Implementation Techniques of Complex Objects

Patrick Valduriez, Setrag Khoshafian, George Copeland

MCC, Austin Texas 78759

Abstract: *Efficient support for retrieval and update of complex objects is a unifying requirement of many areas of computing such as business, artificial intelligence, office automation, and computer aided design. In this paper, we investigate and analyze a range of alternative techniques for the storage of complex objects. These alternatives vary between the direct storage representation of complex objects and the fully decomposed storage representation of complex objects. Qualitative arguments for each of the strategies are discussed. Analytical results and initial implementation results based on fully decomposed schemes are presented.*

1. Introduction

Many areas of computing such as business (conventional data processing applications), artificial intelligence, office automation, and computer aided design exhibit the common requirement of efficiently supporting complex objects. An attribute of a complex object need not be simple but may be an object itself. Complex hierarchical terms as present in logic [ZANI85], CAD design objects [BATO85] or objects used in office automation systems [ADIB84] are examples of complex objects. Although relational technology brings many nice features (e.g., set oriented operations), it relies on additional tools to provide the complex objects the user needs (e.g., report generator). This is one reason among others that the database management systems most used today remain hierarchical. Several complex object models [HASK82, LUM85, OZSO85] have been proposed to combine the respective advantages of the relational and hierarchical models. In this paper, we assume a particular conceptual complex object model [BANC86], and we investigate and analyze several strategies for the storage and access of complex objects for this model. All of our examples will be based on a business application.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

One of our primary concerns for chosing among the alternative storage schemes is *the IO cost*. We believe magnetic disks will remain the main type of home repository for medium and large sized databases. We also believe RAM speeds are going to increase at a higher rate than disk access times. Therefore, in comparing between particular storage models, the IO overhead is considered our main criteria.

The main motivation for the efficient manipulation of complex objects is high performance execution of database operations which retrieve and manipulate complex objects. The problem in achieving this goal is that there are multiple access patterns to the data. For example, if the complex object stores orders within their customer, one type of query can retrieve all data pertinent to a particular customer, whereas another type of query can retrieve data pertinent to orders independant of customers. Since the objects can be clustered in only a single way (without replication), favoring some access patterns is generally done at the expenses of others. Also, supporting multiple access patterns leads to additional complexity of storage structures and algorithms.

After having introduced our complex object model, we will investigate two alternative implementation techniques for it. The first one, called *direct storage model,* maps the objects directly into a physical address space so that sub-objects and objects are clustered together. The second model, called *normalized storage model,* has several variants. The idea here is to decompose and store the atomic objects of same type in flat files, and to capture the connections between objects and sub-objects (i.e., the *belongs-to* relationship) in either flat (binary) or hierarchical structures called *join indices.* We give a qualitative analysis of the trade-offs of these alternative storage models on various dimensions such as complexity, efficiency and generality. Finally, we give analytical and observed performance measures of an on-going implementation effort of two variations of the normalized storage model.

The remainder of this paper is organized as follows. In section 2, we define precisely our complex object model. Section 3 discusses the direct storage model while section 4 investigates the normalized storage model. Section 5 gives the performance evaluation and measurements. Section 6 is the conclusion.

2. Complex Object Model

2.1. Definition of Complex Objects

We now expand the notion of a complex object conceptual model. A formal definition of the model and the calculus for complex objects is given in [BANC86]. A

functional language for this complex object model is given in [BANC85]. *Objects* are defined recursively as follows :

(1) Integers, floats, booleans and strings are objects that we call *atomic objects*.

(2) If O_1 , O_2, ..., O_n are objects and a_1, a_2, ..., a_n are distinct attribute names, then

$[a_1 : O_1,\ a_2 : O_2,\ \ldots,\ a_n : O_n]$

is an object that we call a *tuple object*.

(3) If O_1, O_2, ..., O_n are objects, then

$\{O_1,\ O_2,\ \ldots,\ O_n\}$

is an object that we call a *set object*.

Tuples can have atomic, tuple or set valued attributes. The first option puts us in a normalized relational context and hence provides direct support for normalized relations in the storage model. The second option provides us with the possibility of supporting hierarchical terms as in [ZANI85].

Finally, set valued attributes allow us to have nested relations as in [BANC82], or simple sets of atomic values as in [OZSO85]. The recursive definition of objects allows an unbounded degree of nesting.

The following example illustrates a Researcher database schema composed of two set objects : Scientist and Contribution. Tuple is denoted by [] and set is denoted by {}.

```
[Scientist : {[name,
              education : {[degree,
                             year,
                             university]},
              age,
              member : {organization} ]},
 Contribution : {[name,
                  research : {[subject,
                               pub. : {[title,
                                        year,
                                        journal]} ]} ]} ]
```

A graphical representation of this database schema is given in Figure 1, where an arc denotes a tuple or an atom and * a set. The object Scientist gives for each scientist his education as a set of degrees and his membership as a set of professional organizations. The object Contribution gives for each scientist his research as a set of publications by subject. Note that a relational representation of this database would require five relations. Compared to the relational model, a complex object model essentially decreases the explicit use of joins (an expensive operation).

