_1)}:[S_0,\ldots,S_{i_1}] & \textit{for } 0<i_1\leqslant n \\ [\![r_0.A_1.\cdots.A_n]\!]_X^{(i_1,i_2)}:[S_{i_1},\ldots,S_{i_2}] & \textit{for } i_1<i_2\leqslant n \\ \ldots & \\ [\![t_0.A_l.\cdots.A_n]\!]_X^{(i_k,n)}:[S_{i_k},\ldots,S_n] & \textit{for } i_k<n \end{array}$$

are called a decomposition of $[\![t_0.A_1.\cdots.A_n]\!]_X$. The relations $[\![t_0.A_1.\cdots.A_n]\!]_X^{(i_j,i_j+1)}$ are called partitions for $(0\leqslant j\leqslant k)$.† *They are materialized by projecting the corresponding attributes of $[\![t_0.A_1.\cdots.A_n]\!]_X$:*

$$[\![t_0.A_1.\cdots.A_n]\!]_X^{(i_j,i_j+1)}:=\pi_{(S_{i_j},S_{i_j+1},\ldots,S_{i_j+1})}([\![t_0,A_1.\cdots.A_n]\!]_X)$$

If every partition is a binary relation the decomposition is called binary. The above decomposition is denoted by $(0,i_1,i_2,\ldots,i_k,n)$.

Theorem 3.9. *Every decomposition of an access support relation is lossless.*

The proof of this theorem is straightforward since the decomposition is materialized along multi-valued dependencies. For any partition with $(0\leqslant i<q<j\leqslant n)$

$$[\![t_0,A_1.\cdots.A_n]\!]_X^{(i,j)}:[S_i,\ldots,S_q,\ldots,S_j]$$

the following multi-valued dependency holds:

$$\{S_q\}\twoheadrightarrow\{S_i,\ldots,S_{q-1}\}$$

Therefore, the decomposition into

$$[\![t_0.A_1.\cdots.A_n]\!]_X^{(i,q)}:[S_i,\ldots,S_q] \quad \text{and} \quad [\![t_0.A_1.\cdots.A_n]\!]_X^{(q,j)}:[S_q,\ldots,S_j]$$

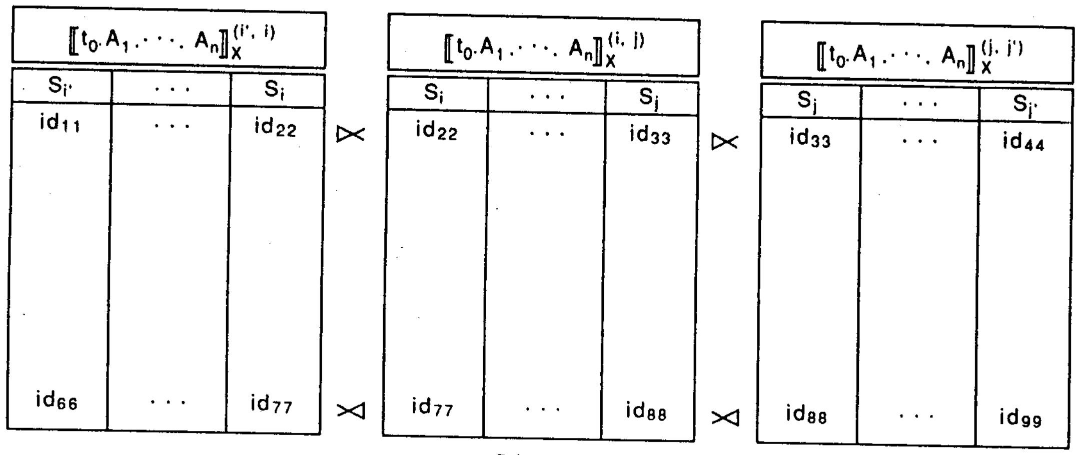
is lossless. Repeated application of this argument yields the theorem. □

The storage structure of access support relations is borrowed from the binary join index proposal by Valduirez [3]. Each partition is redundantly stored in two B^+-trees: the first being clustered (keyed) on the left-most attribute and the second being clustered on the right-most attribute. Graphically, this storage scheme is visualized as shown in Scheme 1.

We will call the left B^+-tree the "forward clustered" tree, and—analogously—the right one the "backward clustered" tree. This storage scheme is well suited for traversing paths from left-to-right (forward) as well as from right-to-left (backward) within the access support relations even if they span over several partitions. Again, let us graphically visualize the situation, Scheme 2.

Scheme 2 illustrates the virtues of the redundant storage model for ASRs. It directly supports the lookup of the OID id_{44} in the right partition $[\![t_0,A_1.\cdots.A_n]\!]_X^{(j,j')}$ (via its B^+-tree clustered on the right attrribute) and the evaluation of the semi-join across the three partitions from right to

†For notational convenience let $i_0:=0$ and $i_{k+1}:=n$.



Scheme 2

to retrieve the object identifier id_{11} in the left partition $[\![t_0.A_1.\cdots.A_n]\!]_X^{(i',i)}$. Thus, the backward traversal constitutes a "right-to-left" semi-join across partitions:

$$\pi_{S_{i'}}([\![t_0.A_1.\cdots.A_n]\!]_X^{(i',i)} \ltimes ([\![t_0.A_1.\cdots.A_n]\!]_X^{(i,j)} \ltimes \sigma_{S_{j'}=id_{44}}([\![t_0.A_1.\cdots.A_n]\!]_X^{(j,j')}))).$$

Analogously, the "forward clustered" B^+-tree supports the semi-join from left to right, such that, for instance, the object identifier(s) of object(s) that are associated with id_{66} are efficiently retrievable —in our case this is the OID id_{99} from the right partition. This corresponds to the "left-to-right" semi-join across partitions:

$$\pi_{S_{j'}}(((\sigma_{S_{i'}=id_{66}}(t_0.A_1.\cdots.A_n]\!]_X^{(i',i)})) \rtimes [\![t_0.A_1.\cdots.A_n]\!]_X^{(i,j)}) \rtimes [\![t_0.A_1.\cdots.A_n]\!]_X^{(j,j')}).$$

Note, that a binary decomposed access support relation in *full* extension is similar to conventional indexes in relational systems. Except, in our storage model of ASRs the index is clustered in both directions—in relational systems an index is only used in one direction, i.e. it relates attribute values to tuples. Binary partitions of other extensions, i.e. *can*, *right* and *left*, of an access support relation, however, contain only selective information—thereby limiting the search space for query evaluation.

4. ANALYTICAL COST MODEL: CARDINALITY OF ACCESS SUPPORT RELATIONS

The different decompositions and extensions provide the database designer a large spectrum of design choices to tune the access support relations for particular application characteristics. The remainder of this paper is devoted to the development of an analytical cost model that supports the task of finding the optimal extension and decomposition of an access support relation for a given database usage profile.

In this section we start the analysis of our indexing scheme based on an analytical cost model. Ultimately, the cost model is used to derive the best physical database design, i.e. to find the best extension and decomposition of the ASR for a given path expression according to the operation mix. First we have to design a model in which the object base extension, in which we consider a path expression, can be described. Then we analyze the storage costs for access support relations in various extensions and decompositions.

4.1. *Preliminaries*

Before deriving formulas for estimating the sizes of the relations we introduce some parameters that model the characteristics of an application. For the examples presented in the paper we can divide the parameters into two groups. The first group comprises the parameters specifying the application characteristics. These are: n which denotes the path length; c_i the total number of objects of type t_i; d_i the number of objects which have non-*NULL* references for the attribute A_{i+1} to objects of type t_{i+1}; f_i (fan-out) is the average number of references emanating from attribute A_{i+1}

application-specific parameters		
parameter	semantics	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*	
f_i	the number of references emanating on the average from the attribute A_{i+1} of an object o_i of type t_i whose A_{i+1} attribute is not *NULL*	
e_i	the number of objects of type t_i which are referenced by an object of type t_{i-1}	$\min(c_i, d_{i-1} * f_{i-1})$
$size_i$	average size of objects of type t_i	
system-specific parameters		
PageSize	net size of pages	*PageSize* = 4056
OIDsize	size of object identifiers	*OIDsize* = 8
PPsize	size of page pointer	*PPsize* = 4

Fig. 3. System and application parameters.

of an object of type t_i; e_i is the number of objects hit by a reference from an attribute A_i of an object of type t_{i-1}; and the parameter $size_i$, denoting the average size of objects of type t_i. The second group consists of fixed system-parameters. These are the page size (PageSize); the number of bytes necessary to store an object identifier (*OIDsize*); and the size of a page pointer, i.e. a reference to a page (*PPsize*). All parameters are summarized in Fig. 3.

4.1.1. Some derived quantities. The average number of objects of type t_i that reference the same object in t_{i+1} is denoted as $shar_i$. A uniform distribution of references from objects in t_i to objects in t_{i+1} is assumed.

