ALTERNATIVES IN COMPLEX OBJECT REPRESENTATION: A PERFORMANCE PERSPECTIVE

by

Anant Jhingran and Michael Stonebraker

Memorandum No. UCB/ERL M89/18

15 February 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ALTERNATIVES IN COMPLEX OBJECT REPRESENTATION: A PERFORMANCE PERSPECTIVE

by

Anant Jhingran and Michael Stonebraker

Memorandum No. UCB/ERL M89/18

15 February 1989

ALTERNATIVES IN COMPLEX OBJECT REPRESENTATION: A PERFORMANCE PERSPECTIVE

by

Anant Jhingran and Michael Stonebraker

Memorandum No. UCB/ERL M89/18

15 February 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Alternatives in Complex Object Representation: A Performance Perspective

Anant Jhingran and Michael Stonebraker

Computer Science Division
University of California, Berkeley

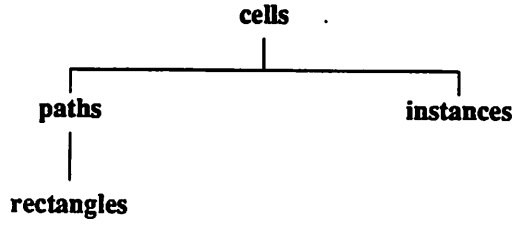
ABSTRACT

With database systems finding wider use in CAD, Office Information Systems (OIS) and logic programming applications, the importance of efficiently representing and manipulating complex objects is growing. In this study we look at a classification of the alternatives for representing complex objects. We then consider the performance aspects of one representation technique based on object identifiers. It is shown that clustering of subobjects with their referencing objects is rarely a good idea. In contrast, we show that caching the intermediate results of query processing can yield large benefits.

1. Introduction

Database management systems have proven to be cost-effective tools for organizing and maintaining large volumes of data. However, in recent years new database applications have occurred which do not store standard business-oriented data. In a significant number of these applications, there is a need to represent complex objects with few structural constraints. Some examples of these applications are CAD and engineering databases [BATO85, KIM87, LORI85], Office Information Systems (OIS), and logic programming [ZANI85].

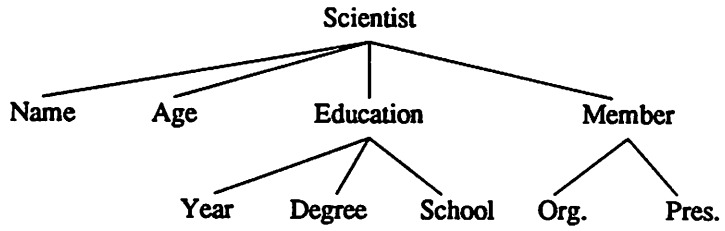
Loosely speaking, complex objects are "entities" that are either composed of, or related to, other (complex) objects. For example, consider the traditional complex object in VLSI [LORI85]:



VLSI **cells** are made of **paths** and **instances** of other cells. Paths are, in turn, made of **rectangles**. Here, a cell is a complex object because it is composed of other subobjects.

Complexity of objects is not due only to IS-PART-OF relationships. The domain of an attribute of an object can be other objects. This is supported by a mechanism similar to set-valued aggregation ([SMIT77]). For example, consider the following complex object:

This research was sponsored by the National Aeronautics and Space Administration under grant NAG 2-530, and by the Army Research Office under contract DAAL03-87-G-0041.



A scientist has a name, an age, (a set of) education qualifications, and membership in some organizations. His education is characterized by the indicated attributes. Similarly each organization has a name and a president.

To encompass both the above mentioned possibilities, we use the following definition:

A complex object is an object that has one or more (complex) objects as the value of one or more of its attributes.

Our definition of complex objects is similar to the one in [VALD86]. Moreover, an object can be a tuple in an extended relational environment such as [LORI85], or it can be an "object" in one of the more recent object-oriented models. The set of objects associated with an attribute of another object are termed **subobjects** of that object. In this study we look at the performance of the alternatives for representing the relationship between an object and its subobjects. The **semantics** of the term "object" are not a consideration in our study.

The rest of the paper is organized as follows. In Section 2 we consider several representation alternatives for complex objects. Section 3 then establishes the background for the representation analyzed in this study (object identifier based representation). It also discusses various strategies that can be used to process queries against a database utilizing such a representation. In Section 4 we discuss the simulation framework which is used for making performance comparisons between the alternatives presented in Section 3. The next two sections then present the results of our experiments. Finally, the paper ends with some concluding remarks in Section 7.

2. Approaches to Complex Object Representation

While there have been numerous proposals for representation of complex objects of various sorts ([BATO84, COPE85, BANE86, KIM87]), there has been little work on the performance aspects of the choices. Furthermore, these studies have generally concentrated on one or two representations, with few looking at the tradeoffs between many alternatives. For example, the emphasis by a group at MCC ([COPE85, VALD86]) is on a "decomposed storage model" of complex objects. The accompanying performance data is used to justify their choice. [LORI85] presents a scheme for representing complex objects in a relational environment. However, it fails to present any performance analysis of the scheme.

Our approach is more general. Instead of proposing and then defending a data model and implementation, we are studying as many alternative representations as possible. Hence we seek to identify the salient features of the representation possibilities and to provide a framework in which performance comparisons can be made. More specifically, for our study we have abstracted the main representation schemes, which we call the **primary** representation alternatives.

Retrieval of information pertaining to complex objects is generally a costly operation (typically involving more than one data set). This can be speeded up if some information is *precomputed* (i.e., before the retrieval in question) and cached. There are various levels at which this precomputation can be done,

and various ways that it can be stored. In our framework we capture this idea by allowing objects to have an auxiliary **cached** representation.

The primary and the cached representation form the axes of a **representation matrix**, which forms a unifying framework for our study.

We use a very simple example of complex objects to illustrate the basic options that are available. In this example and the remainder of this paper, we use record structures (relations) and tuples for notational convenience. Let

group (name, members, ...)

be a collection of complex objects. Some of the groups are:

elders: All persons with age ≥ 60
children: All persons with age ≤15
cyclists: All persons with cycling as their hobby

Moreover, there is information on the various subobjects, e.g.:

person (name, age, ...) /* Contains information on persons */
cyclist (name, ...) /* Contains information on cyclists */

2.1. Primary Representations

We have identified three primary ways of representing the relationship between an object and its subobjects. They are discussed below.

2.1.1. Procedural Representation

In a procedural representation, the set of subobjects associated with an object is identified by a **procedure**, which, when executed, evaluates to the corresponding subobjects. For our purposes, this procedure is a retrieve-only query on the underlying database. Considering our example above, the relation **group** might look like:

name	members	...
elders	retrieve (person.all) where person.age >= 60	...
children	retrieve (person.all) where person.age <= 15	...
cyclists	retrieve (person.all) where person.name = cyclist.name	...

This representation is supported in POSTGRES, a next generation database system being developed at Berkeley ([STON86, ROWE87]).