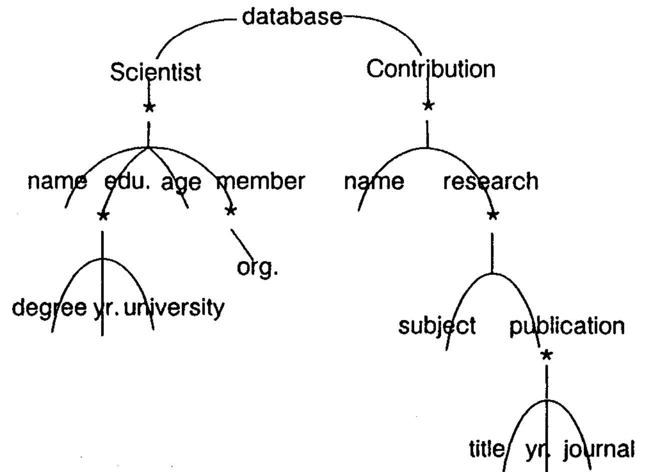

Figure 1: Example of complex object schema

2.2. Object Identity

In addition to the notion of complex object, there is a strong need for object identity [KHOS 86]. Our database language [BANC85] allows the manipulation of object identities through functions. This avoids processing of large objects when not necessary. Also, object identity allows coreferencing of objects and hence provides support of graph structures instead of trees. Each object is assigned by the system a unique identifier.

An efficient approach for representing identity at the implementation level is through the use of *surrogates* [HALL76, MEIE83]. A surrogate is a globally unique value created by the system when an object is instantiated, which the system never modifies and users are not permitted to modify. For storage efficiency an atom's identifier is the atom's value itself. We will use surrogate identifiers for tuples and sets. Furthermore the surrogate could be made invisible to the user (i.e., at the level above the complex object model).

Representing identities through surrogates allows us to have data independence, low-level support for integrity constraints, uniformity [COPE85] and provision for capturing joins [VALD85]. Note that clustering schemes are orthogonal to the existence of surrogates. The use of surrogates introduces a level of indirection through a small index which is RAM resident. However, it permits efficient updates and reorganization since references do not involve physical pointers which would cause disk accesses. Surrogates eliminate the need for user-defined identifier keys which consist of one or more attribute values. This simplifies the update process for users since all attributes can be modified in a uniform way, whereas the use of user-defined identifier keys places restrictions on updates to those attributes which serve the dual role of object descriptive data and object identity. Surrogates are

fixed-length integers and are usually smaller than user-defined identifier keys, so that the storage and processing of entity relationships are more efficient.

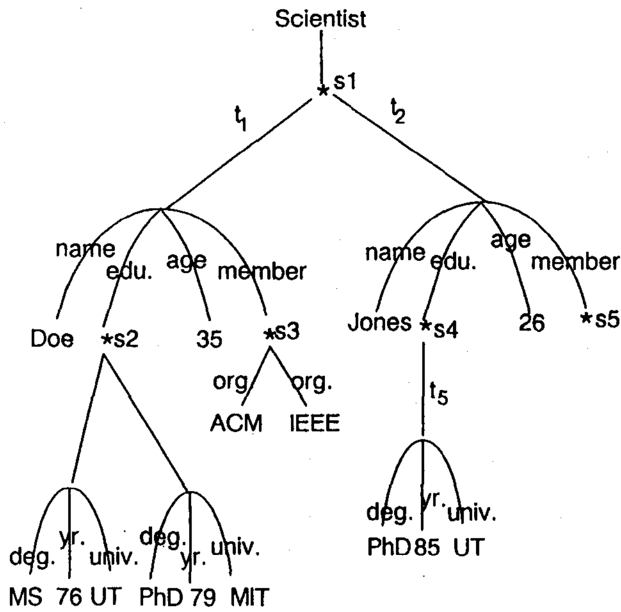

Figure 2: An instance of the object Scientist

Figure 2 gives an instance of the object Scientist where s_i is a set surrogate and t_j a tuple surrogate.

3. Direct Storage Model

In the direct storage model, complex objects are stored directly as they are defined in the conceptual schema. This is a natural way to store conceptual objects. For example if the database is composed of set objects, the direct storage model will store each set object (which can be a nested set) in a separate file. Each record of a file represents a complex object (e.g. the tuple for scientist Doe). Then, we have several solutions for clustering the attributes of a complex object. All these solutions stem from an ordering of the nested sets based on the hierarchy. A simple solution consistent with the hierarchical manipulation of objects in our language is pre-order. For instance, the internal schema of the file storing the set Scientist would be as follows (sur is a surrogate which identifies the following set or tuple) :

```
/sur/ { /sur/ [name : value,
         education : /sur/ {/sur/ [degree: value,
                                   year: value,
                                   university: value]},
          age : value,
          member : /sur/ {organization} ]}
```

The clustering of the records in a file can only be done based on attributes of the root objects. The file Scientist can only be clustered on sur, name and/or age, using a single or multi-attribute file structure. Therefore, the access to objects based on other attributes than those of the root objects must be done with auxiliary structures (e.g., secondary indices) or through sequential scans.

The primary advantage of this approach is that retrievals of entire complex objects are efficient. Compared to a mapping of a relational schema where each relation is stored in a file, this model avoids many joins. Another strong advantage of this model is that the compilation of queries that deal with conceptual complex objects is simplified because there is a 1-1 correspondance between conceptual object and internal object.

The main drawback of this approach is that performance can be hurt by large objects. All clustering techniques usually assume that a record fits in a disk page. For a direct storage model, we would choose the page equal to a track. However, even with increasing disk track capacities, it can be the case that a record does not fit in a track. For example, CAD objects could span several if not many tracks. Since we feel it is not reasonable to impose size constraints on objects, the management of large objects adds complexity in the clustering algorithms. Note that in our model, it is always possible to flatten at the conceptual level a hierarchical object and retrieve it through joins. However, this solution implies a weaker physical independency.

