Access Support in Object Bases

Alfons Kemper *Guido Moerkotte*

Interner Bericht Nr. 17/89 ∗ Oktober 1989

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 (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 performance analysis of access support relations is developed. This analytical cost model is, in particular, used to determine the best access relation extension and decomposition with respect to specific database configuration and usage characteristics.

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 couple of 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 DBMS 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., [1], 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.

The access support relations described in this paper constitute a generalization of the binary join indices proposed by Valduriez [11]. 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 [6] 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 [5] are contained as a special case 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 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 [8] and for the PostGres model [10, 7] are subsumed by our technique.

The remainder of this paper is organized as follows. Section 2 introduces the Generic Object Model (*GOM*), which serves as the research vehicle for this work, and some simplified application examples to highlight the requirements on object-oriented access support. Then, in section 3 the access support relations are formally defined. In section 4 we begin 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 addresses the maintenance of access support relations due to object updates. In each of the sections 4 through 6 we illustrate the analytical model by some comparative results for characteristic application profiles. Section 7 concludes this paper.

2 The Object Model

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 features that *GOM* provides are relatively *generic* such that the results derived for this particular data model can easily be applied to a variety of other object-oriented models.

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 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 subtyping is based on inheritance. A tuple-structured type t may be defined as the subtype of one (single inheritance) or several (multiple inheritance) other tuple-structured type(s) $t_1, \ldots, t_n$ which means that t inherits all attributes of all supertypes $t_1, \ldots, t_n$.

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.

2.1 Type Definitions

If $s_1, \ldots, s_m, s \in T$, $t \neq ANY$ are type symbols with outer type constructor [], the $a_1, \ldots, a_n$ are pairwise distinct attribute names, and the t_i are types then

type t **is**
  supertypes $(s_1, \ldots, s_m)$
  $[a_1 : t_1, \ldots, a_n : t_n]$

type t **is** $\{s\}$

type t **is** $< s >$

are type definitions.

In the first case the s_i are called *supertypes* of t, and t is called a (direct) *subtype* of s_i. Since the access support on ordered collection, i.e., lists, is analogous to sets we will not elaborate on list-structured types in the remainder of this paper.

2.2 (Engineering) Example Applications

Let us first sketch an engineering application that heavily utilizes tuple-structured types: modeling robots. The following schema constitutes an outline of a robot model:

type ROBOT_SET **is** {ROBOT};
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];

var OurRobots: ROBOT_SET;

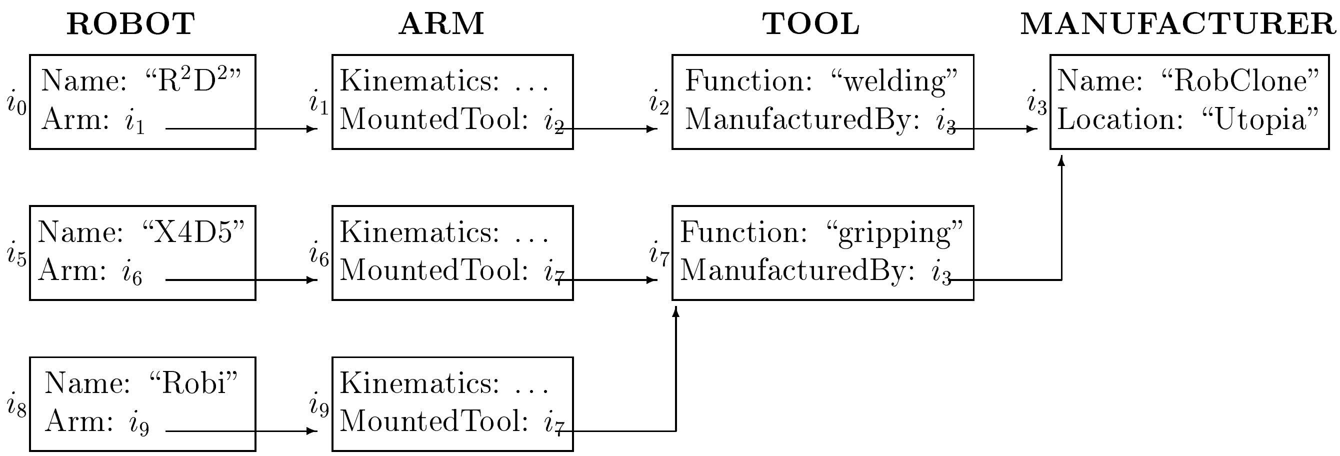

Figure 1: Database Extension with Linear Paths

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*[1] 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 attributes *Name* and *Location*, with the *TOOL* instance.

An extension of such a schema for just three *ROBOT* instances identified by i_0, i_5, and i_8 is graphically depicted in Figure 1. An object instance is a triple (i, v, t) where i denotes the object identifier, v the object value, and t the type of the object. As indicated in Figure 1 references are *uni-directional*, i.e., they are maintained in one direction only. This conforms to (almost) all proposed object models.

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

Query 1: Find the *Robot*s which use a *Tool* manufactured in *“Utopia”*.
Or using SQL-like notation:

> **select** r.Name
> **from** r **in** OurRobots
> **where** r.Arm.MountedTool.ManufacturedBy.Location = “Utopia”

In this example the path expression is r.Arm.MountedTool.ManufacturedBy.Location.

2.3 General Paths (Containing Collection-Valued Attributes)

Note that a linear path 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

[1] not further elaborated here. For more details see [3]