2.2. Object Identifier (OID) Representation

Here the set of subobjects of an object is represented by storing a list of their identifiers with the object. The relation **group** in the example database in OID representation might look like:

name	members	...
elders	7643 | 4367 | 565 | ...	...
children	454 | 7654 | 877 | ...	...
cyclists	4367 | 87 | ...	...

The numbers in group.members are the OID's of the corresponding members. These identifiers can be of many types (physical location based, system unique generated identifiers, primary keys of the objects, etc.). A detailed discussion of the relative merits of each approach can be found in [KHOS86]. In our study, we use the simplest OID's that provide location transparency — the concatenation of the relation identifier and the primary key of a tuple. In most of the recent object-oriented database systems ([BANE87, COPE84, HORN87]), complex objects are represented using OID representation, or some variation of it.

2.2.1. Value-Based Representation

In this representation scheme, subobjects are stored directly in the objects that reference them. Moreover, they have no associated identifiers, and hence cannot be referenced from elsewhere. Basically, the "value" (i.e., a concatenation of the values of each of its attributes) of a subobject is stored with the referencing object. Of course, when a subobject is shared by more than one object, we need to replicate its value wherever required.

Thus our example database might look like:

name	members			...
	name	age	...	
elders	John	62	...	...
	Mary	62	...	
	Paul	68	...	
	...	...	...	
children	Jill	8	...	...
	Bill	12	...	
	...	...	...	
cyclists	Mary	62	...	...
	Mike	44	...	
	...	...	...	

The original NF^2 model [SCHE86] supports this complex object representation. Moreover, EXTRA ([CARE88]) has an "own" data type which provides a similar representation.

2.3. Cached Representations

Generally speaking, the three primary representations capture different levels of knowledge about subobjects. In *procedural* representation, *how* to get the subobjects is known, but not their identities or contents. *Object Identifiers* capture the identities of the subobjects, but not their contents. Finally, in a

value-based representation, the object contains all the information about its subobjects.

It often makes sense to precompute and cache (on the disk) some information about subobjects that is closer to a value-based than the existing primary representation. Using this technique, one can avoid paying the sometimes large cost to determine the values of subobjects. If the primary representation is procedural, we can cache the OID's or the values of subobjects. If the primary representation already uses OID's, we can still cache the values of subobjects. Thus, the cached representation can be classified into *OID's* and *Value-Based*. Of course, we may choose not to cache anything.

If the primary representations is procedural, then caching the values of "elders" might look like:

name	members	...
elders	retrieve (person.all) where person.age >= 60	...
	John | 62 | ...	
	Mary | 62 | ...	
	Paul | 68 | ...	
	... | ... | ...	
children	retrieve (person.all) where person.age <= 15	...
cyclists	retrieve (person.all) where person.name = cyclist.name	...

Caching can be of two types. In **outside** caching, the relevant information of subobjects is cached away from the object that references them. These cached values can be shared with other objects that reference exactly the same set of subobjects. In **inside** caching, however, the information about the subobjects is cached with the referencing object. In this scheme, there can be no sharing of cached information. The example shown above utilizes inside caching.

2.4. Representation Matrix

In Figure 1, a **representation** matrix formed by the two axes, primary and cached representations, is shown. Some points in the matrix are shaded — they represent alternatives which do not make sense. For example, if the primary representation is *value-based*, then caching does not add to the performance — an object has all the required information about the subobjects.

It is our aim to study the properties of each unshaded box in Figure 1, and to determine the query processing strategies applicable to each. Some of the properties that need be studied are: storage requirements, cache maintenance costs, and query processing costs. These must be evaluated as a function of a variety of parameters (e.g., the frequency of updates).

In a previous study ([JHIN88]), we studied *procedural* representations in detail. The cached representations analyzed were *none* and *values*, and various query processing strategies for each were considered. It was shown that caching works, and that outside caching is, in general, better than inside caching. This is especially true when the size of the cache is limited and there is some sharing of subobjects.

In this study we will have a similar look at the possibilities in the column, OID representation. In a future study we will discuss the performance consequence of the other points in the matrix; as well as

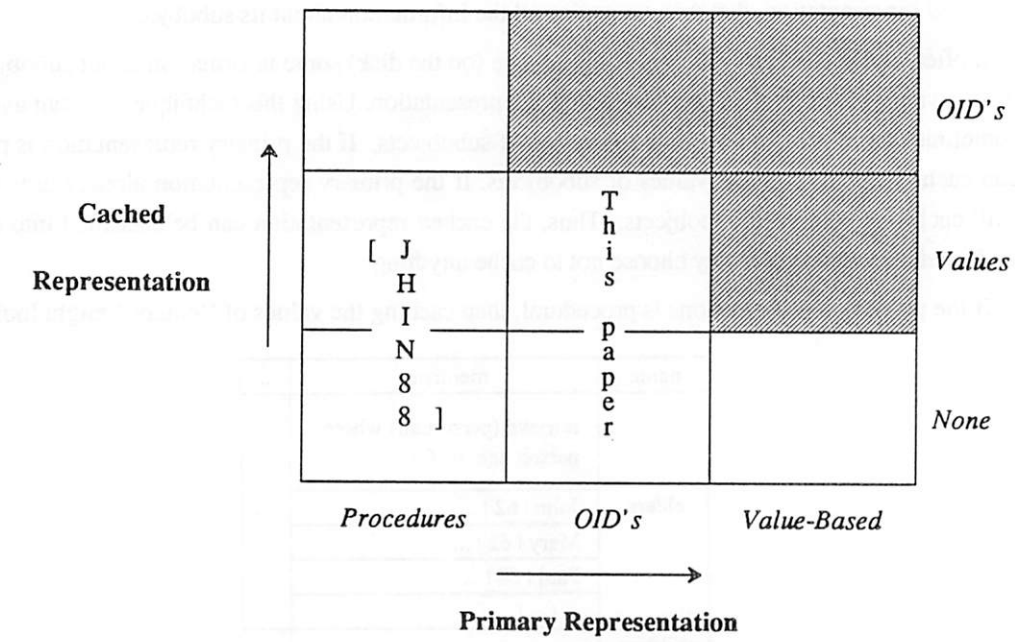
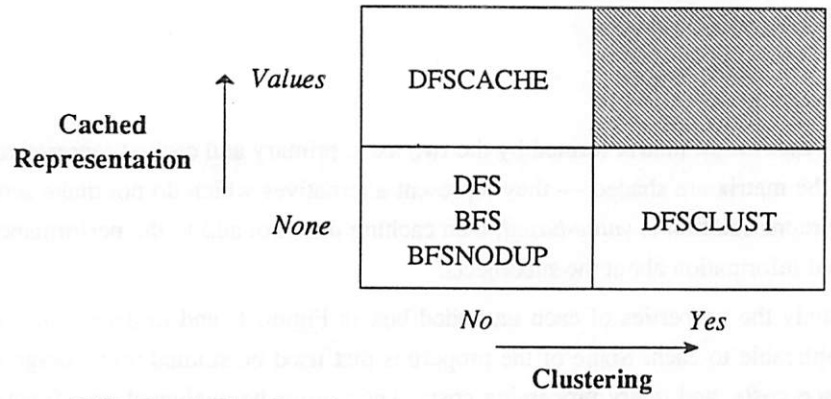


Figure 1: The Representation Matrix

compare points across the columns.