In this case $shar_i$ is derived as:

$$shar_i = \frac{d_i f_i}{e_{i+1}}. \tag{1}$$

The parameter $spread_i$ denotes the relation between the number of defined objects of type t_i and the referenced objects of type t_{i+1}:

$$spread_i = \frac{d_i}{e_{i+1}}. \tag{2}$$

The value ref_i denotes the number of references emanating from objects of type t_i:

$$ref_i = d_i f_i. \tag{3}$$

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

$$P_{A_i} = \frac{d_i}{c_i}. \tag{4}$$

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:

$$P_{H_i} = \frac{e_i}{c_i}. \tag{5}$$

Let us now derive the probability that, for some object o_i of type t_i none of the f_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 number is derived as a fraction of two binomial coefficients (see Ref. [23]):

$$\frac{\binom{e_{i+1}-1}{f_i}}{\binom{e_{i+1}}{f_i}} = \frac{e_{i+1}-f_i}{e_{i+1}} = 1 - \frac{f_i}{e_{i+1}}. \tag{6}$$

Note that

$$1-\frac{f_i}{e_{i+1}}=1-\frac{shar_i}{d_i}=\frac{d_i-shar_i}{d_i}=\binom{d_i-1}{shar_i}\Big/\binom{d_i}{shar_i}. \tag{7}$$

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

$$\binom{d_i-k}{shar_i}\Big/\binom{d_i}{shar_i}. \tag{8}$$

For $0 \leqslant i < j \leqslant n$ 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:

$$RefBy(i,j,k)=\begin{cases} e_j\left[\binom{k*fac}{shar_i}\Big/\binom{d_i}{shar_i}\right] & \text{if } j=i+1 \\ e_j\left[\binom{RefBy(i,j-1,k)P_{A_{j-1}}}{shar_{j-1}}\Big/\binom{d_{j-1}}{shar_{j-1}}\right] & \text{else} \end{cases} \tag{9}$$

where $fac = P_{A_i}$ if the k elements are uniformly distributed among the elements of t_i and $fac = 1$ if the k elements are uniformly distributed among these elements having a non-$NULL$ A_{j+1} attribute.

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

$$P_{RefBy}(i,j)=\begin{cases} 0 & \text{if } i=j \\ \dfrac{RefBy(i,j,d_i)}{c_j} & \text{else} \end{cases} \tag{10}$$

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 $0 \leqslant i < j \leqslant n$. This value can be approximated as:

$$Ref(i,j,k)=\begin{cases} d_i\left[\binom{k*fac}{f_i}\Big/\binom{e_{i+1}}{f_i}\right] & \text{if } j=i+1 \\ d_i\left[\binom{Ref(i+1,j,k)P_{H_{i+1}}}{f_i}\Big/\binom{e_{i+1}}{f_i}\right] & \text{else} \end{cases} \tag{11}$$

where $fac = P_{H_{i+1}}$ if the k elements are uniformly distributed among the elements of t_j and $fac = 1$ if the k elements are uniformly distributed among those elements referenced by t_{j-1}.

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

$$P_{Ref}(i,j)=\begin{cases} 0 & \text{if } i=j \\ \dfrac{Ref(i,j,e_j)}{c_i} & \text{else} \end{cases} \tag{12}$$

We will sometimes use the "two-parameter" versions of $RefBy$ and Ref, which are defined as follows:

$$RefBy(i,j) := RefBy(i,j,c_i) \tag{13}$$

$$Ref(i,j) := Ref(i,j,c_j). \tag{14}$$

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

$$path(i,j)=Ref(i,j,e_j)\prod_{l=i}^{j-1} f_l \tag{15}$$

or equivalently

$$path(i,j)=RefBy(i,j,d_i)\prod_{l=i}^{j-1} shar_l. \tag{16}$$

4.2. Analysis of cardinalities and storage costs of access support relations

We now present the cardinality results for the different extensions. The access support relation partitions $[\![t_0.A_1.\cdots.A_n]\!]_X^{(i,j)}$ which hold all the paths from t_i to t_j of the corresponding extension X, have the following cardinalities:

$$\#[\![t_0.A_1.\cdots.A_n]\!]_{can}^{(i,j)} = RefBy(0,i)P_{Ref(i,n)}\prod_{l=i}^{j-1} f_l \tag{17}$$

$$\#[\![t_0.A_1.\cdots.A_n]\!]_{left}^{(i,j)} = \sum_{k=1}^{j-i} c_i P_{RefBy}(0,i)P_{rb}(i,i+k+1)\prod_{p=i}^{i+k-1} f_p \tag{18}$$

$$\#[\![t_0.A_1.\cdots.A_n]\!]_{right}^{(i,j)} = \sum_{k=1}^{j-i} c_{j-k}P_{lb}(j-k-1,j-k)P_{Ref}(j-k,n)\prod_{p=j-k}^{j-1} f_p \tag{19}$$

$$\#[\![t_0.A_1.\cdots.A_n]\!]_{full}^{(i,j)} = \sum_{k=1}^{j-i}\sum_{l=i}^{j-k} c_l P_{lb}(l-1,l)P_{rb}(l,l+k+1)\prod_{p=l}^{l+k-1} f_p \tag{20}$$

where the subsequently defined probabilities have been used.

Let $P_{lb}(i,j)$ denote† the probability that a particular object of type t_j is not "hit" by any path emanating from some object in t_i for $0 \leqslant i < j \leqslant n$:

$$P_{lb}(i,j) = \begin{cases} 1 - P_{RefBy}(i,j) & 0 \leqslant i < j \leqslant n \\ 1 & \text{else} \end{cases}. \tag{21}$$

Let $P_{rb}(i,j)$ denote‡ the probability that an object of t_i has at least one path to some object of type t_{j-1} but no path to an object of type t_j for $0 \leqslant i < j \leqslant n$:

$$P_{rb}(i,j) = \begin{cases} P_{A_i}P'_{rb}(i,j) & 0 \leqslant i < j \leqslant n \\ 1 & \text{else} \end{cases} \tag{22}$$

with

$$P'_{rb}(i,j) = \begin{cases} (1 - P_{Ref(i,j-1)})P_{ref}(i,j-1) & 0 \leqslant i < j \\ 1 & \text{else} \end{cases}. \tag{23}$$

The approximate number of tuples per page in the access support relation $[\![t_0.A_1.\cdots.A_n]\!]_X^{(i,j)}$ is

$$atpp^{(i,j)} = \left\lfloor \frac{PageSize}{OIDsize(j-i+1)} \right\rfloor. \tag{24}$$

Note that this parameter is not dependent on the extension X.

The approximate number of pages needed to store the ASR $[\![t_0.A_1.\cdots.A_n]\!]_X^{(i,j)}$ is:

$$ap_X^{(i,j)} = \left\lceil \frac{\#[\![t_0.A_1.\cdots.A_n]\!]_X^{(i,j)}}{atpp^{(i,j)}} \right\rceil. \tag{25}$$

4.3. Example cardinalities of access support relations

Subsequently, we graphically demonstrate two results for application characteristics that we deem not untypical in the engineering domain. However, the reader should bear in mind that the size comparison of different access support relation extensions and decompositions does not permit any conclusions as to the performance of the respective physical design. The two results are merely included to give the reader some "feeling" about comparative storage costs.

4.3.1. Comparison between extensions and decompositions. In this experiment we want to compare different extensions and decompositions of the access support relation size for a fixed application characterization, which is listed in Table 3.

†*lb*: left-bound.
‡*rb*: right-bound.

Table 3

Application characteristics					
n	4				
Number of objects	c_0 1000	c_1 5000	c_2 10,000	c_3 50,000	c_4 100,000
Number of objects with defined A_{i+1} attribute	d_0 900	d_1 4000	d_2 8000	d_3 20,000	d_4 —
Fan-out	f_0 2	f_1 2	f_2 3	f_3 4	f_4 —

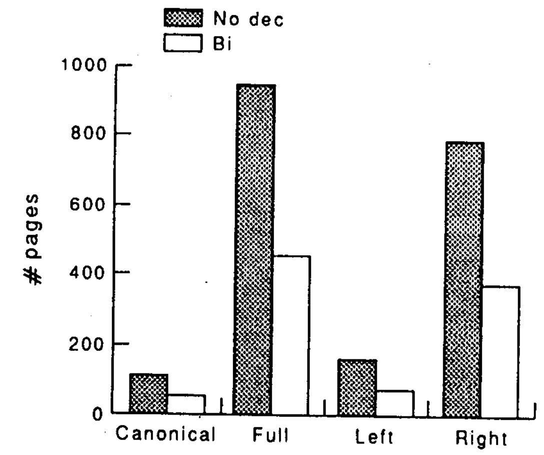


Fig. 4. Comparison of access support relation sizes.

The comparison of storage costs (for non-redundant representation) is graphically plotted in Fig. 4. In this example application there are few objects at the "left" side of the path which causes the canonical and the left-complete extensions to be drastically smaller than the right-complete and full extension. It can be seen that—for this application—the binary decomposition reduces storage costs by a factor of 2.