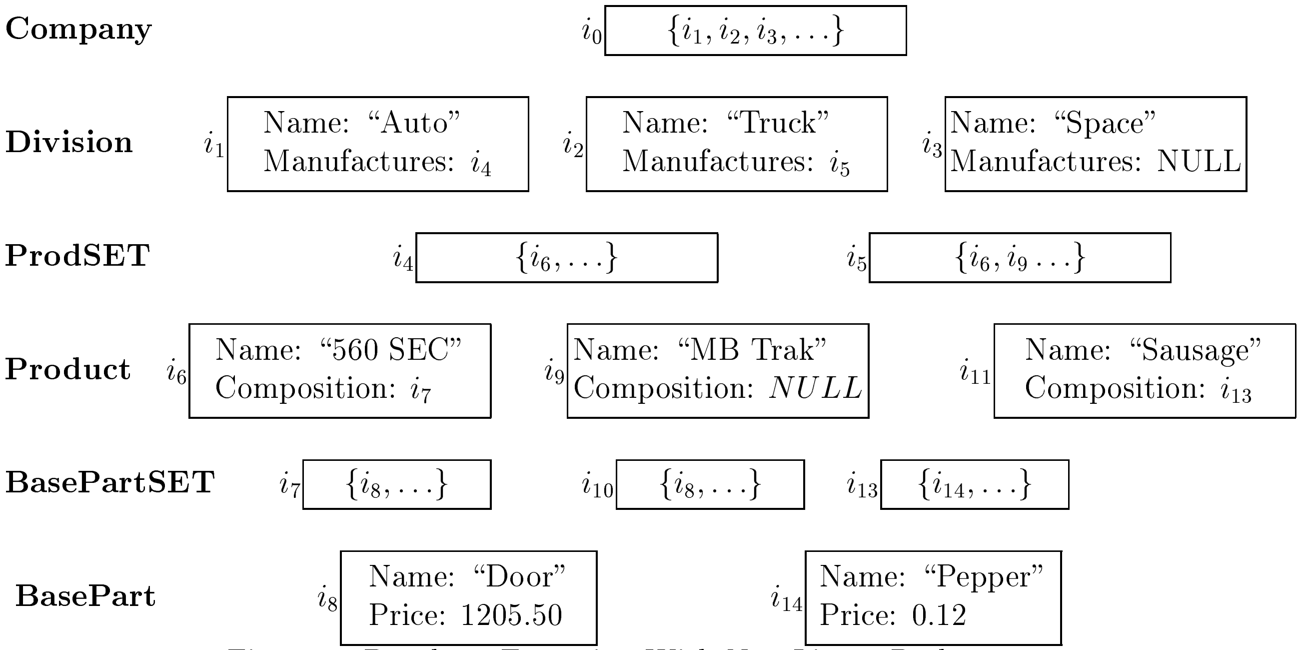

Figure 2: Database Extension With Non-Linear Paths

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 *Products*, which themselves are composed of *BasePart*s.

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;

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

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

Query 2: Which *Division* uses a *BasePart* named "Door"?

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

Query 3: Retrieve all the *BasePart Name*s used by the *Division* named "Auto".

> **select** d.Manufactures.Composition.Name
> **from** d **in** Mercedes.Division
> **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 criterion 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 $t_0, \ldots, t_n$ be (not necessarily distinct) types. A path expression on t_0 is an expression $t_0.A_1.\cdots.A_n$ iff for each $1 \leq i \leq n$ one of the following conditions holds:*

- *The type t_{i-1} is defined as* **type** t_{i-1} **is** $[\ldots, A_i : t_i, \ldots]$.
- *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** $\{t_i\}$. *In this case we speak of a set occurrence at A_i in the path $t_0.A_1.\cdots.A_n$.*

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

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

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

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 of objects of type

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

- 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 the underlying path expression is clear from context we will write E *instead of* $E_{t_0.A_1.\cdots.A_n}$.

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

We distinguish several possibilities for the extension of such relations. To define them for a path expression $t_0.A_1.\cdots.A_n$ we need n auxiliary relations $E_1, \ldots, E_n$.

Definition 3.3 *For each* A_j $(1 \leq j \leq n)$ *we construct the auxiliary 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*

2. *ternary, if* A_j *is a set-valued attribute*

In case (1) 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* $o_{j-1}.A_j = o_j$[3].

In case (2) 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 2. For the underlying schema we could declare the access support relation on the path expression *Division.Manufactures.Composi*
This results in 3 auxiliary 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"
...	...

□

[3]If t_j is an atomic type then $id(o_j)$ corresponds to the value $o_{j-1}.A_j$.

Let us now introduce different possible extensions of the access support relation E. The first extension, called the *canonical extension*, is the obvious one. It contains only information about complete paths spanning the attribute chain $t_0.A_1.\cdots.A_n$. Let us illustrate the canonical extension on a linear path. Here for all objects o_0 in t_0, o_1 in t_1, ..., which fulfill $o_0.A_1 = o_1$, ...,$o_0.A_1.\cdots.A_n = o_n$ the canonical extension, denoted E_{can}, of the access support relation E contains the tuple $(id(o_0), \ldots, id(o_n))$.

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.

Definition 3.4 (Canonical Extension) *Let $t_0.A_1.\cdots.A_n$ be a path expression. The canonical extension E_{can} is defined as*

$$E_{can} := E_0 \bowtie \ldots \bowtie E_{n-1}$$

□

The canonical extension contains only complete paths in the sense that every tuple represents the attribute values for an object o in t_0 for which $o.A_1.\cdots.A_n$ exists, i.e., there is no NULL value somewhere along the path. This is the minimum information that must be contained within the access relation in order to allow access support for all queries spanning the whole attribute chain.

Example: For our example auxiliary relations E_0, E_1, and E_2 we obtain the following canonical extension E_{can}:

E_{can}					
$OID_{Division}$	$OID_{ProdSET}$	$OID_{Product}$	$OID_{BasePartSET}$	$OID_{BasePart}$	$VALUE_{Name}$
...	...	...	...	...	...
i_1	i_4	i_6	i_7	i_8	"Door"
...	...	...	...	...	...

Note that E_{can} contains only complete paths originating in t_0 and leading to t_n. But there could also be more information in the extension of an access support relation. For a path $t_0.A_1.\cdots.A_n$ consider the case where all the information concerning the attribute value of A_1 of every object in t_0, and the attribute values of A_i of every object of type t_{i-1} is contained in the access support relations. This is, naturally, the maximum information concerning the access path. The extension containing all this information is defined next.

Definition 3.5 (Full Extension) *Be $t_0.A_1.\cdots.A_n$ a path expression. The full extension E_{full} is defined as*

$$E_{full} := E_0 ⟗ \ldots ⟗ E_{n-1}$$

□

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 below) or those not originating in an object o_0 of type t_0 (the second tuple in E_{full} shown below). Even partial paths not originating in t_0 and leading to a NULL are to be included

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"
...	...	...	...	...	...

Obviously there are many intermediate forms between these two cases. We will restrict our discussion to *left-* and *right-complete* extensions.

Definition 3.6 (Left-complete Extension) *Be $t_0.A_1.\cdots.A_n$ a path expression. The left-complete extension E_{left} is defined as*

$$E_{left} := (\ldots(E_0 ⟕ E_1) ⟕ \ldots ⟕ E_{n-1})$$

□

Example: The left-complete extension contains all those (partial) paths that originate in some o_0 of type t_0 (even if the path eventually leads to a NULL as, e.g., in the first tuple below).

E_{left}					
$OID_{Division}$	$OID_{ProdSET}$	$OID_{Product}$	$OID_{BasePartSET}$	$OID_{BasePart}$	$VALUE_{Name}$
...	...	...	...	...	...
i_2	i_5	i_9	NULL	NULL	NULL
i_1	i_4	i_6	i_7	i_8	"Door"
...	...	...	...	...	...

Definition 3.7 (Right-complete Extension) *Be $t_0.A_1.\cdots.A_n$ a path expression. The right-complete extension E_{right} is defined as*

$$E_{right} := (E_0 ⟖ (\ldots ⟖ (E_{n-2} ⟖ E_{n-1}) \ldots)$$

□

Example: The right-complete extension contains all (partial) paths that are at least defined for the attribute A_n in some object o_{n-1} of type t_{n-1}. The path, however, need not necessarily originate in t_0, as exemplified by the first tuple in the extension shown below:

E_{right}					
$OID_{Division}$	$OID_{ProdSET}$	$OID_{Product}$	$OID_{BasePartSET}$	$OID_{BasePart}$	$VALUE_{Name}$
...	...	...	...	...	...
NULL	NULL	i_{11}	i_{13}	i_{14}	"Pepper"
i_1	i_4	i_6	i_7	i_8	"Door"
...	...	...	...	...	...