Finally, retrievals of certain sub-objects is inefficient because they are clustered according to a topological order. This is typically the main drawback of hierarchical systems.

4. Normalized Storage Model

In the normalized storage model, complex objects are not stored directly. Rather, they are decomposed into sets of tuples of atomic values and/or surrogates. Thus, each set object corresponds to a normalized relation. For instance, the object Scientist would be decomposed into three flat relations as shown in Figure 3. Ed-sur is a surrogate of education (set of degrees) and D-sur is a surrogate of a tuple degree. The connection between Scientist and Education is thus given by Ed-sur (i.e., the join attribute) in set Education. Note that for optimization purposes, we can replace Ed-sur by S-sur in Education because there is only one Ed-sur value per S-sur value and then remove the attribute education in Scientist.

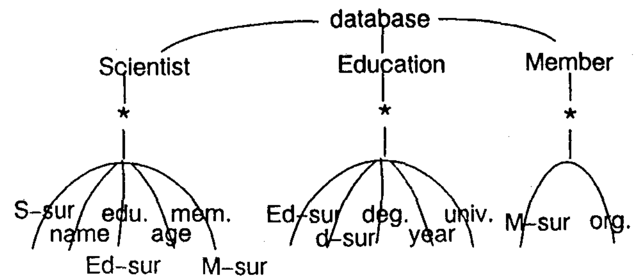

Figure 3: Normalized schema for Scientist

The main value of this normalized approach is a better performance of partial object retrievals. In turn, each relation is mapped into file(s) using a uni-relation storage structure. In section 4.1., we will discuss the alternative solutions for the mapping of relations into files. As in the relational approach, retrieval of complex objects requires joining relations. In order to make these operations efficient, we will propose in section 4.2 storage structures called join indices that store in a uniform and compact way the complex object structures.

4.1. Uni-Relation Storage Structures

In this section, we summarize the properties of the known uni-relation storage structures that affect the processing of the main relational operations (project, select, join, update). Note that operations on complex objects can be seen as extended relational operations (including transitive closure). We distinguish these structures according to two partitioning functions applied to relations called vertical and horizontal partitionings. Vertical partitioning maps relations into files, where a file corresponds to an attribute, several attributes, or the entire relation. Horizontal partitioning clusters a file based on the values of a single attribute or based on several attributes. In the following, for each possible vertical partitioning, which we name NSM, DSM and P-DSM, we discuss the possible horizontal partitionings and their performance.

4.1.1. NSM

This approach, named N-ary Storage Model, is the most commonly used in database systems. Each conceptual relation is stored in a single file. The vertical partitioning function is thus trivial. The update of tuples is thus efficient since a single file is affected.

For selections, if horizontal partitioning is performed on a single attribute then selection is most efficient for exact match and range queries on that clustering attribute. If the selection is based on inverted attributes there is considerable degradation in performance. Furthermore, if a multikey clustering scheme is utilized, the performance of selection gets better as the query binds more attributes of the multikey.

The best operation supported by NSM is projection on many attributes. Projection on a few attributes is generally inefficient since the ratio of data needed to data touched (entire file) is low.

Join is acceptably efficient only when it is based on clustered or indexed attributes and only when it is preceded by selection and projection [SELI79]. Finally, the presence of a single long attribute in the file degrades performance of all the operations based on other attributes.

4.1.2. DSM

This approach, called Decomposition Storage Model, stores all values of each attribute of a relation together on a separate file [BATO79, COPE85]. Each attribute value is associated with the surrogate of its conceptual tuple. In [COPE85], there are two physical copies per decomposed binary relation: one copy is clustered on the surrogate and the other copy is clustered on the attribute values. Having two copies of each data item is also the only good solution to reliability. The DSM approach is best suited for selection and projection on a few attributes.

Complex joins are performed through a cascade of semi-joins, and are usually very efficient. However, the result of a join phase provides only the surrogates of the tuples that match. Therefore, in a final projection phase additional semi joins are needed to associate attribute values with the surrogates. Compared with NSM, DSM requires more operations but on smaller data sets.

Also, tuple insertion/deletion has a poor performance, for it can generate as many updates as attributes.

4.1.3. P-DSM

This approach, called partial DSM, is a hybrid between DSM and NSM. This storage model vertically partitions a relation based on the attribute affinities, such that attributes which are frequently used together are stored in the same file [HOFF75, NAVA84]. The knowledge about the most frequent queries in user workloads is thus exploited to organize storage structures for efficient access. Each file contains several attributes and a surrogate of the corresponding tuple. Therefore, the operations best supported are selections and projections on the groups of attributes which are frequently accessed together.

With a general P-DSM approach some attributes might be replicated in more than one file. Since the construction of the P-DSM files is based on affinities and user hints, replication would enhance the performance of retrievals but would penalize updates.

The performance of joins depends on the partitioning by the join attributes and can be good if they are clustered or indexed. If the attribute groups are based on affinities, then the projection on many attributes should seldom involve joining different files. Updates of conceptual tuples require updating all the partially decomposed files.

The fact that P-DSM is a hybrid of NSM and DSM provides us opportunities to have a compromise of the advantages and disadvantages of both schemes. However, the accurancy of the vertical partitionning is a key factor of efficiency. Thus, this model supports poorly highly dynamic workloads.