Table 4

Application characteristics					
n	4				
Number of objects	c_0 10,000	c_1 10,000	c_2 10,000	c_3 10,000	c_4 10,000
Number of objects with defined A_{i+1} attribute	d_0 $2500 \cdots 10^4$	d_1 $2500 \cdots 10^4$	d_2 $2500 \cdots 10^4$	d_3 $2500 \cdots 10^4$	d_4 —
Fan-out	f_0 2	f_1 2	f_2 2	f_3 2	f_4 —

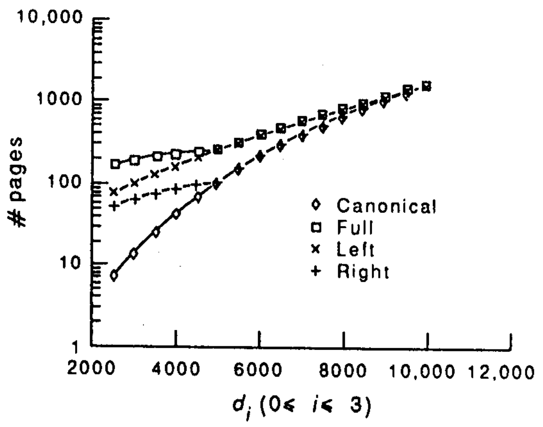


Fig. 5. Varying the number of not-NULL attributes.

4.3.2. Varying all d_i parameters. In the subsequent experiment we want to demonstrate the effect of varying the number of defined attributes, i.e. simultaneously varying d_i for $(0 \leqslant i \leqslant 3)$, while keeping the number of objects and the fan-out fixed (Table 4). The parameters d_0, d_1, d_2 and d_3 were simultaneously increased, i.e. the values are kept identical. The plot in Fig. 5 shows the access support relation sizes for all different extensions under no decomposition. As the d_i values increase the sizes of the different extensions grow proportionally. As the d_i values approach the c_i values, the storage costs for all different extensions approach each other—because then (almost) all paths originate in t_0 and lead to t_n.

5. QUERY PROCESSING

In this section we evaluate the usefulness and the costs of the different extensions and decompositions to query processing.

5.1. Kinds of queries

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

5.1.1. Backward queries. In this query expression the objects o_i of type t_i are retrieved based on the membership of some other object o_j of type t_j in the path expression $o_i.A_{i+1}.\cdots.A_j$.

$$\begin{aligned} Q^{(i,j)}(bw) := \textbf{range} \quad & o_i : t_i, o_j : t_j \\ \textbf{retrieve} \quad & o_i \\ \textbf{where} \quad & o_j \textbf{ in } o_i.A_{i+1}.\cdots.A_j \textbf{ and } Pred(o_j) \end{aligned}$$

Here we assume that $Pred(o_j)$ is some predicate that is satisfied for only one (unique) object o_j of type t_j. In our cost analysis the cost to find this one object is neglected. We assume, for example, that a direct reference to o_j is given.

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

$$\begin{aligned} Q^{(i,j)}(fw) := \textbf{range} \quad & o_i : t_i, o_j : t_j \\ \textbf{retrieve} \quad & o_j \\ \textbf{where} \quad & o_j \textbf{ in } o_i.A_{i+1}.\cdots.A_j \textbf{ and } Pred(o_i) \end{aligned}$$

Again, $Pred(o_i)$ is assumed to be a predicate that uniquely identifies one object of type t_i.

5.2. Estimating query costs

In the following estimates we will frequently use a well-known formula. Yao [23] has determined the number of page accesses for retrieving k out of n objects distributed over m pages, where each page contains n/m objects. This number, denoted as $y(k, m, n)$, is:

$$y(k, m, n) = \left\lceil m\left(1 - \prod_{i=1}^{k} \frac{n(1 - 1/m) - i + 1}{n - i + 1}\right)\right\rceil. \tag{26}$$

5.2.1. No access support. We generally assume that objects are clustered dependent on their type. Thus, the number of pages needed to store all objects of type t_i is estimated as:

$$op_i = \left\lceil \frac{c_i}{(PageSize/size_i)} \right\rceil. \tag{27}$$

Forward Query The cost of evaluating a forward query $Q^{(i,j)}(fw)$ without any access support is given as:

$$Qnas^{(i,j)}(fw) = 1 + \sum_{l=i+1}^{j-1} y(\lceil RefBy(i, l, 1)\rceil, op_l, c_l) \tag{28}$$

This cost is deduced as one page access to retrieve the object o_i plus the access to all objects of type $t_l (i < l < j)$ that lie on a path originating in o_i.

Backward query Backward queries induce the following cost—under the assumption that no access support exists:

$$Qnas^{(i,j)}(bw) = op_i + \sum_{l=i+1}^{j-1} y(\lceil RefBy(i, l, d_i)\rceil, op_l, c_l) \tag{29}$$

Basically the backward query is evaluated by an exhaustive search. All objects of type t_l for $(i < l < j)$ that are connected with any object of type t_i have to be inspected, i.e. $RefBy(i, l, d_i)$ objects have to be retrieved for each type t_l $(i < l < j)$.

5.2.2. With access support. In the following we will need the parameter B^+_{fan} which is the fan-out of the B^+-tree. This parameter can be derived as:

$$B^+_{fan} = (PageSize/(PPsize + OIDsize)). \tag{30}$$

The height of the B^+-tree—not considering the leaves—for the relation $[\![t_0 . A_1 . \cdots . A_n]\!]^{(i,j)}_X$ is denoted $ht^{(i,j)}_X$. We will omit the derivation of the height—in realistic database applications the height will mostly be 2.

From the height and the fan-out of the B^+-tree we can deduce the number of (internal) pages (without leaves) in the B^+-tree for the relation $[\![t_0 . A_1 . \cdots . A_n]\!]^{(i,j)}_X$, which is denoted $pg^{(i,j)}_X$.

The number of leave pages of the B^+-tree per object of type t_i in the access support relation depends clearly on the extension. It can be estimated as follows:

$$nlp^{(i,j)}_{full} = \left\lceil \frac{ap^{(i,j)}_{full}}{d_i + 1} \right\rceil \tag{31}$$

$$nlp^{(i,j)}_{left} = \left\lceil \frac{ap^{(i,j)}_{left}}{RefBy(0, i)P_{A_i}} \right\rceil \tag{32}$$

$$nlp^{(i,j)}_{right} = \left\lceil \frac{ap^{(i,j)}_{right}}{Ref(i, n) + 1} \right\rceil \tag{33}$$

$$nlp^{(i,j)}_{can} = \left\lceil \frac{ap^{(i,j)}_{can}}{Ref(i, n)P_{RefBy}(0, i)} \right\rceil \tag{34}$$

For the B^+ tree for the inversely clustered access support relation the value is denoted $Rnlp^{(i,j)}_X$ and is derived analogously.

If the B^+ tree has exactly one intermediate layer of nodes then the cost for a supported forward query can roughly be approximated as follows:

$$\begin{aligned} Qsup^{(i,j)}_X(fw, dec) = & \sum_{\substack{i_\alpha, i_{\alpha+1} \in dec \\ (i_\alpha = i < i_{\alpha+1})}} (ht^{(i_\alpha, i_{\alpha+1})}_X + nlp^{(i_\alpha, i_{\alpha+1})}_X) + \sum_{\substack{i_\alpha, i_{\alpha+1} \in dec \\ (i_\alpha < i < i_{\alpha+1})}} (ap^{(i_\alpha, i_{\alpha+1})}_X) + \\ & \sum_{\substack{i_\alpha i_{\alpha+1} \in dec \\ (i < i_\alpha < j)}} (1.0 + y(\lceil RefBy(i, i_\alpha, 1))\rceil, pg^{(i_\alpha, i_{\alpha+1})}_X - 1, (pg^{(i_\alpha, i_{\alpha+1})}_X - 1)B^+_{fan}) + p \\ & y(\lceil RefBy(i, i_\alpha, 1)\rceil nlp^{(i_\alpha, i_{\alpha+1})}_X atpp^{(i_\alpha, i_{\alpha+1})}_X, ap^{(i_\alpha, i_{\alpha+1})}_X, \#[\![t_0 . A_1 . \cdots . A_n]\!]^{(i_\alpha, i_{\alpha+1})}_X)). \end{aligned} \tag{35}$$

In this formula we are given a decomposition $dec := (0 = i_0, i_1, \ldots, i_k = n)$. Depending on this decomposition the forward query $Q^{(i,j)}(fw)$ is evaluated. We distinguish two cases:

1. The first sum covers the case that $i = i_\alpha$ for some $0 \leqslant \alpha < k$. In this case only one path through the B^+-tree has to be traversed and the leave pages for one value $(nlp^{(i_\alpha, i_{\alpha+1})}_X)$ are retrieved.
2. The second sum handles the special case that i is not the left border of some decomposition, i.e. there is no $i_\alpha \in dec$ such that $i_\alpha = i$. All pages of the access support relation partition $[\![t_0 . A_1 . \cdots . A_n]\!]^{(i_\alpha, i_{\alpha+1})}_X$ that covers i have to be inspected. This number equals $ap^{(i_\alpha, i_{\alpha+1})}_X$.