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 3.8 (Decomposition) *Let R be an $(m+1)$-ary relation with attribute $S_0, \ldots, S_m$. Then the relations*

$$\begin{array}{lll} R^{0,i_1}: & [S_0, \ldots, S_{i_1}] & \textit{for } 0 < i_1 \leq m \\ R^{i_1,i_2}: & [S_{i_1}, \ldots, S_{i_2}] & \textit{for } i_1 < i_2 \leq m \\ & \ldots & \\ R^{i_k,m}: & [S_{i_k}, \ldots, S_m] & \textit{for } i_k < m \end{array}$$

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

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

Example: For our example the binary decomposition consisting of five relations of the canonical extension is shown below:

$E^{0,1}_{can}$	
$OID_{Division}$	$OID_{ProdSET}$
i_1	i_4
...	...

$E^{1,2}_{can}$	
$OID_{ProdSET}$	$OID_{Product}$
i_4	i_6
...	...

$E^{2,3}_{can}$	
$OID_{Product}$	$OID_{BasePartSET}$
i_6	i_7
...	...

$E^{3,4}_{can}$	
$OID_{BasePartSET}$	$OID_{BasePart}$
i_7	i_8
...	...

$E^{4,5}_{can}$	
$OID_{BasePart}$	$VALUE_{Name}$
i_8	"Door"
...	...

4 Analytical Cost Model: Cardinality of Access Relations

In this section we start the development of an analytical cost model to evaluate the access relation concept. 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 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 relations in various extensions and decompositions.

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.

4.1.1 Some Derived Quantities

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

$$P_{A_i} = \frac{d_i}{c_i} \tag{1}$$

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:

$$P_{H_i} = \frac{e_i}{c_i} \tag{2}$$

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, may be approximated as

$$\left(1 - \frac{1}{e_{i+1}}\right)^{fan_i} \tag{3}$$

application-specific parameters		
parameter	semantics	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 $NULL$	
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	
$shar_i$	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, a normal distribution of references from objects in t_i to objects in t_{i+1} is assumed. In this case $shar_i$ is derived as shown on the right.	$shar_i = \frac{d_i * fan_i}{c_{i+1}}$
e_i	the number of objects in t_i which are referenced by an object in t_{i-1}	$e_i = \frac{d_{i-1} * fan_{i-1}}{shar_{i-1}}$
$spread_i$	the relation between the number of defined objects of type t_i and the referenced objects of type t_{i+1}	$spread_i = \frac{d_i}{e_{i+1}}$
ref_i	the number of references of objects of type t_j	$ref_i = d_i * fan_i$
$size_i$	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	$PPsize = 4$
B^+_{fan}	fan out of the B^+ tree	$\left\lfloor \frac{PageSize}{PPsize + OIDsize} \right\rfloor$

Figure 3: System and Application Parameters

However, this formula contains a slight error: it assumes that all fan_i references are independent—which is not the case when no two references emanating from the one object in t_i can hit the same object in t_{i+1}. This error manifests itself for large fan_i values and correspondingly small e_{i+1} values.

Therefore, a better approximation 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:

$$\frac{\binom{e_{i+1}-1}{fan_i}}{\binom{e_{i+1}}{fan_i}} = \frac{e_{i+1} - fan_i}{e_{i+1}} = 1 - \frac{fan_i}{e_{i+1}} \tag{4}$$

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

$$\left(1 - \frac{fan_i}{e_{i+1}}\right)^k \tag{5}$$

For $0 \leq i < j \leq n$ we now define $RefBy(i, j)$ which denotes the number of objects in t_j which are referenced by some object in t_i (via at least one (partial) path):

$$RefBy(i,j) = \begin{cases} e_{i+1} & j = i + 1 \\ e_j * \left(1 - \left(1 - \frac{fan_{j-1}}{e_j}\right)^{RefBy(i,j-1)*P_{A_{j-1}}}\right) & \text{else} \end{cases} \tag{6}$$

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

$$P_{RefBy}(i,j) = \begin{cases} 1 & i = j \\ \frac{RefBy(i,j)}{c_j} & \text{else} \end{cases} \tag{7}$$

Let $Ref(i, j)$ denote the number of objects of type t_i which have a path leading to some object of type t_j for $0 \leq i < j \leq n$. This value can be approximated as:

$$Ref(i,j) = \begin{cases} d_i & j = i + 1 \\ d_i * \left(1 - \left(1 - \frac{shar_i}{d_i}\right)^{Ref(i+1,j)*P_{H_{i+1}}}\right) & \text{else} \end{cases} \tag{8}$$

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

$$P_{Ref}(i,j) := \begin{cases} 1 & i = j \\ \frac{Ref(i,j)}{c_i} & \text{else} \end{cases} \tag{9}$$

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

$$path(i,j) = ref_i * \prod_{l=i+1}^{j-1} (P_{A_l} * fan_l) \tag{10}$$

4.2 Cardinalities of Access Support Relations

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

4.2.1 Canonical Extension

No Decomposition In this special case of no decomposition the number of tuples, $\#E_{can}$ in the access relation E_{can} is given as:

$$\#E_{can} = path(0,n)$$

General Decomposition For a general decomposition $(\ldots, i, j, \ldots)$ the indicated part $E_{can}^{i,j}$ of the decomposition contains the following number of tuples:

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

4.2.2 Full Extension

General Decomposition 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 $0 \le i < j \le n$:

$$P_{lb}(i,j) = \begin{cases} 1 - P_{RefBy}(i,j) & i < j \\ 1 & \text{else} \end{cases} \tag{11}$$

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 $0 \le i < j \le n$:

$$P_{rb}(i,j) = \begin{cases} 1 - P_{Ref}(i,j) & i < j \\ 1 & \text{else} \end{cases} \tag{12}$$

Using these quantities we can then estimate that the relation $E_{full}^{i,j}$ contains the following number of tuples:

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

[4] *lb*: left-bound
[5] *rb*: right-bound

4.2.3 Left-complete Extension

The relation $E^{i,j}_{left}$ which holds all the paths from t_i to t_j which are left-complete, i.e., which originate in t_0, has the following cardinality:

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

4.2.4 Right-complete Extension

Finally, the cardinality of the $(\dots,i,j,\dots)$ partition of the right-complete access support relation is derived as:

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

4.3 Storage Costs for Access Relations

Let X denote an extension of the access relation E, i.e., $X \in \{can, full, left, right\}$.

The size of a tuple in the access relation $E^{i,j}_X$ in bytes is:

$$ats^{i,j} = OIDsize * (j-i+1) \tag{13}$$

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

$$atpp^{i,j} = \left\lfloor \frac{PageSize}{ats^{i,j}} \right\rfloor \tag{14}$$

The size of the access relation $E^{i,j}_X$ in bytes:

$$as^{i,j}_X = \#E^{i,j}_X * ats^{i,j} \tag{15}$$

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

$$ap^{i,j}_X = \left\lceil \frac{\#E^{i,j}_X}{atpp^{i,j}} \right\rceil \tag{16}$$

4.4 Some Sample Results

Subsequently, we graphically demonstrate two results for—in our view—typical engineering application characteristics. However, the reader should bear in mind that the size comparison of different access 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.

4.4.1 Comparison between Extensions and Decompositions

In this experiment we want to compare different extensions and decompositions of the access relation size for a fixed application characterization, which is listed in the table below:

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	1000	5000	10000	50000	100000
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	900	4000	8000	20000	—
fan-out	f_0	f_1	f_2	f_3	f_4
	2	2	3	4	—

The comparison of storage costs (for non-redundant representation) is graphically plotted in Figure 4

Figure 4: Comparison of Access Relation Sizes

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.