4.2. Join Indices

In this section, we present simple data structures called join indices which capture in a uniform way the connections existing between objects. We first give the basic version of the join index called binary join index proposed in [VALD85] for simple objects and then a more generalized version called hierarchical join index adapted to complex objects.

4.2.1. Binary Join Indices

We recall the definition given in [VALD85]. Let R and S be two relations not necessarily distinct, we consider the join of R and S on attributes A from R and B from S giving a result relation. Intuitively, a binary join index (BJI), or simply join index, is an abstraction of the join of the two relations. The surrogate of a tuple of R is noted r_i and the surrogate of a tuple of S is noted s_j. More formally, the binary join index on R and S is the set

$$BJI = \{ (r_i, s_j) \mid f(\text{tuple } r_i.A , \text{ tuple } s_j.B) \text{ is true } \}$$

where f is a boolean function that defines the join predicate.

A BJI is implemented by a binary relation. For performance reasons, we may keep two copies of this relation, one clustered on r (using a B+-tree, for example) and the other clustered on s. A BJI is created by joining the relations R and S and projecting the result on attributes (r,s).

For example, the connection between Scientist and Education (Figure 3) was given by storing explicitely the surrogate of Education in both relations Scientist and Education. This connection can be stored separately as shown in figure 4.

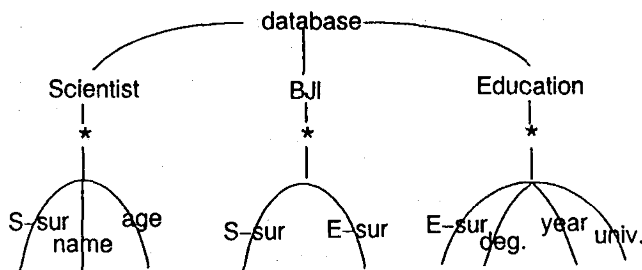

Figure 4: Example of Join Index

BJI are very efficient for optimizing joins. This is mainly because a join index is separated from base data and so small that it can fit in RAM. They can be used systematically for capturing the joins materializing complex objects. However, they can also be very useful for optimizing value based joins. For example, the join between Scientist and Contribution on name can be captured by a join index. In this latter case, a join index is an accelerator for joins.

When intended as an acceleration mechanism, BJI's should be used only for most important joins. Join indices are shown to be a very attractive tool for optimizing both relational queries and recursive queries [VALD86].

Note that binary join indices are subsumed by the DSM applied to our complex object model. In other words, with the mapping presented in Section 4.1 (see Figure 3), DSM will automatically give us the binary join indices that capture the connections between sub-objects of the same object. Therefore, join indices are subsequent decompositions which make sense for NSM and P-DSM storage models. Here we have attempted to keep the discussion of join indices more generic since their properties as join accelerators hold in many models, including the relational model.

4.2.2. Hierarchical Join Indices

In order to support complex objects, we extend the notion of join index to this of a more general structure, called hierarchical join index (HJI). A hierarchical join index can capture the structure of a complex object by using the surrogates of the connected relations involved in the whole object. Figure 5 proposes two examples of HJI for two different complex objects.

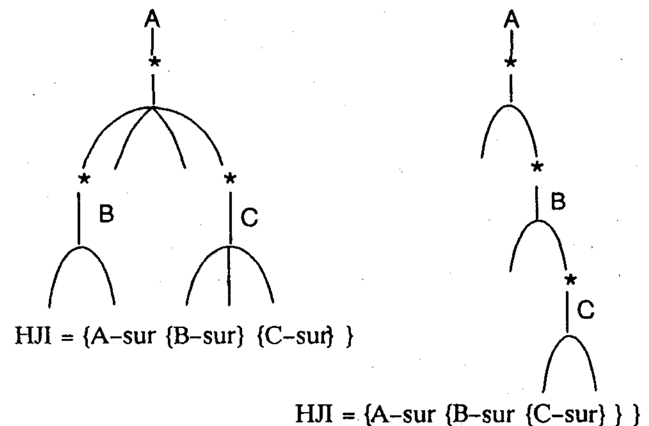

Figure 5: Examples of hierarchical join indices

Therefore, rather than having several binary join indices, a single and bigger hierarchical join index can be used. Similarly to the direct storage model, a HJI can be only clustered on the root surrogate. When the root surrogate of a complex object is obtained (through a secondary index), then the whole structure of the complex object is given directly. HJI's are better than BJI's for retrievals of entire objects. In fact the tradeoffs between HJI and BJI is very similar to the tradeoffs between the DSM and NSM storage organizations. In other words, the hierarchical scheme will always involve fewer updates, but if very few joins through the joining surrogate are performed, some retrievals will be more expensive (this corresponds to the curves of the number of projected attributes in [COPE85]).

One interesting point to remember is that we never have range queries on surrogates. Therefore, access is approximately random. Thus, in the comparison of the binary and the join indices the number of blocks accessed as a function of the number of surrogates can be approximated through Yao's function [YAO77]. The results in [COPE85] indicate that the main advantage of the DSM over NSM, when the number of projected attributes is kept constant, comes through increasing the number of selected attributes. Therefore, since the accesses for both the binary and hierarchical join indices will be "scattered", and since the HJI's provide better performance in updates, we believe this scheme presents a competitive alternative to BJI's. Furthermore, since only surrogates are stored in the join index abstraction of complex object, recursive structures could be supported very easily. However, BJI are still necessary to complete HJI in performing partial object retrievals.