3. Strategies in OID Representation

When the primary representation is OID, there exists another axis of choice which is not shown in the matrix in Figure 1. The two possibilities on this axis are whether or not the subobjects are clustered with the referencing objects. This gives us four points to explore and compare. Figure 2 shows the four possibilities, along with the query processing (QP) strategies (to be explained later) for each option.



DFS: Depth-First Search
BFS: Breadth-First Search
BFSNODUP: BFS with Duplicate Removal
DFSCACHE: DFS in presence of Caching
DFSCLUST: DFS in presence of Clustering

Figure 2: Representations and QP Strategies Studied in this Paper

We do not explore the case when there is both clustering and caching, for reasons mentioned later.

We will use the following query to explain the query processing strategies available to us:

retrieve (group.members.name) where
group.name = "elders" or
group.name = "children"

which asks for the names of the members of the groups, elders and children, using a multiple-dot notation similar to [STON86, JHIN88]. This query has characteristics similar to transitive closure queries on arbitrary networks. Instead of exploring all subobjects that are transitively related to the selected objects, the above query requires only objects that are **directly** related to the selected tuples. Queries involving more than two dots in the target list require more levels of relationships to be explored.

We next discuss in detail the implications on query processing of each of the representation possibilities shown in Figure 2.

3.1. No Caching, No Clustering

There are two broad techniques for obtaining the transitive closure — depth-first exploration (generally termed as *recursion*), and breadth-first exploration (generally termed as *iteration*) [BANC86]. From the discussion above, each of these has a corresponding strategy for processing queries on complex objects.

[1] **DFS**: For each OID of "elders", fetch the corresponding subobject from the relation **person**, and return its name. Repeat the same for the OID's of "children".

[2] **BFS**: Collect the OID's from qualifying tuples of **group** into a temporary relation **temp** whose single attribute is OID. Next, execute the following query:

retrieve (person.name) where
person.OID = temp.OID

The optimal joining strategy in this query depends on the sizes of the relations involved. Iterative substitution is best when **temp** is small. In that case, subobjects are fetched **exactly** as in DFS, and consequently BFS is likely to perform slightly worse than DFS (due to the extra cost of forming the temporary relation). In contrast, merge-join is the optimal strategy when the size of the temporary is large. Clearly BFS will outperform DFS in this case. Consequently, whenever we talk of a competitive BFS strategy, we imply a merge-join.

[3] If subobjects are shared, then **temp** in BFS may have some duplicates. In that case it may be profitable to eliminate the duplicates before executing the above query, and we call the corresponding strategy **BFSNODUP**.

3.2. Value Caching, No Clustering

As discussed before, caching helps save some or all of the page accesses required for fetching the values of subobjects. In the presence of caching, a depth-first processing of the query requesting names of elders and children will proceed thus:

Check if the value of the subobjects of "elders" is cached. If so, fetch the attribute name from the cache. Otherwise, fetch the subobjects from the **person** relation (this is called *materialization*), cache their values, and return the attribute name.

Repeat the above for "children".

We define a **unit** of subobjects to be a collection of subobjects which belong to **one** relation and which are referenced by **one** object. In our example database, there are three units, one for each of the three complex objects shown. It is best to cache the values of the subobjects of a unit together in one place, since they will often be needed together.

It is easy to see that in the depth-first strategy outlined above (henceforth called DFSCACHE), units can be materialized and cached if they are not already cached. In a query processing strategy utilizing the cache, it is important that the cache is periodically "refreshed" (this aspect is called cache maintenance). If this is not done, then updates will invalidate the units in the cache, and it will dwindle, leading to poor performance.

In contrast to DFSCACHE, a breadth-first approach to query processing will prevent us from caching any freshly materialized unit. To see this, consider the following breadth-first strategy:

Check if the unit for "elders" is cached. If not, append the corresponding OID's to **temp**. Do the same for "children". In the worst case, **temp** will contain OID's for both children and elders.

Now execute the following query:

retrieve (person.all) where
person.OID = temp.OID

If this query is processed by a merge join, then the tuples will be returned sorted on the OID's. Since there is no relationship between ordering of OID's and ordering of units, the identity of the units would be lost. Consequently, in the basic BFS strategy, we cannot cache anything. Modifications to the basic strategy can permit cache maintenance, but the corresponding strategies lose the competitiveness of a breadth-first approach. Of course, if the above query is processed using iterative substitution, the performance will be slightly worse than DFSCACHE. Therefore, a breadth-first query processing strategy in the presence of caching is unviable, whatever be the joining strategy.

Updates to the subobjects invalidate cached values. We employ a very simple invalidation scheme. Associated with each subobject is a lock called an invalidation lock (I-lock, for short) for each unit that it belongs to. Consequently, when a subobject is updated, we invalidate all the (cached) units whose I-locks are held by the subobject in question. Details of this implementation can be found in [JHIN88, STON87].

In [JHIN88] we found that for a procedural representation, the parameters that determine the relative performance of inside and outside caching are the frequency of updates, the level of sharing, and the size of the cache. None of these is affected by the choice of the primary representation. Consequently, inside caching should also lose to outside caching over most of the parameter space when OID representation is used. Therefore we restrict our attention in this study to outside caching.

3.3. Clustering, No Caching

Clustering subobjects with their referencing objects is often proposed and its virtues have been propounded in several studies (e.g., [BANE86]). In our example, it means omitting **group** and **person**, and instead storing all objects and their subobjects in one relation called **cluster**. A depth-first processing of the query requesting the names of elders and children is straightforward (the general strategy is called DFSCLUST).

In a breadth-first approach, **cluster** is scanned for the qualifying objects (in this case, "elders" and "children"). The OID's from these qualifying records are then collected in **temp**, and the following query is executed:

```
retrieve (cluster.name) where
cluster.OID = temp.OID
```

For efficient merge join to occur for this query, **cluster** needs to be sorted on OID. However, if **cluster** stores each subobject with (one of) its parents, then it is impossible to ensure that the relation is sorted on OID. This can be seen from our example database. If we choose to cluster **person** tuples with **group** tuples, we can ensure an order among the tuples from **group**, and an order among the tuples of **person** that are clustered with the same tuple from **group**, but we cannot ensure a global OID order. Consequently a BFS strategy in the presence of clustering becomes unviable.