Finally, the third sum accounts for accessing the partitions that lead to j. Within each partition $[\![t_0 . A_1 . \cdots . A_n]\!]^{(i_\alpha, i_{\alpha+1})}_X$ we have to retrieve

- the root of the B^+-tree
- the intermediate pages of the B^+-tree that contain (the intervals of) the $RefBy(i, i_\alpha, 1)$ object identifiers of type t_{i_α}.

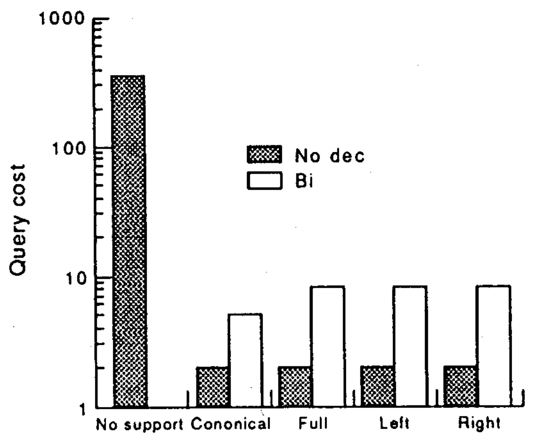


Fig. 6. Query costs for a backward query.

Table 5
Application characteristics

n	4				
Number of objects	c_0 100	c_1 500	c_2 1000	c_3 5000	c_4 10,000
Number of objects with defined A_{i+1} attribute	d_0 90	d_1 400	d_2 800	d_3 2000	d_4 —
Fan-out	f_0 2	f_1 2	f_2 3	f_3 4	f_4 —
Size of objects	$size_0$ 500	$size_1$ 400	$size_2$ 300	$size_3$ 300	$size_4$ 100

- the data pages of the access support relation partition $[\![t_0.A_1.\cdots.A_n]\!]_X^{(i_\alpha,i_\alpha+1)}$ that contain the $RefBy(i, i_\alpha, 1)$ object identifiers of type t_{i_α}

The costs of a backward query are derived analogously (see Ref. [24]).

5.3. Sample results

5.3.1. Query costs in comparison. Figure 6 visualizes the cost of a backward query of the form $Q^{(0,4)}(bw)$ for the application specific parameters shown in Table 5. 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 support relations are lower than for the binary decomposed relations—the reason being that the semi-join across partitions is avoided.

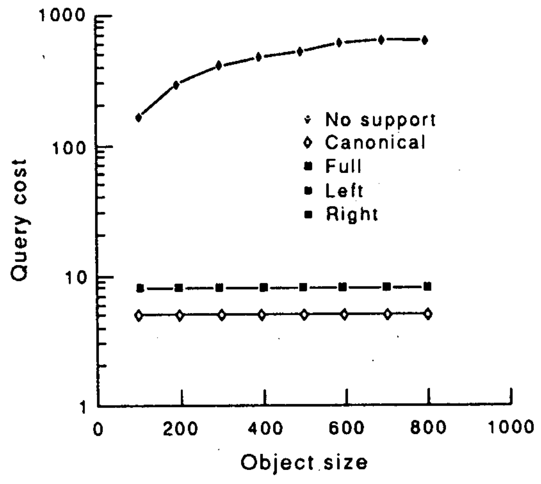


Fig. 7. Query costs for a backward query under varying object size.

Table 6

Application characteristics					
n	4				
Number of objects	c_0 100	c_1 500	c_2 1000	c_3 5000	c_4 10,000
Number of objects with defined A_{i+1} attribute	d_0 90	d_1 400	d_2 800	d_3 2000	d_4 —
Fan-out	f_0 2	f_1 2	f_2 3	f_3 4	f_4 —
Size of objects	$size_0$ 100 ··· 800	$size_1$ 100 ··· 800	$size_2$ 100 ··· 800	$size_3$ 100 ··· 800	$size_4$ 100 ··· 800

5.3.2. Query costs depending on object size. Figure 7 shows the cost of a backward query of the form $Q^{(0,4)}(bw)$ depending on the size of the stored objects, i.e. the parameter $size_i$ is simultaneously varied for $(0 \leqslant i \leqslant 4)$ (Table 6). The access support relations are decomposed into binary partitions. As can be seen in Fig. 7 the object size does not influence the query costs for supported queries (as expected). Only the cost of non-supported queries grows proportional to the object size. Note, that in Fig. 7 the values for full, left, and right extensions overlap (marked with filled squares).

5.3.3. Which queries are supported? As described before, not all queries are supported by certain extensions of the access support reation. Also, the decomposition of the access support relations has a major effect on the cost of a query. For demonstration, let us use the application characteristics shown in Table 7.

The plot in Fig. 8 shows the query costs of a backward query of the form: $Q^{(0,3)}(bw)$. We computed the results for two decompositions: (1) the decomposition into binary partitions and (2) the non-decomposed representation. From our preceding discussions we know that only the left-complete and the full extension of the access support relation can possibly be used to evaluate the query. It turns out, that the evaluation utilizing the full/left-complete, non-decomposed access support relations are costlier than the non-supported evaluation. The reason is that the rather large access support relations have to be exhaustively searched under no decomposition, i.e. all pages have to be inspected. This is due to the lack of a cluster index on OIDs of type t_3—remember, the

Table 7

Application characteristics					
n	4				
Number of objects	c_0 10^4	c_1 10^4	c_2 10^4	c_3 10^4	c_4 10^4
Number of objects with defined A_{i+1} attribute	d_0 10 ··· 10^4	d_1 10 ··· 10^4	d_2 10 ··· 10^4	d_3 10 ··· 10^4	d_4 —
Fan-out	f_0 2	f_1 2	f_2 2	f_3 2	f_4 —
Size of objects	$size_0$ 120	$size_1$ 120	$size_2$ 120	$size_3$ 120	$size_4$ 120

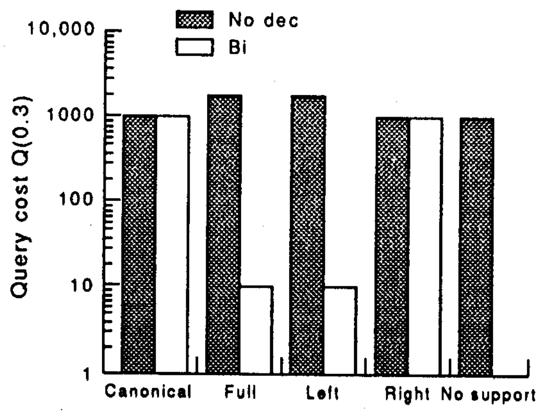


Fig. 8. Query costs for a backward query $Q^{(0,3)}(bw)$.

Table 8

Application characteristics					
n	4				
Number of objects	c_0 400,000	c_1 400,000	c_2 400,000	c_3 400,000	c_4 400,000
Number of objects with defined A_{i+1} attribute	d_0 10	d_1 100	d_2 1000	d_3 100,000	d_4 —
Fan-out	f_0 $10\cdots100$	f_1 $10\cdots100$	f_2 $10\cdots100$	f_3 $10\cdots100$	f_4 —
Size of objects	$size_0$ 120	$size_1$ 120	$size_2$ 120	$size_3$ 120	$size_4$ 120

access support relations are only clustered at partition boundaries (cf. Section 3.3). Therefore, if the database designer anticipates the frequent occurrence of such a query the non-decomposed extension of the access support relation should be avoided.

5.3.4. An application favoring canonical/left over full/right. The parameters shown in Table 8 describe an application that favors canonical and left-complete extensions over full and right-complete extensions of the access support relation. The evaluation costs of a backward query $Q^{(0,4)}(bw)$ for varying fan-out values are plotted in Fig. 9.

6. MAINTENANCE OF ACCESS SUPPORT RELATIONS

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 support relation extensions. The problem of automatic maintenance of the access support relations is addressed and the cost analyzed.

For notational convenience we will assume that all attributes $A_1, \ldots, A_n$ in the path expression $t_0 . A_1 . \cdots . A_n$ are set-valued, i.e.:

$$\textbf{type } t_{i-1} \textbf{ is } [\ldots, A_i : \{t_i\}, \ldots]$$

In the subsequent discussion we will denote objects of type t_i as o_i for $(0 \leqslant i \leqslant n)$.

Then only the following two primitive update operations have to be considered for the maintenance of the access support relations:

$$ins^i \equiv o_{i-1}.insert\_A_i(o_i)$$

$$del^i \equiv o_{i-1}.remove\_A_i(o_i)$$

In statement ins^i a new object is inserted into the set associated with attribute A_i of object o_{i-1}. In statement del^i an object o_i is removed from the set. All other—higher-level—update operations can be decomposed into these primitives.