5. Performance Evaluation

Since most of the research efforts have concentrated so far on NSM, an important goal of our research was to investigate the possibility of storing complex objects through DSM and binary join indices. Although, as we have indicated earlier, DSM subsumes binary join indices for our particular model of complex object representation, we like to keep our observations more generic since the implications of this combination (DSM + BJI) also apply to more normalized models such as the relational model. Furthermore, even in the framework of complex objects, some BJI's might be introduced to accelerate value based joins of complex sub-objects. These types of join indices are not subsumed by DSM but are more characteristic of the BJI's. In the rest of this paper, we will understand DSM for short of DSM + BJI's that represent sub-object connections.

At first sight this approach seems to be unreasonable since a fully decomposed storage system for complex objects will necessarily entail multiple joins for complex object construction. The semantic clustering of the complex object will be lost in the storage model. It might be argued that the direct storage representation (or one of its variants) is the only obvious storage of choice. In the previous section we attempted to present qualitative tradeoffs for the alternative storage schemes. The main problem with the decomposition scheme DSM seems to be *performance*.

However, to our knowledge, no quantitative evaluation has been done to characterize the performance issues for the range of complex object storage schemes presented in the previous sections. Therefore we are currently underway in determining quantitatively the performance issues of the decomposition schemes for programming environments which manipulate complex objects.

To this end, we have first attempted to compare the DSM + BJI storage scheme with the full NSM storage scheme. Subsequently we shall be measuring the relative performance of DSM + BJI with respect to the direct storage scheme.

The analytical results for DSM and join indices are drawn from [COPE85] and [VALD85] respectively. Section 5.1 will summarize the DSM results and Section 5.2 will summarize the results pertaining to (binary) join indices. Finally, in Section 5.3 we shall present some initial implementation results which show the relative performance of DSM and (binary) join indices combination with respect to NSM.

5.1. DSM vs NSM

In [COPE85] an anlytical model for the performances of 2-copy DSM (one copy clustered on surrogate and the second clustered on attribute values) was presented. DSM was compared against the full NSM storage model (i.e. NSM without join indices). A number of parameters were evaluated. First it was shown that, using run-length compression, the data storage requirement of DSM is more than NSM by, approximately, a factor of 2.1. Second, on the average, the number of probes for an update with DSM is worse by a factor of 3 compared to NSM. However, the most interesting part of the performance analysis was the retrieval performances of the two schemes.

A closed form analytical expression was developed, which gave the total IO requirement of a select/project/join operation as a function of the relation sizes, the number of select and project attributes, the number of joined relations, and the average number r of records retrieved from the base relations.

It was consitently observed that DSM would comparatively perform better if the selectivity (i.e., r/(number of tuples in base relations)) is beyond a certain threshold (in most cases approximately 1%). In fact the performance curves showed that the ratio of number of blocks accessed by NSM divided by the number of blocks accessed by DSM *as a function of r* achieves an optimum when r is approximately 10% of the average number of tuples in the base relations. Figure 6 shows a family of curves where the number of projected attributes is varied for a select on one base NSM relation. ("npa" stands for number of projected attributes from each base relation; "nb" stands for the number of NSM blocks retrieved; "db" stands for the number of DSM blocks retrieved: therefore the y-axis presents the ratio (total IO for NSM)/(total IO for DSM)). It is assumed that base NSM relations have, on the average, 10 attributes. As can be seen from these curves that for small values of r and/or larger number of projected attributes DSM looses. One could argue that the cases where DSM wins (namely selectivity greater than 1% and percent of projected attributes greater than 50%) are the least interesting.

However, thsese family of curves do not involve any NSM joins. Figure 7 shows another family of curves where here the number of preojected attributes is hald fixed, but the number of NSM joins (i.e.,"njr") is varied. Observe that the relative performance of DSM increases as the number of joins is increase. Therefore, if we were to retrieve a complex object of several levels of nesting (which would involve multiple NSM joins), and few attributes from each level, we would expect DSM to outperform NSM in most cases. We shall further substantiate this argument in Section 5.3.

5.2. Join Indices

We now summarize the analytical results of the BJI's performance given in [VALD85] and relate them to complex object retrieval. Join indices are useful with any vertical partitionning function (DSM, P-DSM, NSM). However, the mapping of our complex object model into DSM automatically implies BJI's that capture the connections between sub-objects of an object. The purpose of this section is to show that, more generally, join indices provide excellent performance in doing arbitrary joins and can outperform the best known join algorithms. In [VALD85], we limited our analysis to the join algorithm itself since it is the most critical operation. However, the real value of join indices increases as queries become complex because the most complex operations are done on small data structures (select indices, join indices, etc). The join algorithm using a join index takes advantage of all available memory and is easily adaptable to parallel execution.

In order to evaluate the performance of the join algorithm using BJI ,noted JOINJI, we compared it against the hybrid hash join algorithm [DEWI84], noted JOINHH, because this latter is very efficient (it outperforms easily the sort-merge join algorithm), takes advantage of large RAM and is ameanable to parallel execution [DEWI85]. Except for highly selective joins (i.e., producing a small result), JOINJI outperforms JOINHH. The reason is that the efficiency of a join index is inversely proportional to its size. A tuple in a BJI is small. The size of the BJI depends on the join selectivity factor , noted JS, which determines the number of tuples in the BJI. If the join has good selectivity (JS is low), the join index is small. This is a frequent case in existing databases (e.g. join on foreign key). However, a join of poor selectivity, which can be close to the Cartesian product, can make the index quite large. In this case, we claim that no good optimization is possible and a simple nested loop join algorithm is sufficient. The way in which joined relations are physically clustered have generally an impact on join performance. Surrogates contained in a join index are used for retrieving attribute values in relations. Therefore, a file mapping the relation must be either clustered or indexed on surrogate.