4.4.2 Varying all d_i Parameters

In the subsequent experiment we want to demonstrate the effect of varying the number of defined attributes, i.e., varying d_i for $(0 \leq i \leq 3)$, while keeping the number of objects and the fan-out fixed.

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	10000	10000	10000	10000	10000
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	$2500 \cdots 10^4$	$2500 \cdots 10^4$	$2500 \cdots 10^4$	$2500 \cdots 10^4$	—
fan-out	f_0	f_1	f_2	f_3	f_4
	2	2	2	2	—

The parameters d_0, d_1, d_2, d_3 were simultaneously increased, i.e., the values are kept identical. The plot in Figure 5 shows the access relation sizes for all different extensions under no decomposition.

Figure 5: Varying the Number of Not-NULL Attributes

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 $o \in C$ are retrieved, where C is a collection of t_0 instances, based on the membership of some other object o_n of type t_n in the path expression $o.A_1.\cdots.A_n$.

$Q^{i,j}(bw) :=$ **select** o
from o **in** C /* C is some collection of t_i instances */
where o_j **in** $o.A_{i+1}.\cdots.A_j$

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 of type t_i.

$Q^{i,j}(fw) :=$ **select** $o.A_{i+1}.\cdots.A_j$
from o **in** C /* C is some collection of t_i instances */
where ...

5.2 Storage Representation of Access Support Relations

Following the proposal of 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.

5.3 Query Evaluation

Canonical Extension The canonical extension of an access support relation over a path expression $o.A_1.A_2.\cdots.A_n$ is only useful for evaluating full paths of the form:

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

where o ranges over a collection of t_0 instances.

The canonical extension cannot be used to evaluate an expression of the form $o.A_1.\cdots.A_j$ where $j < n$ or of the form $o'.A_j.\cdots.A_n$ where $j > 1$ and o' ranges over a collection of t_{j-1} instances.

Right-Complete Extension The right-complete extension of the access support relation can be utilized to evaluate path expressions of the form:

$$t_0.A_{j+1}.\cdots.A_n$$

where $0 \leq j$ and o ranges over a collection of t_j instances.

Left-Complete Extension The left-complete extension is utilized for any path expression originating in t_0, i.e.:

$$o.A_0.\cdots.A_j$$

for $j \leq n$ and o ranges over a collection of t_0 instances.

Full Extension Finally, the full extension may be used to evaluate any path of the form

$$o.A_{i+1}.\cdots.A_j$$

for $0 \leq i < j \leq n$ and o ranging over a collection of t_i instances.

Before we start developing the cost model, we would like to give an extended remark on the sharing of access support relations.

5.4 Sharing of Access Support Relations

Consider the following two path expressions:

$$t_0.A_1.\cdots.A_i.A_{i+1}.\cdots.A_{i+j}.A_{i+j+1}.\cdots.A_n \quad (1)$$
$$t'_0.A'_1.\cdots.A'_{i'}.A_{i+1}.\cdots.A_{i+j}.A'_{i+j+1}.\cdots.A'_{n'} \quad (2)$$

If $t_0.A_1.\cdots.A_i$ and $t'_0.A'_1.\cdots.A'_{i'}$ are path expressions both leading to objects of type t_i then part of the access support relations may be shared.

This, in general, is only possible when a *full* extension of the access support relation is maintained. Let E_{full} be the full extension for the path (1), and $\bar{E}_{full}$ the full extension of the access support relations for path (2). Then the decomposition $(0, i, i+j, n)$ of E and $(0, i', i'+j, n)$ of $\bar{E}$ share a common partition, i.e., $E_{full}^{i,i+j} = \bar{E}_{full}^{i',i'+j}$.

Thus we obtain the following five partitions:

$E_{full}^{0,i} : [OID_{t_0}, \ldots, OID_{t_i}]$ $\quad$ $\bar{E}_{full}^{0,i'} : [OID_{t'_0}, \ldots, OID_{t_i}]$
$E_{full}^{i,i+j} = \bar{E}_{full}^{i',i'+j} : [OID_{t_i}, \ldots, OID_{t_{i+j}}]$
$E_{full}^{i+j,n} : [OID_{t_{i+j}}, \ldots, OID_{t_n}]$ $\quad$ $\bar{E}_{full}^{i'+j,n'} : [OID_{t_{i+j}}, \ldots, OID_{t'_{n'}}]$

The five partitions may then, individually, be further decomposed.

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

- if both paths (1) and (2) originate in t_0 <, i.e., $i = i' = 1$ then the sharing is also possible for left-complete extensions.
- if both paths lead to t_n, i.e., $i + j = i' + j = n = n'$, then the corresponding partition of the right-complete extensions may be shared.

This should indicate that there may exist a higher level of organization of access support relations which constrains the possible extensions or decompositions.

5.5 Preliminaries for the Cost Estimation

In the subsequent work we will frequently use the following variables. Let X denote the extension of some access relation, i.e., $X \in \{can, full, left, right\}$. The variables i and j denote some intermediate types in the path expression $t_0.A_1.\cdots.A_n$ such that $0 \leq i < j \leq n$.

The number of objects of type t_i per page:

$$opp_i = \left\lfloor \frac{PageSize}{size_i} \right\rfloor \tag{17}$$

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:

$$op_i = \left\lceil \frac{c_i}{opp_i} \right\rceil \tag{18}$$

The height of the B^+ tree—not considering the leaves—for the relation $E_X^{i,j}$:

$$ht_X^{i,j} = \left\lceil \log_{B^+_{fan}}(ap_{ext}^{i,j}) \right\rceil \tag{19}$$

The number of pages (without leaves) in the B^+ tree for the relation $E_X^{i,j}$ is computed as:

$$pg_X^{i,j} = \begin{cases} ht_X^{i,j} & ht_X^{i,j} \leq 1 \\ 1 + \left\lceil \frac{ap_X^{i,j}}{B^+_{fan}} \right\rceil & ht_X^{i,j} = 2 \end{cases} \tag{20}$$

The number of leave pages of the B^+ tree per value in the access relation depends clearly on the extension. They can be estimated as follows:

$$nlp_{full}^{i,j} = \left\lceil \frac{as_{full}^{i,j}}{PageSize * d_i} \right\rceil \tag{21}$$

$$nlp^{i,j}_{right} = \left\lceil \frac{as^{i,j}_{right}}{PageSize * d_i} \right\rceil \tag{22}$$

$$nlp^{i,j}_{can} = \left\lceil \frac{as^{i,j}_{can}}{PageSize * ref(i,n) * P_{RefBy}(0,i)} \right\rceil \tag{23}$$

$$nlp^{i,j}_{left} = \left\lceil \frac{as^{i,j}_{left}}{PageSize * RefBy(0,i)} \right\rceil \tag{24}$$