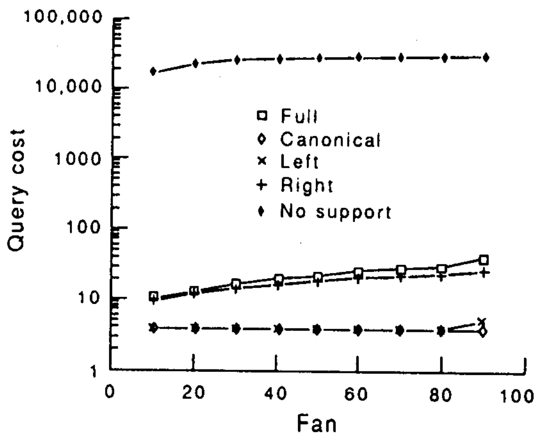


Fig. 9. Cost of a backward query $Q^{(0,4)}(bw)$.

6.1. *Terminology*

For describing the maintenance algorithms we need a few preliminary definitions:

Definitions 6.1. (Direct predecessors and direct successors)

The set of direct predecessors $DP_{A_i}^{t_{i-1}}$ (and successors $DS_{A_{i+1}}$, respectively) of an object o_i of type t_i with respect to the attribute A_i and the type t_{i-1} (w.r.t. the attribute A_{i+1}) is defined as follows:

$$DP_{A_i}^{t_{i-1}}(o_i) = \{o_{i-1} \mid o_i \in o_{i-1}.A_i \text{ for } o_{i-1} \text{ of type } t_{i-1}\}$$

$$DS_{A_{i+1}}(o_i) = \{o_{i+1} \mid o_{i+1} \in o_i.A_{i+1} \text{ for } o_{i+1} \text{ of type } t_{i+1}\} \qquad \square$$

As long as no misunderstandings are possible we will often drop the super-and/or the subscripts.

In the above definition we have to note that all instances of subtypes belong to the extension of a type—therefore substitutability of subtype instances in place of supertype instances is implicitly accounted for.

The direct predecessors and successors, respectively, are computed as follows:

- *Computation of the direct predecessors $DP_{A_i}^{t_{i-1}}(o_i)$*

```
$DP_{A_i}^{t_{i-1}}(o_i) := \emptyset$;          !! initialize to the empty set
foreach ($o_{i-1}$ in ext($t_{i-1}$))
  if ($o_i$ in $o_{i-1}.A_i$)
    $DP_{A_i}^{t_{i-1}}(o_i) := DP_{A_i}^{t_{i-1}}(o_i) \cup \{o_{i-1}\}$;
```

- *Computation of the direct successors $DS_{A_{i+1}}(o_i)$*

```
$DS_{A_{i+1}}(o_i) := o_i.A_{i+1}$;
```

It should be obvious that the computation of the direct predecessors incurs much higher cost (in the order of the cardinality of the extension of type t_{i-1}) than the simple look-up of the direct successors.

Definition 6.2. (Predecessor-/successor-relation)

The predecessor relation PR^i—a set of $(i+1)$-ary tuples of object identifiers, which represent reference chains leading into the object o_i w.r.t. the given path expression $t_0.A_1.\cdots.A_n$—and the successor relation SR^i of $(n-i+1)$-ary tuples representing reference chains emanating from the object o_i are defined as follows:

$$PR^i(o_i) = \{(NULL, \ldots, NULL, id(o_k), \ldots, id(o_{i-1}), id(o_i)) \mid$$
$$(k = 0 \text{ or } DP(o_k) = \emptyset) \text{ and } o_{j-1} \in DP(o_j) \text{ for all } k < j \leqslant i\}$$

$$SR^i(o_i) = \{(id(o_i), id(o_{i+1}), \ldots, id(o_s), NULL, \ldots, NULL) \mid$$
$$(s = n \text{ or } DS(o_s) = \emptyset) \text{ and } o_j \in DS(o_{j-1}) \text{ for all } i < j \leqslant s\} \qquad \square$$

The specification of the position i within the path expression $t_0.A_1.\cdots.A_n$ is relevant since the same object type can occur at different positions within a single path expression.

$\widehat{PR}^i(o_i)$ denotes the predecessor relation which contains only left-complete paths, i.e. only those paths that originate in an object o_0 of type t_0 and lead to o_i—in this case $k = 0$ has to hold in the terminology of the above definition. $\widehat{SR}^i(o_i)$ is defined analogously as the successor relation containing only right-complete paths.

6.2. *Maintenance upon object insertion*

We will now sketch the maintenance algorithm for the update operation (ins^i):

$$o_{i-1}.\,insert\_A_i(o_i);$$

We can extend the definition of predecssor and successor relations to argument sets in a natural way: $PR^i(M) := \cup_{o_i \in M} PR^i(o_i)$ and $PR^i(\emptyset) := \{(NULL, \ldots, NULL)\}$. Analogously, $SR^i(M) := \cup_{o_i \in M} SR^i(o_i)$ and $SR^i(\emptyset) := \{(NULL, \ldots, NULL)\}$.

Furthermore, in the specification of the (recursive) update algorithm we will implicitly assume the following: $PR^0(o_0) := \{(id(o_0))]$ and $SR^n(o_n) := \{(id(o_n))\}$.

6.2.1. Full extension: $X = full$.

1. Compute the predecessor relation of o_{i-1}

$$PR^{i-1}(o_{i-1}) := \begin{cases} \pi_{S_0, \ldots, S_{i-1}}(\sigma_{S_{i-1} = id(o_{i-1})} [\![t_0 . A_1 . \cdots . A_n]\!]_{full}) & \text{if it is } \neq \emptyset \\ \{(NULL, \ldots, NULL, id(o_{i-1}))\} & \text{else} \end{cases}$$

2. *Compute the successor relation of* o_i

$$SR^i(o_i) := \begin{cases} \pi_{S_i, \ldots, S_n}(\sigma_{S_i = id(o_i)} [\![t_0 . A_1 . \cdots . A_n]\!]_{full}) & \text{if it is} \neq \emptyset \\ \{(id(o_i), NULL, \ldots, NULL)\} & \text{else} \end{cases}$$

3. *Update the ASR*

$$[\![t_0 . A_1 . \cdots . A_n]\!]_{full} := [\![t_0 . A_1 . \cdots . A_n]\!]_{full} \cup (PR^{i-1}(o_{i-1}) \times SR^i(o_i))$$

4. *Delete obsolete information*

The following information

$$\sigma_{S_{i-1} = id(o_{i-1}) \wedge S_i = NULL} [\![t_0 . A_1 . \cdots . A_n]\!]_{full} \text{ and } \sigma_{S_{i-1} = NULL \wedge S_i = id(o_i)} [\![t_0 . A_1 . \cdots . A_n]\!]_{full}$$

has to be removed from the ASR—if the sets are non-empty.

In the maintenance of the full extension all update information, i.e. the predecessor as well as the successor relation, can be derived from the access support relation. Thus, no search within the object representation has to be performed. The cross product of the predecessor and the successor relation is inserted into the access support relation. Subsequently, some tuples have to be removed, which may have become obsolete due to step (3). Note that the various steps of the algorithm should be "meshed" in order to optimize the performance—for better clarity they are separated in this description.

6.2.2. Left-complete extension: $X = left$. We will now see, that the update information necessary to maintain the left-complete extension may not be available within the access support relation.

1. *Compute the left-complete predecessor relation of* o_{i-1}

$$\widehat{PR}^{i-1}(o_{i-1}) := \pi_{S_0, \ldots, S_{i-1}}(\sigma_{S_{i-1} = id(o_{i-1})} [\![t_0 . A_1 . \cdots . A_n]\!]_{left})$$

Note, that for $(i - 1 = 0)$ implicitly $PR^0(o_0) := \{(id(\sigma_0))\}$ is assumed.

If $\widehat{PR}^{i-1}(o_{i-1}) = \emptyset$ holds, skip steps (2), (3) and (4), because no update of the access support relation $[\![t_0 . A_l . \cdots . A_n]\!]_{left}$ is necessary.

2. *Compute the successor relation of* o_i

$$SR^i(o_i) := \begin{cases} \pi_{S_i, \ldots, S_n}(\sigma_{S_i = id(o_i)} [\![t_0 . A_1 . \cdots . A_n]\!]_{left}) & \text{if this is} \neq \emptyset \\ \{(id(o_i))\} \times SR^{i+1}(DS_{A_{i+1}}(o_i)) & \text{else} \end{cases}$$

Again, for $i = n$ the exception $SR^n(o_n) := \{(id(o_n))\}$ is implicitly assumed.

3. *Update the access support relation*

$$[\![t_0 . A_1 . \cdots . A_n]\!]_{left} := [\![t_0 . A_1 . \cdots . A_n]\!]_{left} \cup (\widehat{PR}^{i-1}(o_{i-1}) \times SR^i(o_i))$$

4. *Delete obsolete information*

The set of tuples

$$\sigma_{S_{i-1} = id(o_{i-1}) \wedge S_i = NULL} [\![t_0 . A_1 . \cdots . A_n]\!]_{left}$$

has to be removed from the ASR—if it is not empty.