Assuming a conventional architecture, the main parameters affecting performance are : the number of pages in an operand relation, the number of tuples in an operand relation, the number of RAM pages available to the operation, the join selectivity factor and the semi-join selectivity factors. The RAM size allocated to the join was generally 5% of the operand relation sizes. With high join selectivity (low JS), JOINJI can outperform JOINHH by two orders of magnitude. Having the joined relations clustered on surrogate instead of indexed on surrogate improves join's performance by a factor 2. For less selective joins, the performance difference between JOINJI and JOINHH is much less. Note that we did not take into account the performance degradation of hashing in presence of many collisions that arise for low join selectivities. Finally, varying the RAM size does not change the performance difference Therefore, we feel that JOINJI would almost always outperform JOINHH.

5.3. Implementation Results

To substantiate some of the claims made in [COPE85] and [VALD85], we implemented a fully decomposed storage scheme based on DSM and BJI. The implementation is based on WISS [CHOU83] -the Wisconsin Storage model. To compare DSM (+ BJI) with NSM we have run some tests based on the Wisconsin Benchmarks [BITT83]. However, we are in the process of augmenting the Wisconsin benchmarks to test the performance of the decomposition scheme for complex object retrievals.

Thus far the implementation results confirmed our analysis. In Figure 8 we give the ratio for number of blocks accessed by NSM/DSM for 1% and 10% selects on 10K tuple relations. In the Wisconsin Benchmarks the projection was done on all of the attributes. However, we have varied the number of projected attributes in our runs. Therefore the x-axis is the fraction of the total number of projected attributes (the total number is 16 - 1 surrogate and 15 attributes). First observe that the relative performance of DSM is better when the selectivity is higher. In fact with 10% selectivity, DSM will perform better if the fraction of projected attributes is less or equal to approxiamtely 70% of the total number of attributes. With 1% selectivity DSM is better only if the total number of projected attributes is about 30% of the total number of attributes.

To show the performance of the joins with DSM versus NSM, we have analyzed the performance of several types of 2-way joins, namely: 1-1, 1-10, 10-1, and 10-10. These are illustrated in Figure 9. For all these joins, there is a 1% select on the first relation and both relations are 10k tuple relations. First we note that the total number of tuples retrieved from both relations decreases in the order 1-10, 10-10, 1-1,and 10-1. For example, with 1-1 a total of 200 (100 from each relation) tuples will be retrieved. However, with 1-10 a total of

1100 tuples will be retrieved (100 from the first relation (i.e., the one on which the selection is performed) and 1000 from the second). For the 1-10 and 10-10 joins DSM performs better of the number of projected attributes is approximately less than or equal to 50% of the total number of attributes in both relations. For the 1-1 and 10-1 cases DSM performs better if the total number of projected attributes is less or equal to approximately 30% of the total numebr of attributes. However, we should emphasize that the x-axis here is the fraction of the *total* number of attributes from both relations. In other words a 30% fraction represents 10 attributes (and not 5 as in the previous selection curve).

We feel these results are preliminary. In particular, these performance measurements were made with at least 1% selectivity of retrieval queries. When selectivity is reduced to a single object, our analytical model predicts that the worse case factor for DSM (all attributes projected) becomes much higher. This is clearly shown with the update queries. Our goal is to analyze the performance of DSM for a mix of typical queries in programming environments which manipulate complex objects. Furthermore, we will be comparing the decomposition scheme against the direct storage scheme to have a precise appreciation of the implied performance issues due to the decomposition of the complex object.

6. Conclusion

A complex object storage model must be able to provide efficient support for a wide variety of query types. The difficulty lies in achieving two conflicting goals : efficient support for retrieving a single entire complex object and at the same time retrieving its components. The first goal leads to clustering of a full complex object in the same memory extent (i.e., direct), while the second goal leads to clustering of the individual components (i.e., DSM).

The relative advantages of DSM (+ BJI's capturing sub-objects connections) and direct can be summarized as follows:

(1) DSM is significantly simpler than direct to implement. Storage structures, clustering, indexing and compilation of conceptual queries into internal queries are much simpler.

(2) DSM is significantly simpler than direct to use. Users and database administrators need not be involved in deciding which attributes to cluster or index. Instead, these are done in a uniform way. Also, reorganization due to such performance tuning is not needed.

(3) DSM causes significantly less system resources for reorganization due to either performance tuning or conceptual schema modification.

(4) DSM causes access to significantly more physical blocks when the number of projected attributes is large and selectivity is low.

(5) When locality of use among attributes is higher than locality among complex objects, DSM causes fewer disk IOs (accesses to physical blocks are more often in RAM), since individual attributes can more easily be buffered. We expect this to usually be the case, since each application or user view uses a fixed pattern of attributes but varies predicate bindings as a parameter.

(6) In a parallel disk machine, DSM can more easily achieve load balancing then direct. A hot complex object may cause one disk to be overloaded using direct, whereas DSM can spread the attributes of the complex object over several disks.