For the B^+ tree for the inverse clustered access relation we have:

$$Rnlp^{i,j}_{full} = \left\lceil \frac{as^{i,j}_{full}}{PageSize * e_i} \right\rceil \tag{25}$$

$$Rnlp^{i,j}_{left} = \left\lceil \frac{as^{i,j}_{right}}{PageSize * e_i} \right\rceil \tag{26}$$

$$Rnlp^{i,j}_{can} = \left\lceil \frac{as^{i,j}_{can}}{PageSize * ref(j,n) * P_{RefBy}(0,j)} \right\rceil \tag{27}$$

$$Rnlp^{i,j}_{right} = \left\lceil \frac{as^{i,j}_{right}}{PageSize * Ref(j,n)} \right\rceil \tag{28}$$

5.6 Query Cost: No Access Support Relation

In estimating the query evaluation cost we will neglect the CPU cost and merely compare the number of page accesses on secondary storage. In the following cost model we will frequently use a well-known formula. Yao [13] 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:

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

We extend the definitions of *RefBy* and *Ref* supplied in (6) and (8). For $0 \leq i < j \leq n$ and $0 \leq k$ we define the three argument function $RefBy(i,j,k)$ 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:

$$RefBy(i,j,k) = \begin{cases} e_{i+1} * \left(1 - \left(1 - \frac{fan_i}{e_{i+1}}\right)^k\right) & j = i+1 \\ e_j * \left(1 - \left(1 - \frac{fan_{j-1}}{e_j}\right)^{RefBy(i,j-1,k)*P_{A_{j-1}}}\right) & \text{else} \end{cases} \tag{29}$$

Analogously, let $Ref(i,j,k)$ denote the number of objects of type t_i which have a path leading to some object of a k-element subset of type t_j for $0 \leq i < j \leq n$ and $0 \leq k$. This value can be derived as:

$$Ref(i,j,k) = \begin{cases} d_i * \left(1 - \left(1 - \frac{shar_i}{d_i}\right)^k\right) & j = i+1 \\ d_i * \left(1 - \left(1 - \frac{shar_i}{d_i}\right)^{Ref(i+1,j,k)*P_{H_{i+1}}}\right) & \text{else} \end{cases} \tag{30}$$

If the object references are only stored within the object representation the best possible algorithm without any access support structures has to inspect every page containing a referenced object at least once.

5.6.1 Forward Query

$$Qnas^{i,j}(fw) = 1 + \sum_{l=i+1}^{j-1} y(\lceil RefBy(i,l,1)\rceil, op_l, c_l) \tag{31}$$

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.

5.6.2 Backward Query

$$Qnas^{i,j}(bw) = op_i + \sum_{l=i+1}^{j-1} y(\lceil RefBy(i,l,d_i)\rceil, op_l, c_l) \tag{32}$$

Basically, the backward query is evaluated by an exhaustive search. All objects of type $t_l (i < l < j)$ 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.

5.7 Query Cost: With Access Support Relation

5.7.1 Forward Query

The cost for a supported forward query can be calculated as follows:

$$\begin{aligned} Qsup_X^{i,j}(fw, dec) \;&=\; \sum_{\substack{i_\alpha, i_{\alpha+1} \in dec \\ (i_\alpha = i < i_{\alpha+1})}} \left(ht_X^{i_\alpha, i_{\alpha+1}} + nlp_X^{i_\alpha, i_{\alpha+1}}\right) + \sum_{\substack{i_\alpha, i_{\alpha+1} \in dec \\ (i_\alpha < i < i_{\alpha+1})}} \left(ap_X^{i_\alpha, i_{\alpha+1}}\right) \\ &+ \sum_{\substack{i_\alpha, i_{\alpha+1} \in dec \\ (i < i_\alpha < j)}} \Big(1.0 \;+\; y(\lceil RefBy(i,i_\alpha,1)\rceil, pg_X^{i_\alpha,i_{\alpha+1}} - 1, (pg_X^{i_\alpha,i_{\alpha+1}} - 1) * B_{fan}^{+}) \\ &\qquad +\; y(\lceil RefBy(i,i_\alpha,1)\rceil * nlp_X^{i_\alpha,i_{\alpha+1}}, ap_X^{i_\alpha,i_{\alpha+1}}, \#E_X^{i_\alpha,i_{\alpha+1}})\Big) \end{aligned} \tag{33}$$

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 $0 \leq \alpha < k$. In this case only one path through the B^+ tree has to be traversed and the leave pages for one value $(nlp_X^{i_\alpha,i_{\alpha+1}})$ 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 relation partition $E_X^{i_\alpha,i_{\alpha+1}}$ that covers i have to be inspected. This number equals $ap_X^{i_\alpha,i_{\alpha+1}}$.

Finally, the third sum accounts for accessing the partitions that lead to j. Within each partition $(i_\alpha, i_{\alpha+1})$, 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}
- the data pages of the access relation partition $E_X^{i_\alpha,i_{\alpha+1}}$ that contain the $RefBy(i, i_\alpha, 1)$ object identifiers of type t_{i_α}

5.7.2 Backward Query

The cost for a supported backward query can be calculated as follows:

$$\begin{aligned} Qsup_X^{i,j}(bw, dec) \;=\; & \sum_{\substack{i_\alpha,i_{\alpha+1}\in dec\\(i_\alpha<j=i_{\alpha+1})}} \left(ht_X^{i_\alpha,i_{\alpha+1}} + Rnlp_X^{i_\alpha,i_{\alpha+1}}\right) + \sum_{\substack{i_\alpha,i_{\alpha+1}\in dec\\(i_\alpha<j<i_{\alpha+1})}} \left(ap_X^{i_\alpha,i_{\alpha+1}}\right) \\ + \sum_{\substack{i_\alpha,i_{\alpha+1}\in dec\\(i<i_{\alpha+1}<j)}} \Big(1.0 \;&+\; y(\lceil Ref(i_{\alpha+1}, j, 1)\rceil, pg_X^{i_\alpha,i_{\alpha+1}} - 1, (pg_X^{i_\alpha,i_{\alpha+1}} - 1) * B^+_{fan}) \\ &+\; y(\lceil Ref(i_{\alpha+1}, j, 1)\rceil * Rnlp_X^{i_\alpha,i_{\alpha+1}}, ap_X^{i_\alpha,i_{\alpha+1}}, \#E_X^{i_\alpha,i_{\alpha+1}})\Big) \qquad (34) \end{aligned}$$

The cost for evaluating a supported backward query is derived analogously to a forward query. The major distinction is, that now the reverse clustered access relation is used.

5.8 General Formula for Query Cost

Given the query costs for a supported query and for a non supported query the costs for the different cases can be calculated as follows:

$$Q_X^{i,j}(kind, dec) = \begin{cases} Qsup_X^{i,j}(kind, dec) & i = 0 \wedge j = n \quad X = can \\ Qnas^{i,j}(kind) & i \neq 0 \vee j \neq n \quad X = can \\ Qsup_X^{i,j}(kind, dec) & \qquad\qquad\quad\; X = full \\ Qsup_X^{i,j}(kind, dec) & i = 0 \qquad\quad\;\; X = left \\ Qnas^{i,j}(kind) & i \neq 0 \qquad\quad\;\; X = left \\ Qsup_X^{i,j}(kind, dec) & j = n \qquad\quad\;\; X = right \\ Qnas^{i,j}(kind) & j \neq n \qquad\quad\;\; X = right \end{cases} \tag{35}$$

Again, the parameters have the following meaning: $kind \in \{fw, bw\}$, the parameter X denotes the chosen extension, i.e., $X \in \{can, full, left, right\}$. The parameter dec denotes the (chosen) decomposition of the access relation, and $0 \leq i < j \leq n$.

