Evaluating View Materialization Strategies for Complex Hierarchical Objects*

Matthew C. Jones and Elke A. Rundensteiner
Software Systems Research Lab
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI, 48105-2122
{mjones,rundenst}@eecs.umich.edu

Phone: 313-936-2971 **Fax: 313-763-1503**

Abstract

In many design applications, it is common practice to store complex hierarchical objects in a compact folded form to save storage space and to reduce processing costs for accessing the objects. In these folded representations, complex objects are built up from identical and otherwise indistinguishable design objects. However, it is often necessary, especially during the refinement of data, to distinguish between these identical folded objects by personalizing a subset of them. The established practice is to explicitly unfold the hierarchical objects and thus create space in which to store distinct personalization data for each object occurrence. However, this explicit unfolding is costly and time consuming, resulting in a potentially much larger structure, and substantially increasing the costs of querying and updating the design. Therefore, we propose an `unfold` *view operator and provide the basis for updating of customized values for each hierarchical sub-object through the unfolded view. We propose alternative strategies for the maintenance of personalization values, representing various portions of the view materialization spectrum. We present a performance evaluation comparing these strategies as well as the traditional explicit unfolding approach. Our evaluation indicates the trade-offs in terms of storage and query costs and compares the costs to do implicit unfolding through a view rather than explicit unfolding of complex hierarchical objects.*

Keywords: hierarchical object model, unfolding of hierarchical structures, data transformations, object-oriented database views, specialized view materialization, performance evaluation

* This work is supported in part by the NSF RIA grant #IRI-9309076, NSF NYI grant #IRI-94-57609, and the University of Michigan Faculty Award Program. We are also grateful to Digital Equipment Corporation for fellowship support during early development of this work.

1 Introduction and Problem Description

Introduction. An increasing number of applications must be able to store, query, and retrieve complex hierarchical objects. To reduce their size, especially when they contain repeated sub-structures, complex hierarchical objects are often stored in a compact, folded representation. The folding most frequently occurs along the *part-of* relationship, and is accomplished by representing the relationship by multiple references to identical objects rather than by replicating subobjects. The result is a compact hierarchical representation that stores the distinct objects in the design *implicitly,* via *occurrence paths* in the folded representation rather than through explicit object instances. For example, Figure 1(a) represents a stack of two cubes c1 and c2. Each cube is represented by 6 faces. The bottom face of the cube `c1` in the stack is represented by the occurrence path `c1::btm` in the DAG. Similarly, the "most-significant" full-adder (FA) in the electrical design shown in Figure 1(b) is represented implicitly by the occurrence path `a3::f3` in the electrical design object. The representation of explicit objects as folded implicit occurrence paths is a very powerful construct, and as a consequence is commonly used in application domains where there are repeated substructures. These domains include mechanical and materials design, electrical and electronic design, genome databases, and graphical applications containing repetitive images created by tiles and nested graphical objects, to name a few.

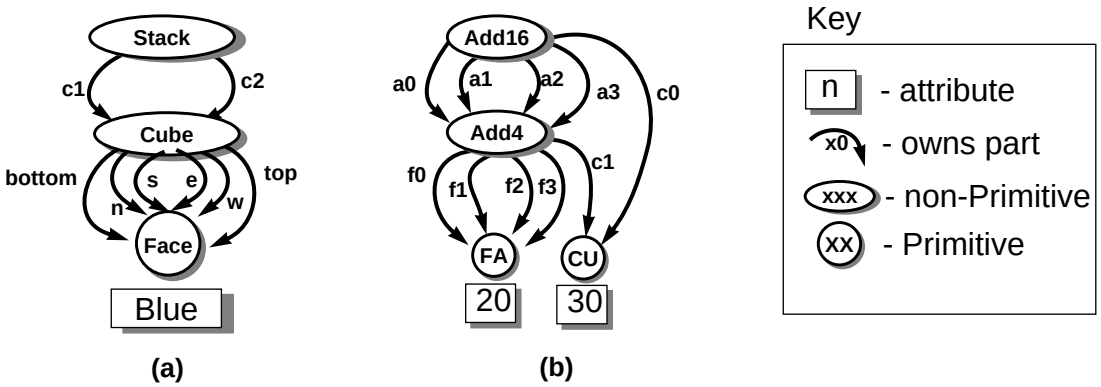


Figure 1: (a) Hierarchical Graphical Object (b) Hierarchical Digital Design Object.