The effectiveness of clustering depends on the level of sharing of subobjects between the objects. Consider the following sets:

$$OS = \{(o,s) \mid s \text{ is a subobject of } o\}$$

$$OU = \{(o,u) \mid \text{object } o \text{ contains unit } u\}$$

$$US = \{(u,s) \mid \text{unit } u \text{ contains subobject } s\}$$

Then OS can also be expressed as:

$$OS = \{(o,s) \mid (o,u) \in OU \wedge (u,s) \in US\}$$

Two objects, o_1 and o_2, share the same subobject s if one of the following two conditions holds:

[1]

$$\exists\ u \text{ s.t. } (o_1,u) \in OU \wedge (o_2,u) \in OU \wedge (u,s) \in US$$

i.e. the two objects share the unit containing the subobject. The expected number of objects containing the same unit is called *UseFactor*.

[2]

$$\exists\ u_1,u_2 \text{ s.t. } (o_1,u_1) \in OU \wedge (o_2,u_2) \in OU \wedge (u_1,s) \in US \wedge (u_2,s) \in US$$

i.e. the units in the two objects share the subobject. The expected number of units that share the same subobject is called *OverlapFactor*.

From [1] and [2], the expected number of objects that share the same subobject (*ShareFactor*) is given by:

$$ShareFactor = UseFactor \times OverlapFactor$$

Let

$$C = \{(o,s) \mid \text{subobject } s \text{ is clustered with object } o\}$$

be the clustering assignment, with $C \subseteq S$. Depending on the values of *UseFactor* and *OverlapFactor*, clustering either succeeds or fails. Three cases arise:

[1] *ShareFactor*=1: (Both *UseFactor* and *OverlapFactor* are unity.) Each subobject belongs to exactly one parent object, and is best clustered with it. This is obviously the ideal condition for clustering, and here $C = S$.

[2] *OverlapFactor*=1: Subobjects in a unit are shared in their entirety by an expected *UseFactor* parent objects. Furthermore, no other object shares the subobjects in this unit. Thus if we cluster these subobjects with one of its parents (say o), the other parents can access the subobjects by fetching the physically clustered page(s) near o. Thus while the ideal situation is achieved for o, the situation for other parents is not too bad, since their subobjects are still physically clustered, albeit elsewhere, and can be fetched in one random access. The choice of o depends on the access patterns. In the absence of any knowledge, o should randomly chosen from *UseFactor* possibilities, and this is the approach used in our study.

[3] *OverlapFactor*>1: With an increasing *OverlapFactor*, it is impossible to ensure that the subobjects of an object o are either physically clustered with it, or physically clustered together somewhere else. To see this, consider the following assignments of subobjects to units:

$$U_{-1} = \{s_{-2}, s_{-1}, s_0, s_1, s_2\}$$
$$U_0 = \{s_0, s_1, s_2, s_3, s_4\}$$
$$U_1 = \{s_2, s_3, s_4, s_5, s_6\}$$

where U_i refers to i^{th} unit and s_j refers to the j^{th} subobject. Further assume that *UseFactor* = 1 and hence U_i occurs in the object o_i. Let us assume that in clustering subobjects with their parents, we assign subobjects of U_{-1} and U_1 to their respective parents before treating U_0. Consequently the subobjects of o_0 (which has the unit U_0) are now present in two places — some with o_{-1} and the rest with o_1. Thus to fetch the subobjects of o_0, we have to do at least two random accesses. The larger the *OverlapFactor*, the worse this situation is likely to be. In an extreme case, no two subobjects of an object are clustered together, and hence a random access is required for each subobject.

3.4. Caching, Clustering

Both clustering and caching attempt to improve performance by reducing the number of page accesses required to fetch the values of the subobjects. However, the approaches taken in the two cases are different. Thus it does not make sense to combine the two, and we ignore this representation option.

4. Experimental Setup

The approach used by us to do a performance comparison between various strategies was the following:

[1] Store a close approximation to the database structures in a relational DBMS.

[2] Map the queries and the processing strategies into corresponding queries and execution plans on the DBMS.

[3] Run a sequence of queries (containing a mix of retrieves and updates, satisfying some parameters) on the database and note the average I/O traffic. This average I/O cost was the performance yardstick.

We used commercial INGRES to store our databases. A main memory buffer size of 100 INGRES data pages was used throughout our study. The relations involved in our experiments were the following:

ParentRel (OID,ret1,ret2,ret3,dummy,children) /* This contains the objects */
ChildRel (OID,ret1,ret2,ret3,dummy) /* This contains the subobjects */
ClusterRel (cluster#,OID,ret1,ret2,ret3,dummy,children)
/* In presence of clustering, this contains both the objects and the subobjects */
Cache (hashkey,value) /* In presence of caching, this contains the cached values */

Ret1, ret2 and ret3 are integer fields and occur in the target lists of the retrieve queries. Dummy is a character field which serves to "pad" a tuple to a desired width. Each tuple of **ParentRel** is a complex object and the OID's of its subobjects are kept in its "children" attribute (which is a character field). Both **ParentRel** and **ChildRel** are structured as B-trees [RTI86] on OID. This facilitates the merge-join in BFS.

ClusterRel has the union of the attributes of **ChildRel** and **ParentRel**. In order to ensure that the subobjects assigned to an object are physically clustered with it, an additional attribute, cluster#, is needed. **ClusterRel** is structured as a B-tree on cluster#. An object and the subobjects clustered with it have the same cluster#, and hence are physically clustered in **ClusterRel**. In order to randomly access an object with a given OID, we need an index on ClusterRel.OID. In our environment, there are no insertions or deletions, and hence the index is static. Consequently, it is maintained as an **isam** structure.

Cache is the relation used to store the cached units. Associated with each unit is a hashkey which is a function of the concatenation of the OID's in that unit. **Cache** is maintained as a **hash** relation, hashed on hashkey.

The character fields in the above relations (dummy, children and value) are implemented as fixed length attributes, but with the blanks "compressed" [RTI86]. This permits implementation of variable length records. A typical length of a **ParentRel** tuple is 200 bytes, and of a **ChildRel** tuple, a hundred bytes. In all our experiments, the cardinality of **ParentRel** was fixed at 10,000 tuples. The results for larger database sizes can be obtained from scaling the results at this cardinality, provided a proportionally larger cache and main memory buffer is used. The size of the other relations depended on the parameters, but a typical database occupied around 10 MBytes.

Let *SizeUnit* be the expected number of subobjects in a unit. *SizeUnit* was fixed at 5 in this study. The cardinality of **ChildRel** is determined by the following equation:

$$|\textbf{ChildRel}| = \frac{|\textbf{ParentRel}| \times SizeUnit}{ShareFactor} = \frac{50000}{ShareFactor} \qquad (1)$$

The tuples of **ParentRel** and **ChildRel** were assigned unique OID's and random values for ret1, ret2, ret3 and dummy. Let *NumUnits* be the number of **distinct** units present in ParentRel.children. Then,

$$NumUnits = \frac{|\textbf{ParentRel}|}{UseFactor} = \frac{10000}{UseFactor}$$

From |**ChildRel**| subobjects, *NumUnits* units were randomly generated. These units were then randomly assigned to the objects in **ParentRel**. The attribute ParentRel.children stores the OID's of the subobjects belonging to the unit assigned to the corresponding object.