5.9 Sample Results

5.9.1 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 below:

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	100	500	1000	5000	10000
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	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 is lower than for binary decomposed relations.

Figure 6: Query Costs for a Backward Query

5.9.2 Query Costs Depending on Object Size

Figure 7 visualizes the cost of a backward query of the form $Q^{0,4}(bw)$ depending on the size of the stored data, i.e., the parameter $size_i$ is varied for $(0 \leq i \leq 4)$:

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	100	500	1000	5000	10000
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	90	400	8000	2000	—
fan-out	f_0	f_1	f_2	f_3	f_4
	2	2	3	4	—
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$
	$100 \cdots 800$	$100 \cdots 800$	$100 \cdots 800$	$100 \cdots 800$	$100 \cdots 800$

The access support relations are decomposed into binary partitions. As can be seen in Figure 7 the object size does not influence the query cost for supported queries (as expected). Only the cost of non-supported queries grows proportional to the object size. Note, that in Figure 7 the values for full, left, and right extensions overlap (marked with filled squares.

diagquvsb

Figure 7: Query Costs for a Backward Query Under Varying Object Size

5.9.3 Which Queries are Supported?

As described before, not all queries are supported by certain extensions of the access relation. Also, the decomposition of the access relations has a major effect on the cost of a query. For demonstration, let us use the following application characteristics:

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	10^4	10^4	10^4	10^4	10^4
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	$10 \cdots 10^4$	$10 \cdots 10^4$	$10 \cdots 10^4$	$10 \cdots 10^4$	—
fan-out	f_0	f_1	f_2	f_3	f_4
	2	2	2	2	—
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$
	120	120	120	120	120

The plot in Figure 8 shows the query costs of a backward query of the form: $Q^{0,3}(bw)$. We computed the results for two decompositions: (1) decomposition into binary partitions and (2) 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.

Figure 8: Query Costs for a Backward Query $Q^{0,3}(bw)$

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 being that the rather large access support relations have to be exhaustively searched under no decomposition, i.e., all pages have to be inspected.

5.9.4 An Application Favoring Canonical/Left over Full/Right

The following parameters describe an application that favors canonical and left-complete extensions over full and right-complete extensions of the access relation.

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	400000	400000	400000	400000	400000
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	10	100	1000	100000	—
fan-out	f_0	f_1	f_2	f_3	f_4
	$10 \cdots 100$	$10 \cdots 100$	$10 \cdots 100$	$10 \cdots 100$	—
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$
	120	120	120	120	120

The query costs for varying fan-out values are plotted in Figure 9.

graph0

Figure 9: Cost of a backward Query $Q^{0,4}(bw)$

6 Maintenance of Predicate Extensions

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

In order to simplify the subsequent discussion we consider only one special—yet characteristic—type of update operation: inserting an object into a set-valued attribute. This operation, denoted as ins^i, could be phrased in our pseudo-SQL language as follows:

$$ins^i := \textbf{insert } o \textbf{ into } o_i.A_i$$

We assume that the object o_i is of type t_i.

Let us now analyze the effect of this insertion 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 $0 \leq k, i \leq n$,$i \neq k$ either o_i is not of type t_k 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 has to be of type t_{i+1}.

The update costs consist of three parts:

1. the costs for updating the object o_i
2. searching the identifiers for the paths $(\cdots, id(o_i), id(o), \cdots)$ that have to be updated, and
3. updating the access support relations.

The cost for updating $o_i.A_i$ amounts to 3, i.e., one page access to retrieve the object representation of o_i and one page access to write the object o_i back to secondary storage.

6.1 Searching for the New Paths

The update of the access support relations involves the following two auxiliary relations

$$\begin{aligned} I_l &:= \{(NULL, \ldots, NULL, i_k, \ldots, id(o_i)) \mid k < i \\ &\qquad\qquad \text{and no object in } t_{k-1} \text{ references } i_k \text{ or } k = 0\} \\ I_r &:= \{(id(o), \ldots, i_s, NULL, \ldots, NULL) \mid s > i + 1 \\ &\qquad\qquad \text{and the } A_s \text{ attribute of } i_s \text{ is NULL or } s = n\}, \end{aligned}$$

Let I_l^0 denote the relation defined analogously to I_l except that $k = 0$, i.e., all paths originate in t_0. Analogously, I_r^n is defined under the condition $s = n$, i.e., considering only paths that lead to t_n.

The next step consists of materializing the relations I_l and I_r, 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 I_r and I_l cannot be materialized from the access 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 relations.

If we have a left-complete extension we have to search the paths from object o in direction t_n to materialize I_r. But this is only necessary if o_i is referenced by some object in t_0, and o_j is not already contained in the access relation, i.e. not yet referenced by some path originating in an object in t_0. Otherwise, I_r 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 I_l is only needed if o was already present in the access support relation and if o_i is absent. Only under this condition one (or more) new right-complete paths have to be added to the access relations.

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 I_r^n. The forward search from o to t_n has only to be performed if there does not already exist a complete path through o. We start the backward search to materialize I_l^0 only if we have found a connection from o to t_n. The backward search itself is only necessary if there does not already exist a complete path through o_i. Thus the total search costs for the different extensions can be estimated by:

$$search_X^i = \begin{cases} Qnas^{i+1,n}(fw) * P_{NoPath}(i+1) + Qsup^{i,i+1}(bw, dec) & \\ \quad + Qnas^{0,i}(bw) * P_{Ref}(i+1, n) * P_{NoPath}(i) + Qsup^{i,i+1}(fw, dec) & \text{for } X = can \\ min(Qsup^{i,i+1}(fw, dec), Qsup^{i,i+1}(bw, dec)) & \text{for } X = full \\ Qnas^{i+1,n}(fw) * (1 - P_{RefBy}(0, i+1)) * P_{RefBy}(0, i) & \\ \quad + min(Qsup^{i,i+1}(fw, dec), Qsup^{i,i+1}(bw, dec)) & \text{for } X = left \\ (\sum_{l=0}^{i} op_l) * (1 - P_{Ref}(i, n)) * P_{Ref}(i+1, n) & \\ \quad + min(Qsup^{i,i+1}(fw, dec), Qsup^{i,i+1}(bw, dec)) & \text{for } X = right \end{cases} \tag{36}$$

Here, $P_{NoPath}(l)$ denotes the probability that no complete path exists, that leads through a particular object o_l of type t_l. This value is computed as:

$$P_{NoPath}(l) = 1 - P_{Path}(l) \tag{37}$$

$$P_{Path}(l) = P_{RefBy}(0, l) * P_{Ref}(l, n) \tag{38}$$

For $X = left$ or $X = right$ we have to perform two queries in order to find out whether a search is necessary. Since both queries are within the same access relation we can use the maximum as the total cost for answering both queries.