In conclusion, DSM has many advantages over direct. Its severe disadvantage is point (4) above. DSM is superior whenever data is shared over multiple applications, since direct can provide optimal tuning only for a particular access pattern. Direct is superior when a particular access pattern heavily dominates and that access pattern consists of accessing few objects (very low selectivity), projecting on many attributes , and using the same attributes as the selection criteria. However, such applications currently exist in sufficient number (e.g., CAD) to warrant database system support. Two open issues remain regarding the form of support of such applications.

One open issue is that currently these applications are usually supported by file systems which store each complex object as a long bit/byte string file with indexing by file name. This approach could be supported within DSM by representing the complex object as a single attribute whose value is a long string. In other words, if the complex object is always used as a single monolithic object within the database system, then a complex direct representation is unnecessary.

The second open issue is whether such applications will continue to have a single access pattern. Most long term visionaries of CAD, for example, argue that eventually CAD data will be heavily shared by multiple applications.

Acknowledgements:

The authors wish to thank Francois Bancilhon, Haran Boral and Marc Smith for their helpful comments on this research and Thomas Jagodits who was involved in the implementation effort of the normalized storage model for complex objects.

References

[ADIB84] Adiba M., Nguyen G.T., "Handling Constraints and MetaData on Generalized Data Management Systems" Int. Workshop on Expert Database Systems, Kiowah Island, South Carolina, October 1984.

[BANC82] Bancilhon F., Richard P., Scholl M., "On Line Processing of Compacted Relations" Int. Conf. on VLDB, Mexico, September 82.

[BANC85] Bancilhon F., Khoshafian S., Valduriez P., "FAD, a Database Machine Language: Formal Semantics" MCC Internal Report, December 1985.

[BANC86] Bancilhon F., Khoshafian S., "A Calculus for Complex Objects" Proc. of ACM Symp. on PODS, Boston, March 1986.

[BATO79] Batory D.S., "On Searching Transposed Files"" ACM Trans. on Database Systems, vol. 4, no. 4, December 1979.

[BATO85] Batory D.S., Kim W., "Modeling Concepts for VLSI CAD Objects" ACM Trans. on Database Systems, vol. 10, no. 3, September 1985.

[BITT83] Bitton D., DeWitt D.J., Turbyfill C., "Benchmarking Database Systems : A Systematic Approach" Int. Conf. on VLDB, Florence, September 1983.

[CHOU83] Chou H.T., DeWitt D.J., Katz R.H., Klug A.C., "Design and Implementation of the Wisconsin Storage System" Technical Report #524, Dept of Computer Sciences, U. of Wisconsin, Madison, November 1983.

[COPE85] Copeland G., Khoshafian S., "A Decomposition Storage Model" ACM-SIGMOD Int. Conf., Austin (Texas), May 1985.

[DEWI84] DeWitt D.J. et al., "Implementation Techniques for Large Memory Database Systems" ACM-SIGMOD Int. Conf., Boston, June 1984.

[DEWI85] DeWitt D.J., Gerber R., "Multiprocessor Hash-Based Algorithms" Int. Conf. on VLDB, Stockholm, August 1985.

[HALL76] Hall P. et al., "Relations and Entities", Modeling in DBMS, edited by Nijssen (North-Holland 1976).

[HASK82] Haskin, R., Lorie, R., "On Extending the Functions of a Relational Database System" ACM SIGMOD Int. Conf., Orlando (Florida), June 1982.

[HOFF75] Hoffer J.A., Severance D.G., "The Use of Cluster Analysis in Physical Database Design" Proc. of 2nd Int. Conf. on VLDB, 1975.

[JARK84] Jarke M., Koch J., "Query Optimization in Database Systems" ACM Computing Surveys, vol. 16, no. 2, June 1984.

[KHOS86] Khoshafian S., Copeland G., "Object Identity" to appear in Proc. of ACM Conf. on OOPSLA, Portland (Oregon), October 1986.

[LUM85] Lum V., et al. "Design of an Integrated DBMS to Support Advanced Applications" Int. Conf. on Foundations of Data Organization, Kioto, May 1985.

[MEIE83] Meier A., Lorie R., "A Surrogate Concept for Engineering Databases" Int. Conf. on VLDB, Florence (Italy), October 1983.

[NAVA84] Navathe S., Ceri S., Wiederhold G., Jinglie D., "Vertical Partitionning Algorithms for Database Design" ACM Trans. on Database Systems, vol. 9, no. 4, December 1984.

[OZSO85] Ozsoyoglu G., Ozsoyoglu Z.M., Mata F., "A Language and a Physical Organization Technique for Summary Tables" ACM-SIGMOD Int. Conf., Austin (TX), May 1985.

[SELI79] Selinger P. et al., "Access Path Selection in a Relational Database Management System" ACM SIGMOD Int. Conf., Boston (Mass.), May 1979.

[VALD85] Valduriez P., "Join Indices" MCC Technical Report Number DB-052-85, Submitted for Publication, July 1985.

[VALD86] Valduriez P., Boral H., "Evaluation of Recursive Queries using Join Indices" Proc. of First Int. Conf. on Expert Database Systems, Charleston, April 1986.

[YAO77] Yao S.B., "Approximating Block Accesses in Database Organizations" Comm. ACM, vol. 20, no. 4, April 1977.

[ZANI85] Zaniolo C., "The Representation and Deductive Retrieval of Complex Objects" Int. Conf. on VLDB, Stockholm, August 1985.