The other parameters relevant to this study are:

[1] *Pr*(*UPDATE*): This is the frequency with which updates occur in a query sequence. Each update modifies a fixed number of tuples of **ChildRel** in place.

The rest of the queries in a query sequence are of the form:

retrieve (ParentRel.children.attr) where
val1 ≤ ParentRel.OID < val2

with attr being randomly chosen (for each query separately) from ret1, ret2, ret3. In the presence of clustering, the updates and the retrieve queries are translated into equivalent queries on **ClusterRel**. *Pr*(*UPDATE*) was varied between 0 and 1.

[2] *NumTop*: This is the number of **ParentRel** tuples that are selected by a retrieve query in a sequence and depends on (val2 - val1). It was varied between 1 and 10,000. Different queries in the same sequence select different objects (by randomly chosing val1). Consequently, each complex object has an equal likelihood of being accessed. Hence the clustering assignment, C, is randomly chosen from the set of possibilities.

[3] *SizeCache*: This is the maximum number of units that can be cached, i.e., the upper bound on |**Cache**|. Since the cache takes up disk space, it is reasonable to place a bound on size of the cache, and hence the parameter. *SizeCache* was fixed at 1000 units for this study, making the cache about 10% of a typical database size in our environment.

[4] *OverlapFactor*: The expected number of units sharing a subobject was set at one for the first set of experiments which are discussed in Section 5. This parameter setting clearly favors clustering. In Section 6 we describe an experiment where *OverlapFactor* was varied.

[5] *UseFactor*: The expected number of objects sharing a unit was varied between 1 and 50, with the default being 5.

The performance of the query processing strategies listed in Figure 2 was studied under varying conditions using a driver program written in EQUEL/C [RTI86]. The driver first generated a sequence of random queries satisfying some parameters. Depending on the query processing strategy being studied, an optimal plan for each query in the sequence was then generated. The plan was then run on the database, and the average I/O performance noted. The I/O activity for a sequence of queries was measured using some system constants in commercial INGRES that can be queried by a user program. The number of retrieve queries in a sequence was typically 1000.

5. Performance Results

This section discusses the performance results that were obtained from the experiments executed. Recall that there are five query processing strategies under consideration in this study. In Section 5.1 we discuss the three strategies in the absence of caching and clustering. We show that only one (BFS) has wide applicability and thus discard DFS and BFSNODUP. That leaves us with three strategies (BFS, DFSCACHE and DFSCLUST), which are compared in Section 5.2.

5.1. DFS, BFS, or BFSNODUP?

Figure 3 plots the cost of these three algorithms with varying *NumTop*.. It is obvious that DFS is a loser when *NumTop* exceeds 50 or so. This is because DFS implements a nested-loop join between **ParentRel** and **ChildRel**, whereas BFS can implement a merge join. At low *NumTop*, the cost of forming a temporary in BFS is significant, and hence it performs slightly worse than DFS. In [GUTT84], a similar analysis is done for VLSI CAD databases and the conclusion reached is that for *NumTop* = 1, DFS is better than BFS.

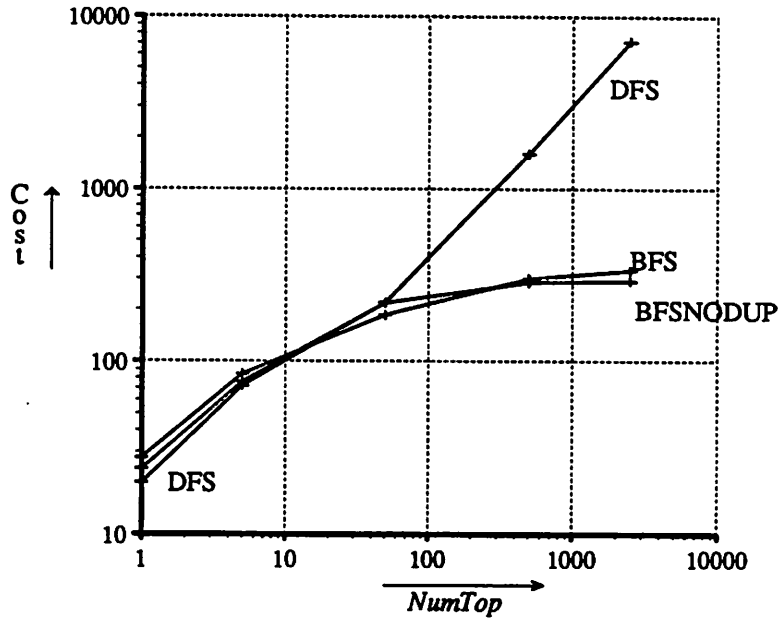


Figure 3: Performance comparison without clustering or caching

(*ShareFactor*=5)

The other notable feature of this graph is that BFSNODUP is not much better than simple BFS in our environment. Thus it is not worthwhile to try and eliminate duplicates, even though *ShareFactor* = 5. It is clear that the benefits of BFSNODUP will increase with an increase in the number of levels explored. But our experiments have shown that the benefit so obtained is marginal at best. Consequently, BFSNODUP is not a strategy worth pursuing. However, [GUTT84, BANC86] discuss cases in other environments where removing duplicates pays off.

In conclusion, BFS is the strategy of choice when there is no caching or clustering, and this is especially true at higher *NumTop* values. A database system supporting complex objects would be expected to implement two or three query processing strategies that have relatively wide applicability. Since DFS and BFSNODUP do not fall into this category, they are not considered from now on.

5.2. BFS, DFSCACHE, or DFSCLUST?

In this subsection we discuss the relative performance of the three remaining query processing strategies — BFS, DFSCLUST and DFSCACHE. Three parameters critically affect these strategies. *NumTop* determines the relative performance of breadth-first and depth-first approaches. *ShareFactor* is central to the performance of clustering. The frequency of updates, *Pr*(*UPDATE*), is a significant determiner of the performance of caching.

Figure 4 plots a three dimensional graph showing the regions where each algorithm performs the best, as a function of the three parameters mentioned above. The graph was obtained by noting the I/O cost of each strategy over approximately 300 points in the enclosing 3-D space, determining the best strategy at each point, and extrapolating these points into regions. DFSCLUST outperforms the other two under the region covered by the surface ABFGHCA. To the right of the surface marked by BCED, and above the DFSCLUST region is the region where BFS is the best. In the rest of the region, DFSCACHE is the optimal strategy.