During the design process a user eventually requires an unfolded view of the data. While modern object-oriented database systems excel at representing folded hierarchical structures [13, 22], the meaning of the folding structure is not known to the database. As a consequence these databases are not capable of defining implicitly unfolded database views on a folded representation. Software tools external to the database usually perform the unfolding explicitly, creating a separate representation [23], resulting in the many known problems caused by redundant data replication. Our hierarchical set model [16] enables a database view system to create an unfolded view of the folded structures automatically and implicitly, without creating a separate representation. Our model offers opportunities for perfor-

mance improvement and query optimization not possible with a translator-based approach to unfolding.

The Personalization Problem. During the refinement of complex folded objects, applications typically need not only to process queries on the complex objects. They also need to perform updates through the unfolded view. For example, in Figure 1(b), we may personalize the full adder (FA) object occurrence `a3::f3` through an unfolded view by updating the *size* attribute, while leaving the *size* attribute of all other 15 occurrences of the full adder object unchanged. The *personalization* problem is concerned with answering the following questions:

- What data structures can allow updates of personalized attribute values in the implicit objects created by the *unfold* operation?
- How do we organize these structures so that data can be efficiently retrieved during a representative set of queries executed on an unfolded view?
- What are the performance trade-offs in querying the spectrum of un-materialized, partially-materialized and fully materialized unfolded view?

In this paper, we focus on the costs to store and to query an unfolded view which has been updated (personalized).

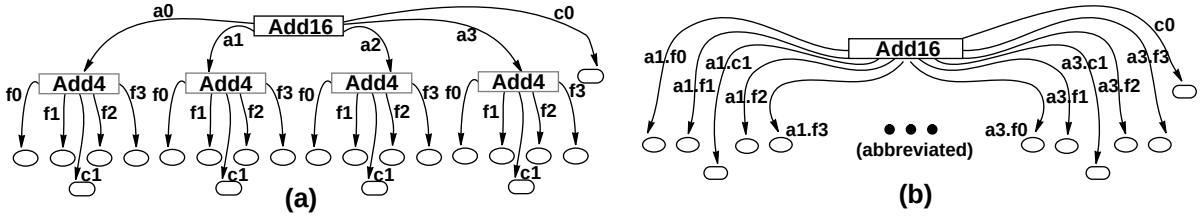


Figure 2: (a) Unfolded Hierarchical and (b) Unfolded Flat Design Descriptions of Figure 1(b).

Current Practice. The traditional method for addressing the personalization problem is to *explicitly* unfold hierarchical database objects to create objects in which to store the personalization values [23]. For example, Figure 2 depicts the unfolded representations of Figure 1(b) with the hierarchy preserved (Figure 2(a)) and removed (Figure 2(b)). Work in [14] and [3] demonstrates approaches to maintaining fully materialized unfolded views. The redundant storage of folded attributes in the unfolded representation not only increases storage space, but it also complicates data consistency maintenance and degrades query performance.

The HS System [24] is the only research system we know of that supports an application programmer's interface (API) to an implicitly unfolded electrical design database. Besides being limited to access via the API, HS does not provide support for personalization of the data. Furthermore, it relies on a special encoding of the data that mandates a complete traversal of the folded design after any update to the folded structure. In the commercial ECAD community, there are systems such as the

Mentor Graphics Design Viewpoint [23]. While these systems provide implicit unfolding and personalization, access to the design objects is provided by a limited industry-standard API [8], rather than a query language, precluding opportunities for *ad-hoc* query specification and query optimization.

Our Contribution. In this paper, we present our model of folded hierarchical structures, including operations that formally describe implicit unfolding. This model and its operations are the basis for defining unfolded and flattened views on complex hierarchical objects as well as the identification and formalization of the personalization problem for folded hierarchical structures. Besides the status quo explicit unfolding solution, we propose two approaches to solving the problem, one adapted from the literature – representing the un-materialized end of the view spectrum, and one which we designed – representing the partially materialized portion of the spectrum. In addition to these approaches we designed and fine-tuned pruning and clustering techniques to improve their performance. We implemented these two personalization strategies and a fully-materialized explicit unfolding strategy, as well as pruning and clustering techniques in a uniform test-bed implementation in order to provide a fair performance comparison of the approaches. Within the test bed we ran extensive experiments varying parameters such as data characteristics, database buffer sizes and percentage of personalization. In this paper we present the results of our performance evaluation along with recommendations for when the different approaches are appropriate. These recommendations can provide guidance to object database designers considering view materialization strategies for domains requiring the unfold view operation.

Structure of This Paper. In Section 2, we define our hierarchical set model. In Section 3, we present three alternative solutions to the personalization problem, representing the spectrum of view materialization approaches. We describe the performance evaluation process in Section 4, and review the results in Section 5. Related work is discussed in Section 6, and we conclude with Section 7.