6.2 Updating the Access Support Relations

Next we have to consider the cost of updating the access support relation (partitions). The general formula is given below:

$$aup_X^i(dec) = \sum_{(i_\alpha, i_{\alpha+1}) \in dec} \Big(1 + y(qfw_X^i(i_\alpha, i_{\alpha+1}), pg_X^{i_\alpha, i_{\alpha+1}} - 1, (pg_X^{i_\alpha, i_{\alpha+1}} - 1) * B_{fan}^+)$$

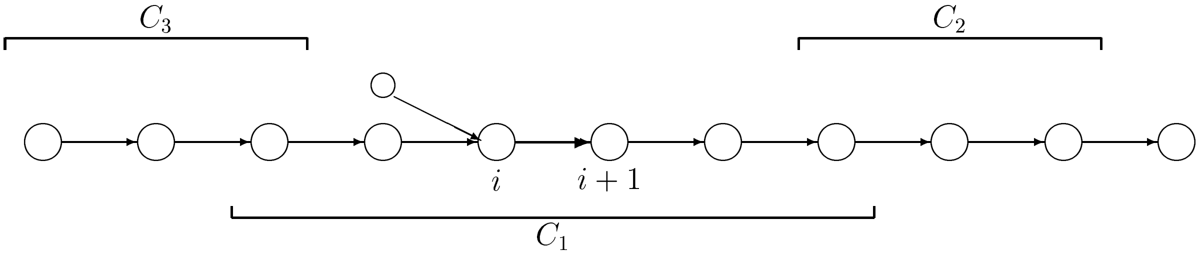

Figure 10: Different Partitions of Access Relations w.r.t. ins^i

$$
\begin{array}{rl}
& + \quad y(qfw^i_X(i_\alpha, i_{\alpha+1}), ap_X^{i_\alpha, i_{\alpha+1}}, \#E_X^{i_\alpha, i_{\alpha+1}}) * 2 \\
+1 & + \quad y(qbw^i_X(i_\alpha, i_{\alpha+1}), pg_X^{i_\alpha, i_{\alpha+1}} - 1, (pg_X^{i_\alpha, i_{\alpha+1}} - 1) * B^+_{fan}) \\
& + \quad y(qbw^i_X(i_\alpha, i_{\alpha+1}), ap_X^{i_\alpha, i_{\alpha+1}}, \#E_X^{i_\alpha, i_{\alpha+1}}) * 2\Big)
\end{array}
$$

In this formula, the first summand constitutes the cost for accessing the non-leaf pages of the forward clustered B^+ tree. The second summand accounts for the cost of accessing and writing back the leaf pages—therefore, the factor 2. Altogether, $qfw^i_X(i_\alpha, i_{\alpha+1})$ clusters have to be updated, where a cluster is a collection of paths with identical first object. The formulas for $qfw^i_X(i_\alpha, i_{\alpha+1})$ are given below. In this cost estimation we made two simplifying assumptions:

- a cluster fits on one page
- page overflows of leaf or non-leaf pages of the B^+ tree do not occur

The third and fourth summand are analogous for the backward clustered B^+ tree. Here the number of clusters to be dealt with is denoted $qbw^i_X(i_\alpha, i_{\alpha+1})$

Let us now derive the formulas for estimating the number of clusters that have to be updated within the partition $(i_\alpha, i_{\alpha+1})$ of the extension X with respect to the operation ins^i.

6.2.1 Number of Clusters under Canonical Extension

$$
qfw^i_{can}(i_\alpha, i_{\alpha+1}) = \begin{cases} Ref(i_\alpha, i, 1) * P_{RefBy}(0, i_\alpha) * P_{Ref}(i+1, n) & i_\alpha \leq i \\ RefBy(i+1, i_\alpha, 1) * P_{RefBy}(0, i) * P_{Ref}(i_\alpha, n) & i < i_\alpha \end{cases}
$$

$$
qbw^i_{can}(i_\alpha, i_{\alpha+1}) = \begin{cases} Ref(i_{\alpha+1}, i, 1) * P_{RefBy}(0, i_{\alpha+1}) * P_{Ref}(i+1, n) & i_{\alpha+1} \leq i \\ RefBy(i+1, i_{\alpha+1}, 1) * P_{RefBy}(0, i) * P_{Ref}(i_{\alpha+1}, n) & i < i_{\alpha+1} \end{cases}
$$

Let us focus on the formula for $qfw^i_{can}(i_\alpha, i_{\alpha+1})$. We consider two cases, depending on where the partition $(i_\alpha, i_{\alpha+1})$ lies relative to i:

1. $i_\alpha \leq i$
   These are the cases C_1 and C_3 in Figure 10. There are $Ref(i_\alpha, i, 1)$ object in t_{i_α} that are connected with o_i. However, these clusters are only relevant if there exists

a path from $i+1$ to n because otherwise no update of the canonical access relation is needed. This probabilistic value is $P_{Ref}(i+1, n)$. Furthermore, for a given object o_{i_α} of type t_{i_α} an update is only needed if this object lies on some path emanating from t_0—which is accounted for by the probability $P_{RefBy}(0, i_\alpha)$.

2. $i < i_\alpha$
   This corresponds to case C_2 in Figure 10. It is handled analogously to case (1), except that now we have to consider the objects of type t_{i_α} that lie on a path emanating from the object o of type t_{i+1}—there are $RefBy(i+1, i_\alpha, 1)$ such objects. However, these clusters are only relevant for update if o_i is connected with t_0 and if the particular object of type t_{i_α} is connected with t_n.

The formula $qbw^i_X(i_\alpha, i_{\alpha+1})$ for the backward clustered B^+ tree is derived analogously.

6.2.2 Number of Clusters under Full Extension

$$qfw^i_{full}(i_\alpha, i_{\alpha+1}) = \begin{cases} Ref(i_\alpha, i, 1) + \sum_{l=i_\alpha+1}^{i} P_{lb}(l-1, l) * Ref(l, i, 1) & i_\alpha \le i < i_{\alpha+1} \\ 0 & else \end{cases}$$

$$qbw^i_{full}(i_\alpha, i_{\alpha+1}) = \begin{cases} RefBy(i+1, i_{\alpha+1}, 1) & \\ \qquad + \sum_{l=i+2}^{i_{\alpha+1}-1} P_{rb}(l, l+1) * RefBy(i+1, l, 1) & i_\alpha \le i < i_{\alpha+1} \\ 0 & else \end{cases}$$

For full extensions we have to consider only the *one* partition that covers $(i, i+1)$. This corresponds to case C_1 in Figure 10. All other partitions need not be updated and, therefore, their number of clusters is set to 0. Consider the forward clustered case: there are $Ref(i_\alpha, i, 1)$ objects of type t_{i_α} that have a path leading to o_i. All of these have to be updated. Furthermore, we have to insert information concerning objects that have a path leading to o_i but are not connected with any object in t_{i_α}, like the object represented by the small circle in Figure 10. The number of such objects is derived in the sum $\sum_{l=i_\alpha+1}^{i} P_{lb}(l-1, l) * Ref(l, i, 1)$.

The number of clusters for the backward clustered case is derived analogously.