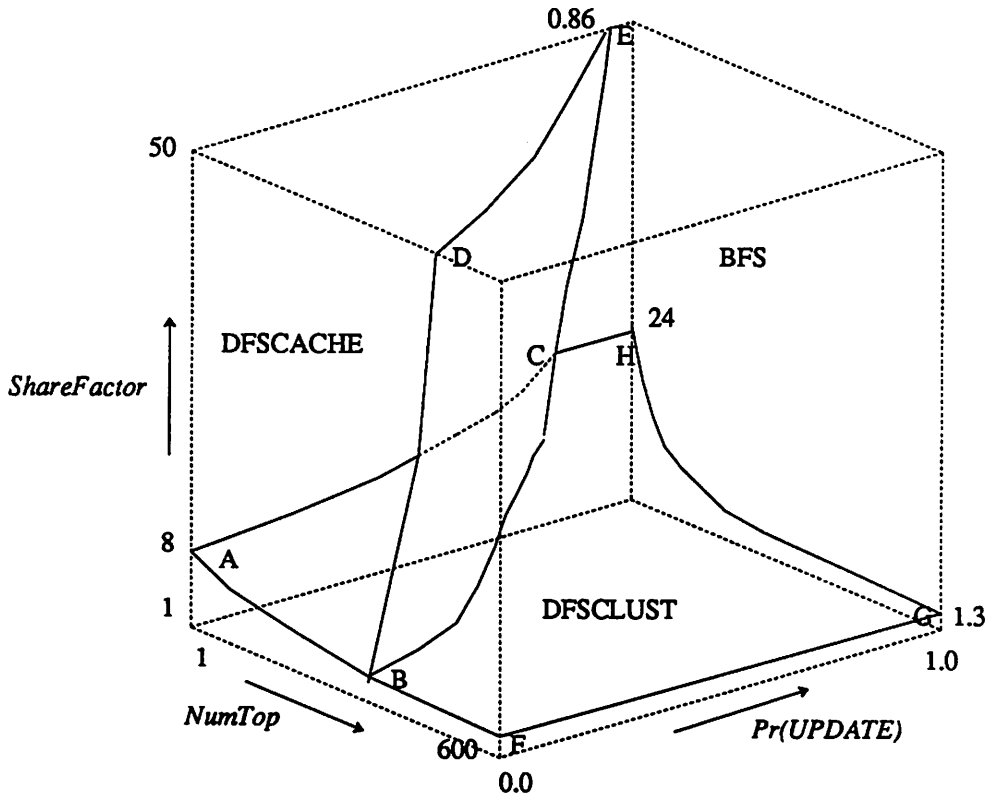


Figure 4: Regions where each strategy performs the best as functions of *ShareFactor*, *NumTop* and *Pr(UPDATE)*

We next explain the projections of this graph along the various faces of the enclosing cuboid. This will help in understanding the regions of space.

5.2.1. *Pr(UPDATE)* $\rightarrow$ 1

This is the right back face in Figure 4. When updates occur with such high frequency, any caching strategy is unviable. The reason for this is two-fold. First, updates invalidate cached units and hence the cost of invalidation has to be paid. The second effect of increased frequency of updates is a decreased number of units that are cached. Consequently, the retrieve-only queries find fewer cached units and have to materialize more units. Thus the cost of the retrieve queries goes up.

With the above two factors, caching is clearly a loser when *Pr(UPDATE)* $\rightarrow$ 1. The only question then remains is whether or not to cluster? We turn to two graphs to answer this question. The cost of a query is separated into two components — the cost of accessing the tuples of **ParentRel** (*ParCost*), and the cost of fetching the subobjects (*ChildCost*). Consider the case where *NumTop*=200. Figure 5(a) plots the *ParCost*,*ChildCost*, and *TotCost*=*ChildCost*+*ParCost* as a function of *ShareFactor* for DFSCLUST, and Figure 5(b) does the same for BFS.

In Figure 5(a), it is clear the *ParCost* increases with decreasing *ShareFactor*. This is because as *ShareFactor* decreases, we approach ideal clustering. As a result, more and more tuples of **ParentRel** have their subobjects clustered with them; consequently, the cost for accessing several contiguous tuples of **ParentRel** (which the retrieve queries in the sequence ask for) increases. Thus, one disadvantage of clustering is **the increase in cost of accessing contiguous objects.**

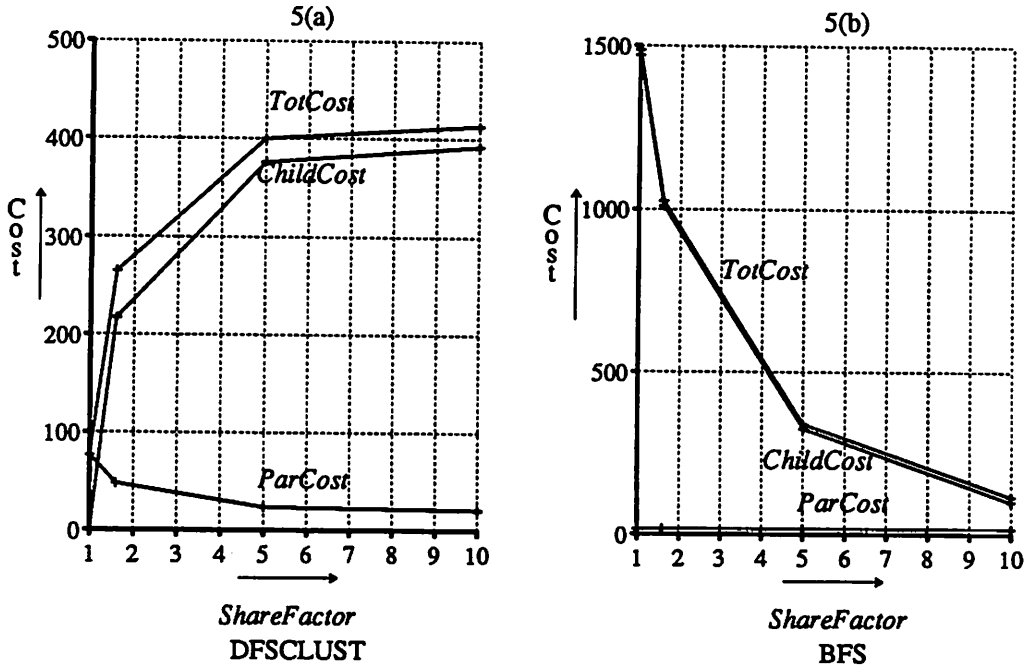


Figure 5: The cost breakups as a function of *ShareFactor*

In a trend opposite to *ParCost*, *ChildCost* decreases as *ShareFactor* decreases. This is to be expected since with a decrease in *ShareFactor*, more and more tuples of **ParentRel** have their subobjects clustered with them, and thus fetching the corresponding subobjects often involves no additional I/O's. The trend in total cost is dominated by *ChildCost*.

In Figure 5(b), we note that *ParCost* is relatively unaffected by *ShareFactor* for BFS. The interesting phenomenon is the **decrease** in *ChildCost* with an increase in *ShareFactor*. This is because an increase in *ShareFactor* causes a decrease in |**ChildRel**| (from eqn. (1)). Consequently, the cost of the merge join in BFS decreases. Here too, the trend of total cost is dominated by *ChildCost*.