Table 1: Notation for Collection Types.

	Empty	**Addition x is element.**	**Singleton x is element.**	**Combination**	**Defining Domains N_1, N_2 are domains.**
Lists	[ ] *nil*	x::L *cons*	[x]	L_1 @ L_2 *append*	[N_1] all sequences of N_1
Tuple	< >	x ; T	< x >	$T_1.T_2$	< N_1, N_2 > all tuples of N_1 and N_1
Sets	{ }	x $\uparrow$ S	{ x }	S_1 U S_2	{ N_1 } all sets of N_1
Common	**empty**	**add(x,C)**	**sng(x)**	**comb(C_1, C_2)**	

2 The HierSet Model and Personalization.

2.1 The Model

Our HierSet model is composed of various types of collections. Table 1 summarizes the notation

that we use to refer to these collections and domains defined with them. The notation is taken from [1].

Our hierarchical set model consists of *HSets* from the domain η and *Abstractions* from the domain α. We say that HSets are constructed as sets of Abstractions, $\eta \rightarrow \{ \alpha \}$. For example, in Figure 3, we have the HSet η_a consisting of three Abstractions, $\eta_a = \{ \alpha_1, \alpha_2, \alpha_3 \}$. This composition is represented by the three labelled edges whose source is η_a. We distinguish empty HSets, called *Primitives*. In Figure 3, both η_d and η_e are Primitives.

We define the function A: $\alpha \rightarrow \eta$. This function represents the *abstractionOf* relationship. For example, we say that α_1 is an *abstractionOf* η_b, denoted by $A(\alpha_1) = \eta_b$. In Figure 3, this is depicted by an edge in the graph labeled α_1 whose destination is η_b. The membership of α_1 in the HSet η_a defines the composition relationship between η_a and η_b, namely that η_a owns an abstraction of (*ownsAbstractionOf*) η_b. Thus there is an edge in the graph with a source of η_a, a destination of η_b, and a label of α_1.

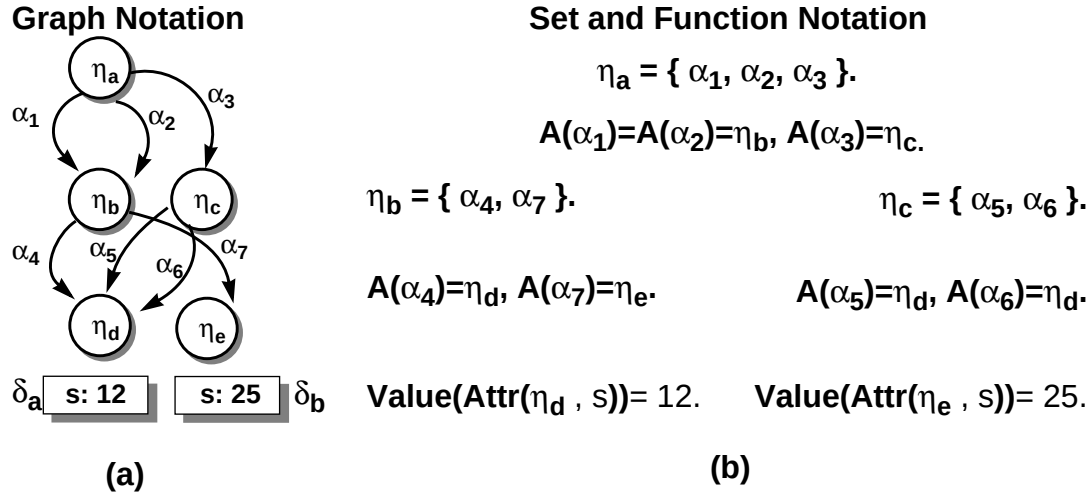


Figure 3: A HierSet rooted at the HSet η_a. (a) Graph Notation, (b) Set and Function Notation.

We distinguish a single HSet as the *root*, and refer to the root HSet, together with all other HSets reachable from the root via the *ownsAbstractionOf* relationship as a *HierSet*. Because a HierSet is always constructed from composition relationships such as *part-of* or *owner-of*, a HierSet can always be expressed as a directed acyclic graph (DAG). Cycles in the graph would result in infinite HierSets which are not currently of practical interest. Throughout this paper, we will refer to a HierSet's graphical properties and call it a HierSet DAG. Figure 3 depicts a HierSet DAG rooted by the HSet η_a.