Example 6.3. Consider the object base shown in Fig. 10. An object o_i is identified by the OID id_i $(1 \leqslant i \leqslant 10)$. Furthermore, assume that the access support relation $[\![t_0 . A_1 . A_2 . A_3 . A_4]\!]_{left}$ exists. The update operation $o_6.insert\_A_1(o_7)$ has to be reflected in the given access support relation $[\![t_0 . A_1 . A_2 . A_3 . A_4]\!]_{left}$.

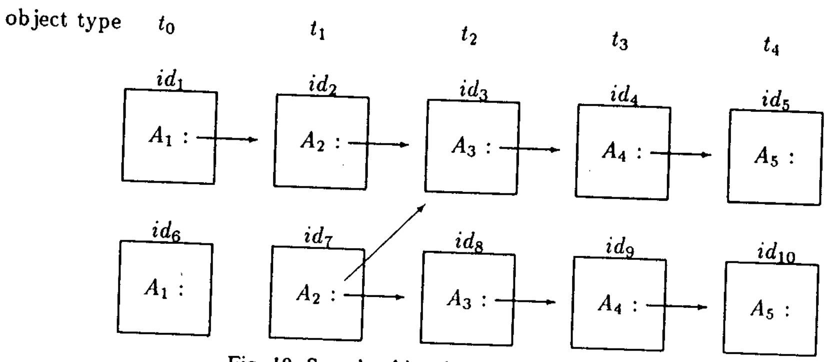


Fig. 10. Sample object base extension.

The predecessor relation of o_6 is $\widehat{PR}^0(o_6) = \{(id_6)\}$. In general, the left-complete predecessor relation is always derivable from the left-complete extension of the ASR†—or applying the exception of $i = 0$, as above.

The computation of the successor relation $SR^1(o_7)$ yields the empty set after inspecting the ASR. Thus, the direct successors of o_7 have to be "computed" by looking up the object representation of o_7. This yields the set $\{o_3, o_8\}$.

The direct successors of o_8 have to be looked up in the object base, whereas the successor relation of o_3 is derivable from the access support relation.

6.3. *Maintenance costs*

Let us now analyze the maintenance costs of the insert operation ($ins^i \equiv o_{i-1}.\, insert\_A_i(o_i)$) on the access support relations for the path expression $t_0.A_1.\cdots.A_i.\cdots.A_n$. For simplicity, we assume that for $0 < k, i \leqslant n, i \neq k$ either o_{i-1} is not of type t_{k-1} or $A_k \neq A_i$. This simplifying condition prevents an object insertion to affect different positions in a single path expression. It follows that o_i has to be of type t_i.

The update costs consist of three parts:

1. the costs for updating the set associated with attribute A_i of object o_{i-1}
2. searching the identifiers for the successor- and predecessor relations that have to be updated, and
3. updating the access support relations.

The cost for updating $o_{i-1}.A_i$ amounts to 3, i.e. one page access to retrieve the object representations of o_{i-1} and $o_{i-1}.A_i$; and one page access to write the modified set-structured object $o_{i-1}.A_i$ back to secondary storage.

The next step consists of materializing the relations $PR^{i-1}(o_{i-1})$ and $SR^i(o_i)$, depending on the selected extension of the access support relations. Here we only consider the costs encountered if the search has to be performed in the object representation, i.e. if $SR^i(o_i)$ and $PR^{i-1}(o_{i-1})$ cannot be materialized from the access support relations.

If we have a full extension we do not need any search in the data since all necessary information is contained in the access support relations.

If we have a left-complete extension we have to search the paths from object o_i in direction t_n to materialize $SR^i(o_i)$. But this is only necessary if o_{i-1} is referenced by some object in t_0, and o_i is not already contained in the access support relation, i.e. it was not yet referenced by some path originating in an object in t_0. Otherwise, $SR^i(o_i)$ is either contained in the access support relations or not needed.

The cost for searching in the case of a right-complete extension can be approximated analogously. A search in the data to create $PR^{i-1}(o_{i-1})$ is only needed if o_i was already present in the access support relation and if o_{i-1} is absent. Only under this condition one (or more) new right-complete paths have to be added to the access support relations.

†Note, that this does not hold for the right-complete and canonical extension of the access support relations.

Table 9

Application characteristics					
n	4				
Number of objects	c_0 1000	c_1 5000	c_2 10,000	c_3 50,000	c_4 100,000
Number of objects defined A_{i+1} attribute	d_0 900	d_1 4000	d_2 8000	d_3 20,000	d_4 —
Fan-out	f_0 2	f_1 2	f_2 3	f_3 4	f_4 —
Size of objects	$size_0$ 500	$size_1$ 400	$size_2$ 300	$size_3$ 300	$size_4$ 100

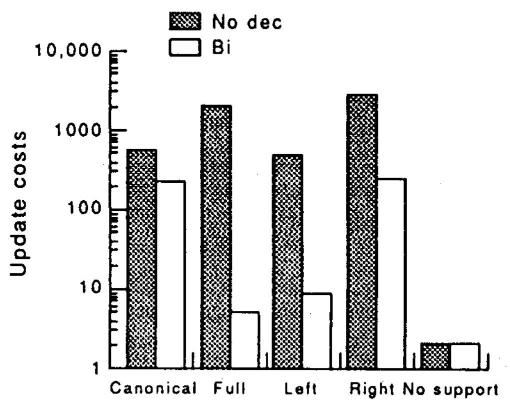


Fig. 11. Update costs for a fixed application profile.

In the case of a canonical extension we have to search for a complete path in both directions. Since a forward search is cheaper than a backward search we start therewith to set up $SR^i(o_i)$. The forward search from o_i to t_n has only to be performed if there does not already exist a complete path through o_i. We start the backward search to materialize $PR^{i-1}(o_i - 1)$ only if we have found a connection from o_i to t_n. The backward search itself is only necessary if there does not already exist a complete path through o_{i-1}.

The derivation of the cost formulas is beyond the scope of this paper and can be found in Ref. [24].

6.4. Sample results

6.4.1. Update costs for fixed application characteristics. We compare update costs for different access support relation extensions and decompositions on the basis of the application profile shown in Table 9. The update costs for an update operation *ins*^{3} are plotted in Fig. 11. The access support relations are, alternatively, in binary decomposition or non-decomposed. Since the update is at the right-hand side of the path expression, the left-complete extension under binary decomposition is very much superior to the right-complete extension. For an update *ins*^{0} the right-complete extension

Table 10

Application characteristics					
n	4				
Number of objects	c_0 1000	c_1 5000	c_2 10,000	c_3 50,000	c_4 100,000
Number of objects with defined A_{i+1} attribute	d_0 900	d_1 4000	d_2 8000	d_3 20,000	d_4 —
Fan-out	f_0 2	f_1 1	f_2 1	f_3 4	f_4 —
Size of objects	$size_0$ 500	$size_1$ 400	$size_2$ 300	$size_3$ 300	$size_4$ 100

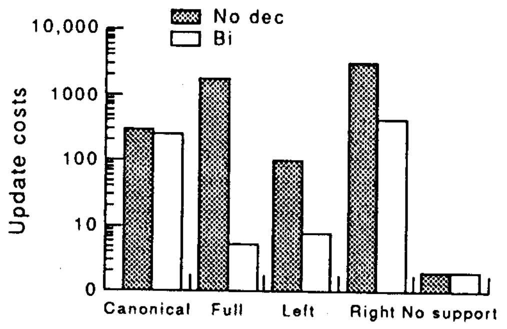


Fig. 12. Update costs for a fixed application profile.

would be drastically better, whereas the canonical extension is problematic under any update because a search in the data is always necessary.

6.4.2. Update costs for another fixed application characteristics. Let us, for comparison, show a slightly different application profile (Table 10). The update costs for an update operation ins^3 are plotted in Fig. 12. Again, the update costs of the left-complete and full extension are almost comparable. For non-decomposed access support relations the cost is extraordinarily high because of the high cost for searching in the non-clustered information.

6.4.3. Update costs under varying object size. Consider the application-specific parameters shown in Table 11, within which we will simultaneously increase the sizes of objects of all types within the interval 100 . . . 800. The plot in Fig. 13 visualizes the effect of varying object sizes on the update costs of ins^1. The access support relations are in binary decomposition. We see that the update costs for canonical and right-complete extension grow as the object sizes increase. This is due to the high search overhead within the data (object representation) that has to be performed. Remember, that in the case of canonical and right-complete extension an exhaustive search may become necessary to establish the paths that lead from t_0 to the object being updated. For the left-complete extension only a forward search is needed which is only marginally affected by increasing object sizes.