Figure 6: Varying the Number of Projected Attributes

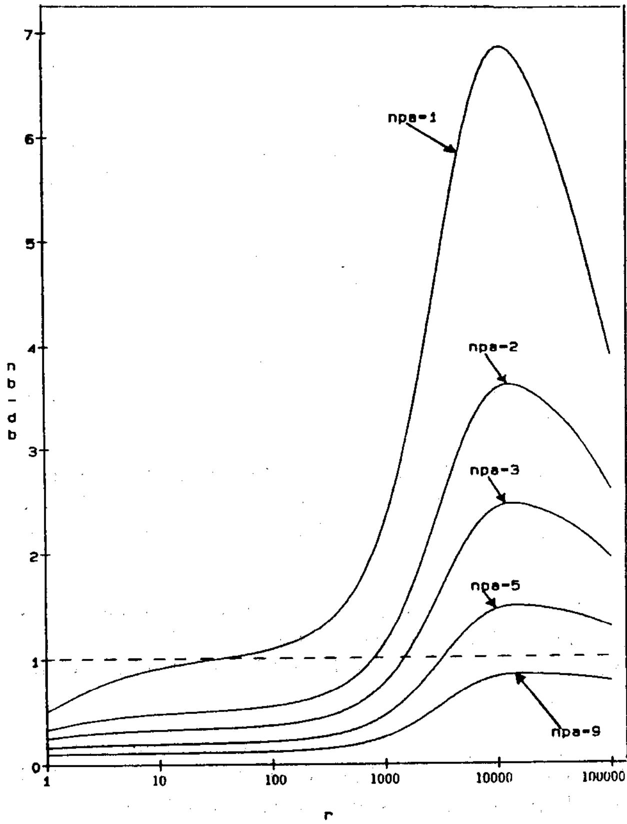

Figure 7: Varying the Number of Joined Attributes

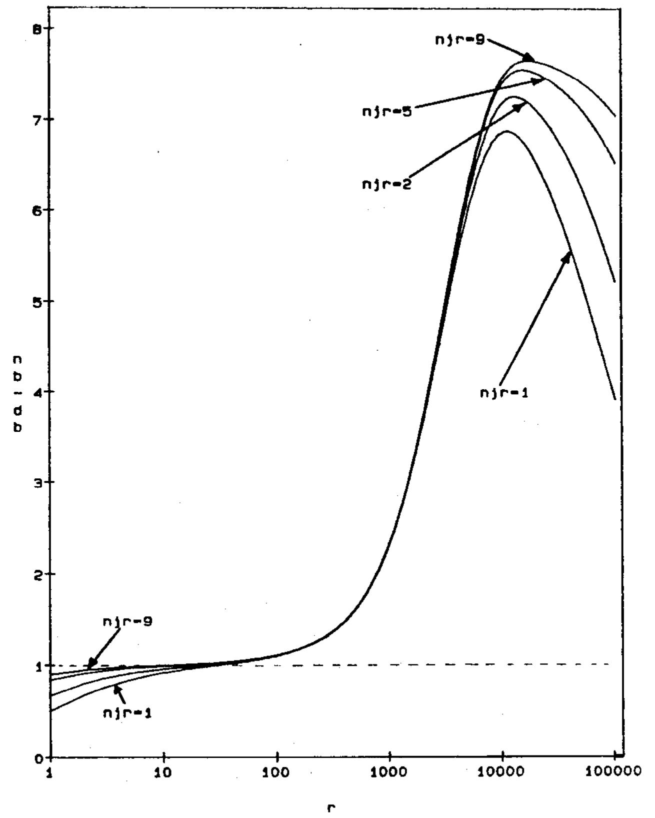

Figure 8: Selections

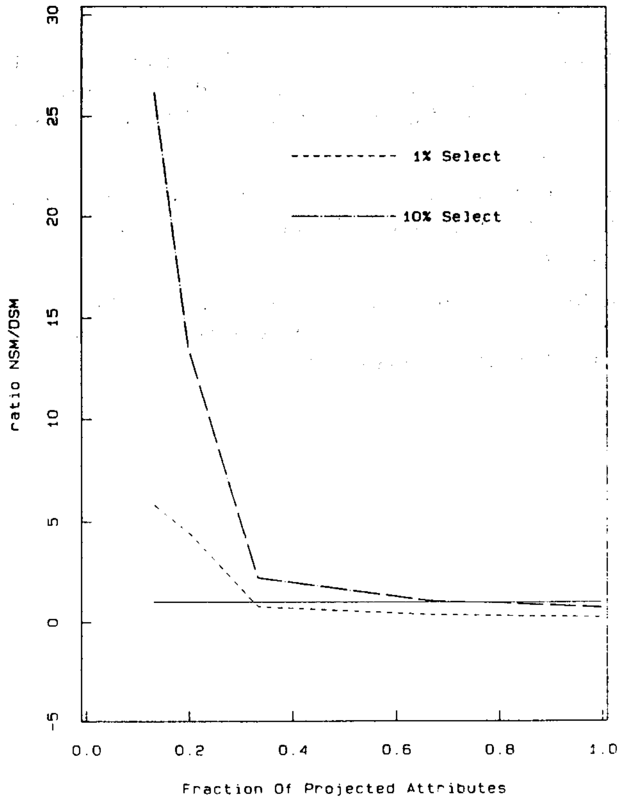

Figure 9: Joins