6.2.3 Number of Clusters under Left-Complete Extension

For completeness we show the formulas for left- and right-complete extensions below. Their derivation is similar to the above explained cases.

$$qfw^i_{left}(i_\alpha, i_{\alpha+1}) = \begin{cases} 0 & i_{\alpha+1} \le i \\ Ref(i_\alpha, i, 1) * P_{RefBy}(0, i_\alpha) & i_\alpha \le i < i_{\alpha+1} \\ P_{lb}(0, i_\alpha) * RefBy(i+1, i_\alpha, 1) * P_{RefBy}(0, i) & i < i_\alpha \end{cases}$$

$$
qbw^i_{left}(i_\alpha, i_{\alpha+1}) = \begin{cases} 0 & i_{\alpha+1} \leq i \\ P_{RefBy}(0,i) * & \\ \quad (RefBy(i+1, i_{\alpha+1}, 1) + \sum_{l=i+2}^{i_{\alpha+1}-1} P_{rb}(l, l+1) * RefBy(i+1, l, 1)) & i_\alpha \leq i < i_{\alpha+1} \\ P_{RefBy}(0,i) * P_{lb}(0, i_\alpha) * & \\ \quad (RefBy(i+1, i_{\alpha+1}, 1) + \sum_{l=i_\alpha+1}^{i_{\alpha+1}-1} P_{rb}(l, l+1) * RefBy(i+1, l, 1)) & i < i_\alpha \end{cases}
$$

6.2.4 Number of Clusters under Right-Complete Extension

$$
qfw^i_{right}(i_\alpha, i_{\alpha+1}) = \begin{cases} P_{rb}(i_{\alpha+1}, n) * P_{Ref}(i+1, n) * & \\ \quad (Ref(i_\alpha, i, 1) + \sum_{l=i_\alpha+1}^{i_{\alpha+1}-1} P_{lb}(l-1, l) * Ref(l, i, 1) & i_{\alpha+1} \leq i \\ P_{Ref}(i+1, n) * & \\ \quad (Ref(i_\alpha, i, 1) + \sum_{l=i_\alpha+1}^{i} P_{lb}(l-1, l) * Ref(l, i, 1) & i_\alpha \leq i < i_{\alpha+1} \\ 0 & i < i_\alpha \end{cases}
$$

$$
qbw^i_{right}(i_\alpha, i_{\alpha+1}) = \begin{cases} P_{rb}(i_{\alpha+1}, n) * Ref(i_{\alpha+1}, i, 1) * P_{Ref}(i+1, n) & i_{\alpha+1} \leq i \\ RefBy(i+1, i_{\alpha+1}, 1) * P_{Ref}(i_{\alpha+1}, n) & i_\alpha \leq i < i_{\alpha+1} \\ 0 & i < i_\alpha \end{cases}
$$

6.3 Sample Results

6.3.1 Update Costs for Fixed Application Characteristics

We compare update costs for different access relation extensions and decompositions on the basis of the following application profile:

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	1000	5000	10000	50000	100000
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	900	4000	8000	20000	—
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 update costs for an update operation ins^3 are plotted in Figure 11. The access relations are, alternatively, in binary decomposition or non-decomposed.

diagupd

Figure 11: Update Costs for a Fixed Application Profile

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

6.3.2 Update Costs for Another Fixed Application Characteristics

Let us, for comparison, show a slightly different application profile:

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	1000	5000	10000	50000	100000
number of objects with **d**efined A_{i+1} attribute	d_0	d_1	d_2	d_3	d_4
	900	4000	8000	20000	—
fan-out	f_0	f_1	f_2	f_3	f_4
	2	1	1	4	
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$
	500	400	300	300	100

The update costs for an update operation ins^3 are plotted in Figure 12.

diagupd_

Figure 12: Update Costs for a Fixed Application Profile

Again, the update costs of the left-complete and full extension are almost comparable.

6.3.3 Update Costs under Varying Object Size

Consider the following application-specific parameters within which we will continuously increase the sizes of objects of all types within the interval $100 \ldots 800$.

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	1000	5000	10000	50000	100000
number of objects with	d_0	d_1	d_2	d_3	d_4
defined A_{i+1} attribute	900	4000	8000	20000	—
fan-out	f_0	f_1	f_2	f_3	f_4
	2	2	3	4	—
size of objects	$size_0$	$size_1$	$size_2$	$size_3$	$size_4$
	$100 \cdots 800$	$100 \cdots 800$	$100 \cdots 800$	$100 \cdots 800$	$100 \cdots 800$

The plot in Figure 13 visualizes the effect of varying object sizes on the update cost of ins^1. The access support relations are in binary decomposition.

diagupdvs

Figure 13: Update Costs for Varying Object Sizes

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.

6.4 Costs of Typical Operation Mix

6.4.1 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 $(1 \leq i \leq p)$ 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.

6.4.2 Update Mix under Binary Decomposition

The following application profile is used:

application characteristics					
n	4				
number of objects	c_0	c_1	c_2	c_3	c_4
	1000	5000	10000	50000	100000
number of objects with defined A_{i+1} attribute	d_0	d_1	d_2	d_3	d_4
	900	4000	8000	20000	—
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 query mix Q_{mix} consists of:

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

The update mix consists of:

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

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

Figure 14 shows the (normalized) costs for different update probabilities P_{up} ranging between $0.1 \ldots 0.9$.

It can be seen that for an update probability less than 0.3 the left-complete extension beats 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).

Figure 14: Operation Mix for Binary Decomposition

6.4.3 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 Figure 15

Figure 15: Operation Mix for the Decomposition $(0, 3, 4)$

6.4.4 Comparison: Left-Complete vs Full Extension

application characteristics						
n	5					
number of objects	c_0	c_1	c_2	c_3	c_4	c_5
	1000	1000	5000	10000	100000	100000
number of objects with	d_0	d_1	d_2	d_3	d_4	d_5
defined A_{i+1} attribute	100	1000	3000	8000	100000	—
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

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 16 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 $(0, 3, 4, 5)$.

graph1

Figure 16: Operation Mix for Full and Left-Complete Access Relations

6.4.5 Comparison: Right-Complete vs Full Extension

The following application profile is being used:

application characteristics						
n	5					
number of objects	c_0	c_1	c_2	c_3	c_4	c_5
	100000	100000	50000	10000	1000	1000
number of objects with **d**efined A_{i+1} attribute	d_0	d_1	d_2	d_3	d_4	d_5
	100000	10000	30000	10000	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 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 shown in the upper plot of Figure 17.

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

graph3

Figure 17: Isolating Right-Complete and Full Extension

It is not possible, to generally determine the best possible design choices: 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 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.

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 cost model is fully implemented as a Lisp program. 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.

Acknowledgements

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

References

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

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

[3] A. Kemper, P. C. Lockemann, and M. Wallrath. An object-oriented database system for engineering applications. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–311, May 1987.

[4] W. Kim, K. C. Kim, and A. Dale. Indexing techniques for object-oriented databases. Technical Report DB-86-006, MCC, 3500 West Balcones Center Drive, Austin, TX 78759, 1987.

[5] 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, Reading, MA, 1989. Addison Wesley.

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

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

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

[9] M. Stonebraker. The case for partial indexes. Memorandum UCB/ERL M89/17, Electronics Research Laboratory, Univ. of California, Berkeley, Berkeley, Ca 94720, Feb 1989.

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

[11] P. Valduriez. Join indices. *ACM Trans. on Database Systems*, 12(2):218–246, Jun 1987.

[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.