Table 11
Application characteristics

n	4				
Number of objects	c_0 1000	c_1 5000	c_2 10,000	c_3 50,000	c_4 100,000
Number of objects with defined A_{i+1} attribute	d_0 900	d_1 4000	d_2 8000	d_3 20,000	d_4 —
Fan-out	f_0 2	f_1 2	f_2 3	f_3 4	f_4 —
Size of objects	$size_0$ 100 · · · 800	$size_1$ 100 · · · 800	$size_2$ 100 · · · 800	$size_3$ 100 · · · 800	$size_4$ 100 · · · 800

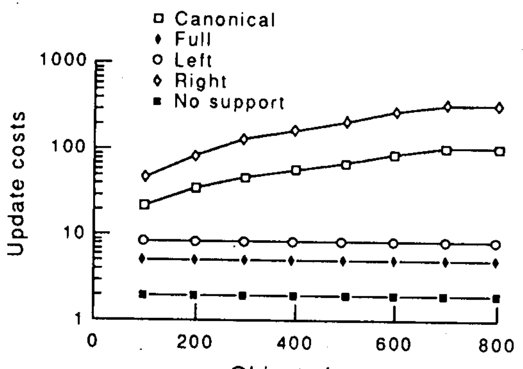


Fig. 13. Update costs for varying object sizes.

Table 12

Application characteristics					
n	4				
Number of objects	c_0 1000	c_1 5000	c_2 10,000	c_3 50,000	c_4 100,000
Number of objects with defined A_{i+1} attribute	d_0 900	d_1 4000	d_2 8000	d_3 20,000	d_4 —
Fan-out	f_0 2	f_1 2	f_2 3	f_3 4	f_4 —
Size of objects	$size_0$ 500	$size_1$ 400	$size_2$ 300	$size_3$ 300	$size_4$ 100

7. COSTS OF TYPICAL OPERATION MIXES

7.1. *Describing 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:

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

where for $(1 \leqslant i \leqslant p)$ 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 $\Sigma_{i=1}^{p} w_i = 1$ 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.

7.2. *Some sample results*

7.2.1. *Update mix under binary decomposition.* The application profile shown in Table 12 is used: The query mix Q_{mix} consists of:

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

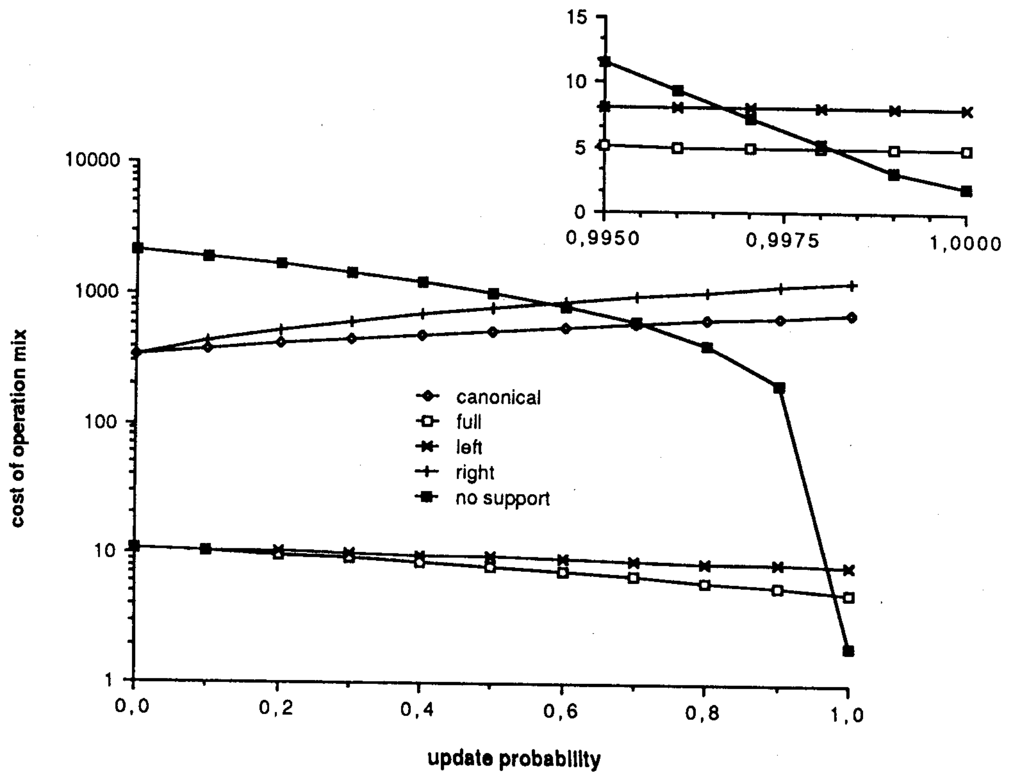


Fig. 14. Operation mix for binary decomposition.

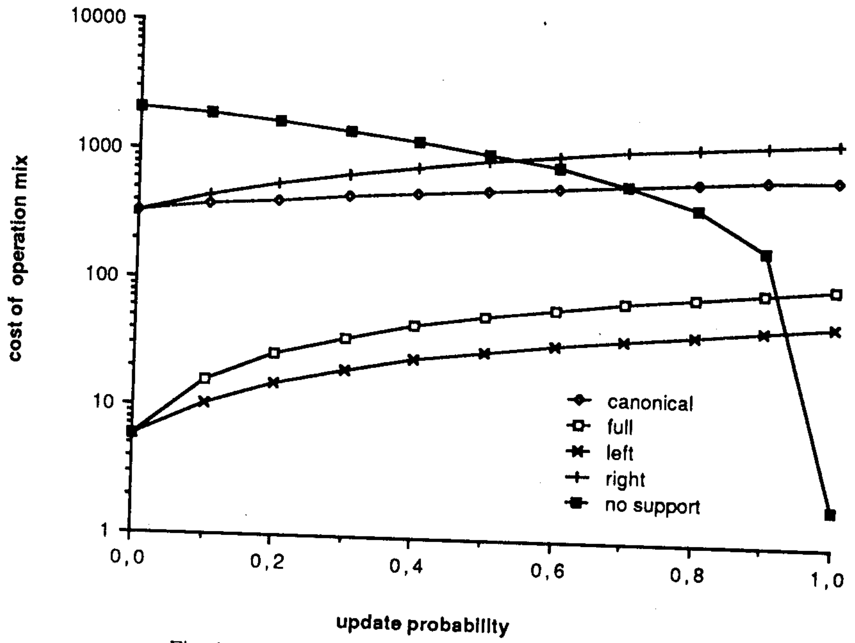


Fig. 15. Operation mix for the decomposition (0, 3, 4).

The update mix consists of:

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

This means that, when a query is performed, any one of the queries is chosen with the specified probability. The same holds for update operations.

Figure 14 shows the costs 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 performs as well as the full extension. The break even point between no support and full extension is at an update probability of 0.998 (not shown in the diagram).

7.2.2. *Non-binary decompositions of the access support relations.* The experiment was run again for the (0, 3, 4) decomposition of the access support relations. The result is shown in Fig. 15.

7.2.3. *Comparison: left-complete vs full extensions.* For the application characterization shown in Table 13 the anticipated costs for a database operation mix consisting of the following queries and updates were computed:

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

In Fig. 16 the costs for the operation mix under left-complete and full extension of the access support relations are plotted for two different decompositions: (1) binary decomposition (0, 1, 2, 3, 4, 5) and (2) the decomposition (0, 3, 4, 5).

Table 13

Application characteristics

n	5					
Number of objects	c_0 1000	c_1 1000	c_2 5000	c_3 10,000	c_4 100,000	c_5 100,000
Number of objects with defined A_{i+1} attribute	d_0 100	d_1 1000	d_2 3000	d_3 8000	d_4 100,000	d_5 —
Fan-out	f_0 2	f_1 2	f_2 3	f_3 4	f_4 10	f_5 —
Size of objects	$size_0$ 600	$size_1$ 500	$size_2$ 400	$size_3$ 300	$size_4$ 300	$size_5$ 100

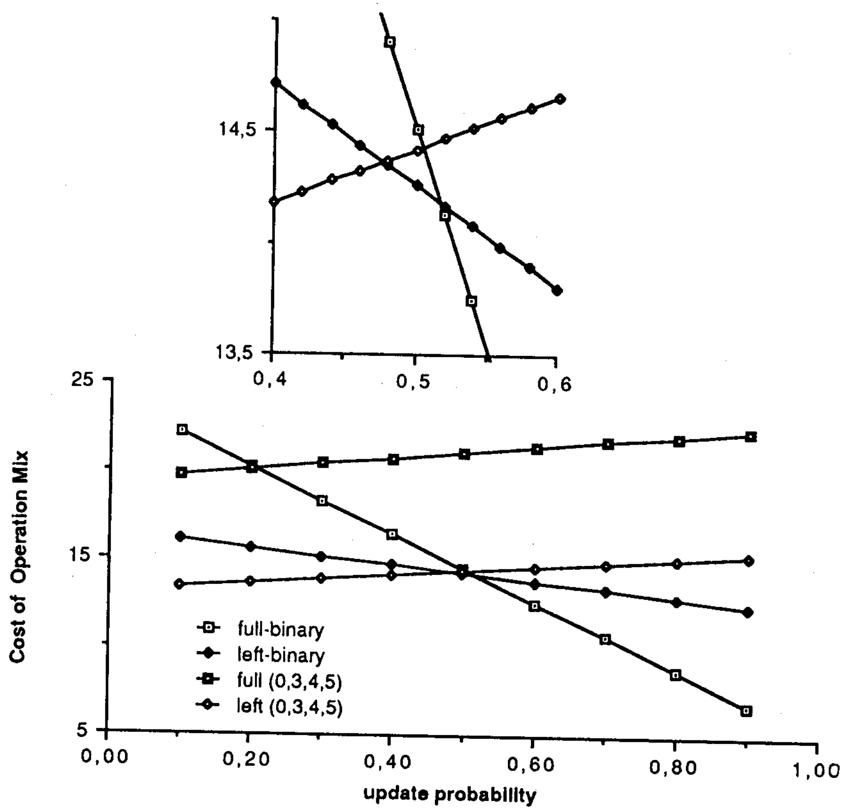


Fig. 16. Operation mix for full and left-complete access support relations.