If we plot the total costs for BFS and DFSCLUST on the same graph, the trend in DFSCLUST is an increase in cost with an increase in *ShareFactor*, and the reverse is true for BFS. Consequently, there exists a *ShareFactor* (here 4.7), beyond which BFS is better than DFSCLUST.

It can be shown that as we increase *NumTop*, the boundary between BFS and DFSCLUST shifts to lower values of *ShareFactor*. This is because of the inherent disadvantage of DFS strategies at higher *NumTop*.

From the above discussion, the back face of the cuboid in Figure 4 is self-explanatory. For large *NumTop*, clustering is the way to go **only if** *ShareFactor* is very close to 1. Of course, if *ShareFactor* is exactly one, then clustering will beat any strategy, regardless of the value of *NumTop*.

5.2.2. *Pr(UPDATE)* $\rightarrow 0$

Clearly, caching is viable here. From the discussion above, clustering continues to be viable at low *ShareFactor*. However, here the upper boundary of the DFSCLUST region is *ShareFactor*=8 as opposed to *ShareFactor*=24 on the back face. Thus caching cuts into the competitiveness of clustering at low *NumTop*.

ShareFactor affects caching directly. In outside caching, a cached unit is shared by all objects containing that unit. Consequently, for a constant *SizeCache*, an increase in *ShareFactor* causes an increase in the number of objects which have their units cached. This results in an improved performance of DFSCACHE.

At large values of *NumTop*, DFSCACHE loses to BFS because of the inherent disadvantage of a depth-first strategy.

5.2.3. Very High *ShareFactor*

Consider the top face of the cuboid. Clustering is useless here, and the boundary shows that caching wins if *NumTop* is low and/or *Pr(UPDATE)* is low. The reasons for this are obvious.

5.2.4. *NumTop* $\rightarrow$ 1

This is the back left face of the cuboid. It is clear that the boundary between DFSCLUST and BFS is independent of *Pr(UPDATE)*. In the absence of caching, this face would have been divided into two regions separated by the line *ShareFactor*=24. Above this line, BFS would be better, and below it, clustering would win. However, in the presence of caching, the regions for BFS and DFSCLUST shrink.

5.3. Summarizing the 3-D plot

In summary, clustering is viable only when the sharing level is relatively low. Even when clustering is better than BFS, it is often not better than DFSCACHE, especially if *Pr(UPDATE)* is low. Two reasons have been identified for the poor showing of clustering in some circumstances — the inapplicability of a breadth-first search, and the complications due to sharing of subobjects.

We have also shown that caching wins in many cases. Hence our results are similar to what were obtained in our study on procedural representation [JHIN88]. However, the query processing strategy associated with caching suffers from a drawback. If we try to maintain the cache (i.e. cache freshly materialized units), we have to abandon a breadth-first approach. As our results show, this leads to a poor performance when *NumTop* is large.

To alleviate this problem, the following SMART strategy which makes the best use of caching is suggested:

> When the query has a low *NumTop*, use DFSCACHE, and maintain the cache. However, if *NumTop* $> N$ (where $N \approx 300$ in our experiments), use a breadth-first strategy, and do not try to maintain cache. In other words, scan the *NumTop* tuples and collect into **temp** the OID's whose units are not cached; and then implement the merge-join. The status of the cache remains invariant during the execution of the breadth-first strategy.

If the queries against the database have a good mix (some low *NumTop* queries, and some large *NumTop* queries), then the above solution will make caching outperform BFS for most values of *NumTop*, provided *Pr(UPDATE)* is not too high. This is because when SMART uses a breadth-first approach, the size of its temporary relation is no larger than the temporary used in BFS (since some units may be cached, and hence their OID's need not be included in the temporary relation). Of course, for low *NumTop*, SMART is identical to DFSCACHE, and hence better than BFS. However, if the database sees very few low *NumTop* queries, SMART will not be able to maintain cache and hence will degrade in performance.

6. Other Cases

We now discuss some performance results when the parameters were different from the experiments described in the previous section. We first study the effect of increasing *OverlapFactor*. Next, the consequences of subobjects belonging to more than one **ChildRel** are discussed.

6.1. *OverlapFactor* > 1

Recall that as *OverlapFactor* is increased, it becomes more and more difficult in clustering to keep the subobjects of a unit clustered together in one place. Consequently, DFSCLUST deteriorates. Our next graph (Figure 7) shows exactly that. *Pr(UPDATE)* is assumed to be 1, so that DFSCACHE does not enter the picture.

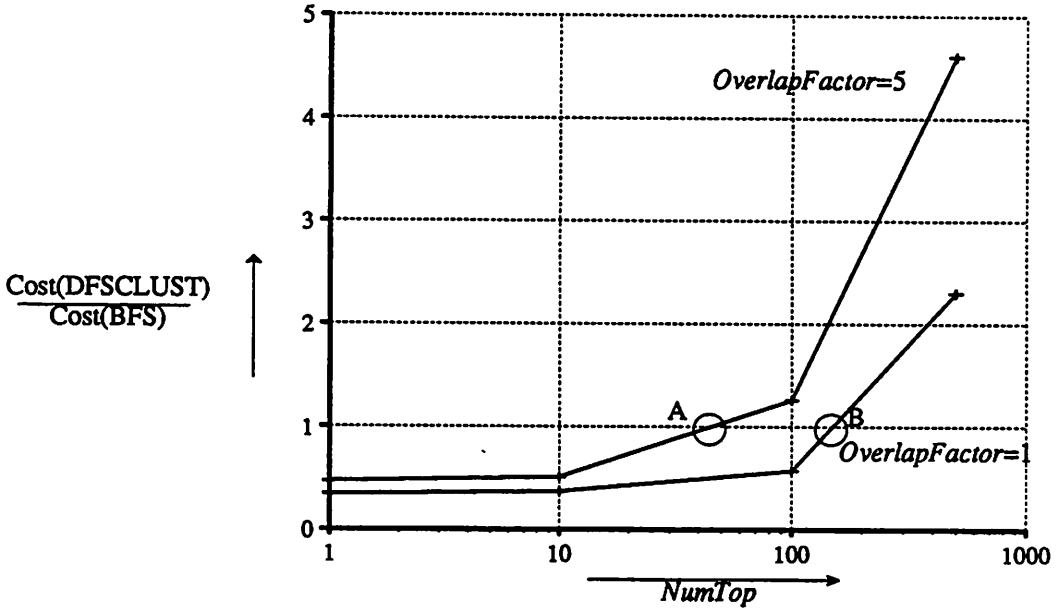


Figure 7: Effect of *OverlapFactor* on clustering (*ShareFactor* is fixed at 5 for both the curves)

It plots $\frac{Cost(DFSCLUST)}{Cost(BFS)}$ vs. *NumTop* for two cases: 1) *OverlapFactor*=1, *UseFactor*=5 and 2) *OverlapFactor*=5, *UseFactor*=1. In both the cases, each subobject is shared by an expected number of 5 tuples of **ParentRel**, but in different ways.