7.2.4. Comparison: right-complete vs full extension. The application profile shown in Table 14 is being used.

For this application characterization the normalized costs for a database operation mix consisting of the following queries and updates were 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 17 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 update probabilities less than 0.005 the right-complete extension is even better than the full extension under this particular decomposition. This break-even point is emphasized in the upper plot of Fig. 17.

Table 14

Application characteristics

n	5					
Number of objects	c_0 100,000	c_1 100,000	c_2 50,000	c_3 10,000	c_4 1000	c_5 1000
Number of objects with defined A_{i+1} attribute	d_0 100,000	d_1 10,000	d_2 30,000	d_3 10,000	d_4 100	d_5 100
Fan-out	f_0 1	f_1 10	f_2 20	f_3 4	f_4 1	f_5 —
Size of objects	$size_0$ 600	$size_1$ 500	$size_2$ 400	$size_3$ 300	$size_4$ 200	$size_5$ 700

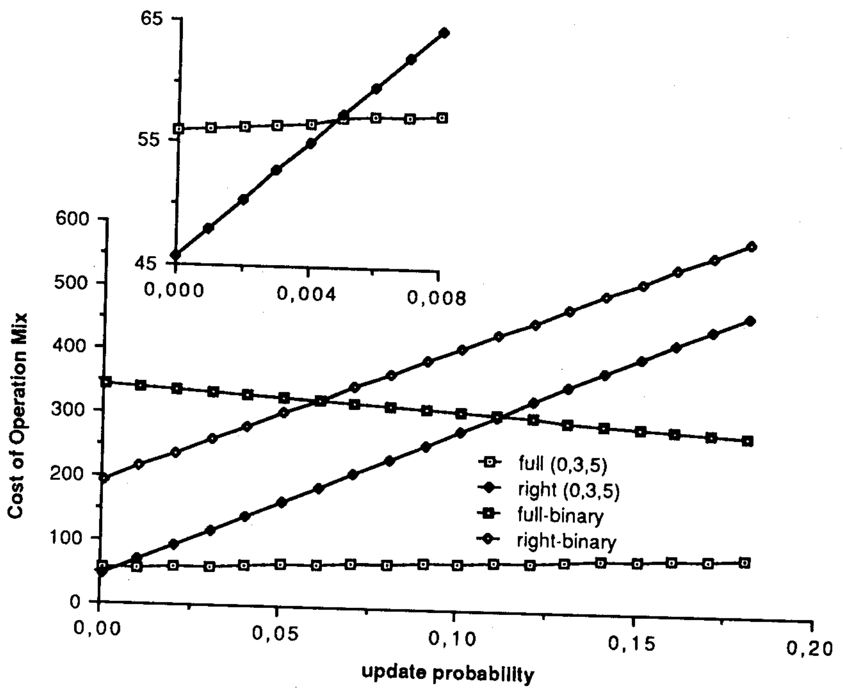


Fig. 17. Isolating right-complete and full extension.

8. CONCLUSION AND FUTURE WORK

In this work we have tackled a major problem in optimizing object-oriented DBMSs: the evaluation of path expressions. We have described the framework for a whole class of indexing structures, which we call *access support relations*. 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 generally predict the best possible design choices: this is highly application dependent. Therefore, a complete analytical cost model was 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 then be used to compute for all (feasible) design choices the expected cost (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. The first step in the direction of such a comprehensive cost model for supporting the physical object base design has been presented in this paper.

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 (extensions and decompositions). It is not generally predictable for a whole application domain which extensions and decompositions will be optimal—this decision is highly application and operation-mix dependent.

The access support relation manager as well as the cost model have been implemented on an experimental basis. First benchmarks with the access support relation manager support our analytical analysis presented in this paper. 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.

Of course, the existing access support relations have to be effectively utilized in the query evaluation. In our view, the exploitation of existing index structures should be transparent to the database user who may not even be aware of the existence of (some of) the access support relations. For this purpose we developed a rule-based query optimizer for GOM that generates a query evaluation plan which exploits existing access support relations [20]. The analytical cost model is being incorporated into the query optimizer for comparing the evaluation costs of different alternative evaluation plans.

Acknowledgements—This work was partially supported by the German Research Council DFG under contract number SFB 346. Chin Chen's help in refining the original cost model is gratefully acknowledged.

REFERENCES

[1] A. Kemper and G. Moerkotte. Access suppoort in object bases. In *Proc. ACM SIGMOD Conf. on Management of Data*, Atlantic City, N.J., pp. 364–374 (1990).
[2] A. Kemper and M. Wallrath. An analysis of geometric modeling in database systems. *ACM Comput. Surv.* **19**(1), 47–91 (1987).
[3] P. Valduriez. Join indices. *ACM Trans. Database Syst.* **12**(2), 218–246, (1987).
[4] T. Härder. Implementing a generalized access path structure for a relational database system. *ACM Trans. Database Syst.* **3**(3), 285–298 (1987).
[5] M. J. Carey, D. J. DeWitt and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. ACM SIGMOD Conf. on Management of Data*, Chicago, Ill., pp. 413–423 (1988).
[6] A. Kemper, C. Kilger and G. Moerkotte. Materialization of functions in object bases. In *Proc. ACM SIGMOD Conf. on Management of Data*, Denver, Colo., pp. 258–267 (1991).
[7] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proc. IEEE Int. Workshop on Object-Oriented Database Systems*, (Edited by K. R. Dittrich and U. Dayal) Asilomar, Pacific Grove, Calif., pp. 171–182. IEEE Computer Society Press (1986).
[8] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowl. Data Engng*, **1**(2), 196–214 (1989).
[9] U. Keßler and P. Dadam. Auswertung komplexer Anfragen an hierarchisch strukturierte Objekte mittels Pfadindexen. In *Proc. German Conference on Databases in Office, Engineering and Science (BTW) Informatik-Fachberichte Vol 270*, pp. 218–237. Springer, Berlin (1991).
[10] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proc. ACM SIGMOD Conf. on Management of Data*, Portland, Ore., pp. 325–336 (1989).
[11] M. Stonebraker, J. Anton and E. Hanson. Extending a database system with procedures. *ACM Trans. Database Syst.* **12**(3), 350–376 (1987).
[12] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems* **13**(2), 175–186 (1988).
[13] A. Kemper, G. Moerkotte, H.-D. Walter and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. German Conference on Databases in Office, Engineering and Science (BTW) Informatik-Fachberichte Vol. 270*, Kaiserslautern, pp. 198–217. Springer, Berlin (1991).
[14] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier and S. Zdonik. The object-oriented database system manifesto. In *Proc. of the Int. Conf. on Deductive and Object-Oriented Database (DOOD)*, Kyoto, Japan, pp. 40–57 (1989).
[15] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In *Readings in Object-Oriented Databases*, (Edited by S. Zdonik and D. Maier), pp. 1–32. Morgan-Kaufman (1989).
[16] P. Butterworth, A. Otis and J. Stein. The GemStone object database system. *Commun. ACM* **34**(10), 64–77 (1991).
[17] O. Deux *et al.* The story of O_2. *IEEE Trans. Knowl. Data Engng* **2**(1), 91–108 (1990).
[18] W. Kim, J. F. Garza, N. Ballou and D. Woelk. Architecture of the Orion next-generation database system. *IEEE Trans. Knowl. Data Engng* **2**(1), 109–124 (1990).
[19] C. Lamb, G. Landis, J. Orenstein and D. Weinreb. The ObjectStore database system. *Commun. ACM* **34**(10) 50–63 (1991).
[20] A. Kemper, P. C. Lockemann and M. Wallrath. An object-oriented database system for engineering applications. In *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 299–311, (1987).
[21] M. Stonebraker, E. Wong, P. Kreps and G. Held. The design and implementation of INGRES. *ACM Trans. Database Syst.* **1**(3), 189–222, (1976).
[22] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Conf. on Very Large Data Bases (VLDB)*, Brisbane, Australia, pp. 290–301 (1990).
[23] S. B. Yao. Approximating block accesses in database organizations. *Commun. ACM* **20**(4), 260–261 (1977)
[24] A. Kemper. Zuverlässigkeit und Leistungsfähigket objektorientierter Datenbanken. In *Informatik Fachberichte*. Vol. 298. Springer, Berlin (1992).