There is little difference in the performance of BFS in the two cases. Consequently the ratio reflects the effects on DFSCLUST. The curve for *OverlapFactor*=5 is considerably above that of *OverlapFactor*=1. This demonstrates the degradation in the performance of clustering strategies with increasing *OverlapFactor*.

Furthermore, the value of *NumTop* beyond which BFS beats DFSCLUST moves from B to A as *OverlapFactor* is increased from 1 to 5. Thus in Figure 4, it can be demonstrated that the projection curve on the plane *Pr(UPDATE)*=1 moves lower with an increasing *OverlapFactor*. Similarly, the volume occupied by the clustering region decreases.

In summary, there exist object-subobject assignments where clustering performs even worse than what was demonstrated before.

6.2. Subobjects from different relations

Increasing the number of relations from which the subobjects of **ParentRel** are drawn (called *NumChildRel*) has little effect on DFS strategies. Consequently, it has little effect on either clustering or caching strategies. However, it affects BFS significantly.

As BFS scans through qualifying **ParentRel** tuples, it encounters OID's from say, $n \leq NumChildRel$ relations. It then needs to execute n queries of the form:

```
retrieve (ChildRel[i].attr) where
ChildRel[i].OID = temp[i].OID
```

where $1 \leq i \leq n$ and **temp[i]** keeps the OID's from **ChildRel[i]** relation. Thus one might expect that BFS deteriorates as *NumChildRel* increases. This is indeed the case, but surprisingly from the experiments conducted, it turns out that the deterioration is far slower than expected.

As *NumChildRel* increases, so does n. But simultaneously, there is a decrease in the cardinalities of each **ChildRel** (this can be shown to be the case if other parameters are kept constant). Furthermore, there is a decrease in the size of each temporary. Consequently, the cost of each of the n queries goes down — almost balancing out any effects of increasing number of queries that need to be executed. This continues till *NumChildRel* approaches *NumTop*. At that point, each temporary contains only one or two OID's, and BFS consequently deteriorates.

Summarizing, none of our algorithms is significantly affected by *NumChildRel*, at least if it is much less than *NumTop*.

7. Conclusions

In this study we first identified three broad approaches to representation of complex objects in database systems — procedural, OID's, and value-based. We also identified the caching possibilities, and then presented a framework in which performance comparisons between various representations can be made. The rest of the paper discussed the alternatives in OID representation, and the relative merits of each of them.

Since the comparisons are performance based, we established a simulation framework in which experiments were run to determine the cost of each strategy, given the parameters. We also identified a few parameters which were central to the strategies under consideration. From the experiments, we reached the following conclusions.

In the absence of caching and clustering, a breadth-first approach to query processing is the best. Furthermore, it is not worth the effort to try and remove duplicates in an intermediate step in this strategy for query processing.

Clustering works, but only in a few circumstances. In most of the parameter space, it is better to leave the objects and the subobjects in their respective physical locations. Outside caching wins when updates occur with low to moderate frequency, and queries request subobjects for only a few objects. It is also desirable to occasionally do away with the "cache maintenance" aspect of retrieve-only queries in the presence of outside caching. In the presence of a good query mix, this helps maintain the competitiveness of caching to even higher number of object requests.

There exist object-subobject assignment distributions where clustering performs even worse than what the first set of experiments suggested. This happens when units of subobjects are overlapping.

Therefore, clustering is appropriate only if subobjects are not shared, or, at worst, they are shared in units — any random sharing is likely to lead to poor performance for clustering.

Finally, the fact that subobjects are drawn from a varied number of relations has little effect on the performance of any algorithm.

References

[BANC86] Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies," Proc. ACM-SIGMOD Conf., 1984.

[BANE86] Banerjee, J. and Kim, W., "Clustering a DAG for CAD Databases," MCC Technical Report Number: DB-128-85, Microelectronics and Computer Technology Corporation, Feb. 1986.

[BANE87] Banerjee, J., et al., "Data Model Issues for Object-Oriented Application," ACM Trans. on Office Info. Sys. 5(1), Jan. 1987.

[BATO84] Batory, D.S., and Buchmann, A.J., "Molecular Objects, Abstract Data Types, and Data Models: A Framework," Proc. VLDB, 1984.

[BATO85] Batory, D.S., and Kim, W., "Modeling Concepts for VLSI CAD Objects," ACM Trans. on Database Systems, 10(3), Sept. 1985.

[CARE88] Carey, M. et al., "A Data Model and Query Language for EXODUS," Proc. ACM-SIGMOD Conf., 1988.

[COPE84] Copeland, G. and Maier, D., "Making Smalltalk a Database System," Proc. ACM-SIGMOD Conf., 1984.

[COPE85] Copeland, G.P. and Khoshafian, S.N., "A Decomposition Storage Model," Proc. ACM-SIGMOD, 1985.

[GUTT84] Guttman, A., "New Features for a Relational Database System to Support Computer Aided Design," Memo. UCB/ERL M84/52, University of California, Berkeley, 1984.

[HORN87] Hornick, M.F. and Zdonik, S.B., "A Shared, Segmented Memory System for an Object-Oriented Database," ACM Trans. on Office Information Systems, 5(1), Jan. 1987.

[JHIN88] Jhingran, A., "A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures," Proc. VLDB, 1988.

[KHOS86] Khoshafian, S.N. and Copeland, G.P., "Object Identity," Proc. of OOPSLA, 1986.

[KHOS87] Khoshafian, S.N. et al., "A Query Processing Strategy for The Decomposed Storage Model," Proc. Conf. on Data Engr., 1987.

[KIM87] Kim, W. et al., "Operations and Implementation of Complex Objects," Proc. Conf. on Data Engr., 1987.

[LORI85] Lorie, R. et al., "Supporting Complex Objects in a Relational System for Engineering Databases," in Query Processing in Database Systems, eds. Kim, W., Reiner, D. and Batory, D., Springer-Verlag, 1985.

[ROWE87] Rowe, L. and Stonebraker, M., "The POSTGRES Data Model," Proc. VLDB, 1987.

[RTI86] Relational Technology Inc. INGRES Release 5.0 Reference Manuals, 1986.

[SCHE86] Schek, H.-J. and Scholl, M., "The Relational Model with Relation-Valued Attributes," Information Systems, 11(2), 1986.

[SMIT77] Smith, J.M. and Smith, D.C.P, "Database Abstractions: Aggregation and Generalization," ACM Trans. on Database Sys., 2(2), June 1977.

[STON86] Stonebraker, M. and Rowe, L., "Design of POSTGRES," Proc. ACM-SIGMOD, 1986.

[STON87] Stonebraker, M. et al., "Extending a Database System with Procedures," ACM Trans. on Database Sys., 12(3), Sept. 1987.

[VALD86] Valduriez, P. et al., "Implementation Techniques for Complex Objects," Proc. VLDB 1986.

[ZANI85] Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects," Proc. VLDB, 1985.