Let χ denote the domain of sequences we refer to as *contexts* and N the set of natural numbers, such that $\chi \rightarrow [N]$. Given a domain of tuples $\Omega \rightarrow < \chi, \eta >$, we refer to an object $o \in \Omega$ as an *occurrence*. The first component of an occurrence, the context, identifies the path in the HierSet DAG that represents the implicitly defined occurrence. The second component, the HSet, identifies the primitive

HSet at the end of the context path. For example, in Figure 3, <[1,4], η_d> is an occurrence of the primitive η_d, resulting from η_d owned by η_b which is in turn owned by η_a. There clearly can be multiple occurrences of the same primitive in a HierSet. For example, the occurrences <[1,4], η_d> and <[2,4], η_d> in Figure 3 both contain the HSet η_d.

We define the function `ID` which maps each abstraction to a unique number within its owner HSet in *N*, `ID` : $\alpha \rightarrow N$. For convenience, we define `ID` for all abstractions α_i in Figure 3 such that `ID`(α_i) = *i*. If each Abstraction within an HSet has a unique ID, then each occurrence path in the HierSet is guaranteed to have a unique context. This is easily proven by induction.

Let δ denote the domain of *attributes*. Given an attribute $d \in \delta$, we define two functions `Name` : $\delta \rightarrow$ `String`, (a 1-to-1 function) and `Value` : $\delta \rightarrow N$. For convenience, and without loss of generality, we assume that all attribute values are from the domain *N* for the remainder of this paper. In Figure 3, for example, we have attributes δ_a, and δ_b, both with the `Name` *s*, and with `Value` 12 and 25 respectively.

We define a function for primitive HSets, `Attr` : η, `String` $\rightarrow \delta$ which is used to access the folded attribute value. For example, in Figure 3, *Attr*(η_d , *s*)=δ_a.

2.2 Operations on HSets

Using the notation described in Section 2.1, we now present the definition of various operators and transformations that are important to defining the explicit and implicit unfolding operations. To help clarify these definitions we will relate them to the example illustrated in Figure 4.

We begin by defining what it means to traverse an edge in the HierSet DAG. Given a path to an HSet identified by the context *c*, and an abstraction *a* within the HSet, we define the function CA as:

$$CA(c,a) = < c::ID(a), A(a) >, \text{ where } c \in \chi, \text{ and } a \in \alpha. \quad (1)$$

For example, in Figure 4, we see that: $CA([1], d_1) = < [1,1], Drawer>$. For syntactic convenience, we sometimes rewrite the function as:

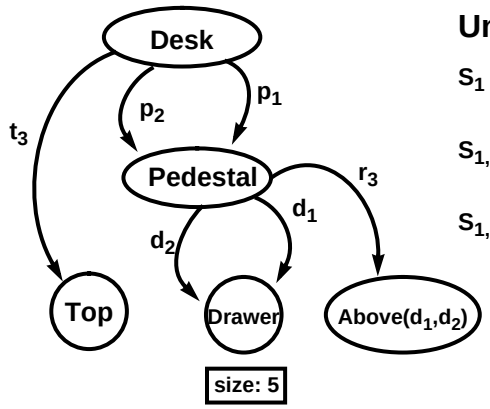
$$CA(c,a) = CA_c(a). \quad (2)$$

The function `map(f)(S)` applies the function `f` to every element of the collection *S* and combines the results into a new collection using the appropriate `comb` operator for *S* (see Table 1). We use the `map` function to apply CA_c to all abstractions in a particular HSet. This permits us to define the recursive `Unfold` function for a HierSet:

$$Unfold(< c,h >) = < c, h >, \text{ for h} \in \eta \text{ and h is a primitive.} \quad (3)$$

$$Unfold(<c, h>) = map(Unfold)(map(CA_c)(h)), \text{ for h} \in \eta \text{ and h not a primitive.} \quad (4)$$

The innermost *map* operation in Equation 4 transforms the set of abstractions to a set of occurrences within the specified context. The outermost *map* recursively unfolds each of the occurrences in the set. The basis for the recursion is an empty HSet (primitive) as shown in Equation 3. The end result of the *Unfold* operation is a set of occurrences of the form <c, h>, where *c* is the constructed context path and *h* is a primitive HSet. For example, Figure 4 shows the steps in unfolding the HSet Desk, detailing the results of each recursive *unfold*. The first level of the *Unfold* operation maps $CA_{[]}$ over every element of the root HSet, resulting in a set of three occurrences (S_1). The recursive call applies Unfold to each of the elements in S_1, resulting in $S_{1,1}$, $S_{1,2}$, and {*<[3], Top>*}. Because the occurrence *<[3], Top>* is a primitive, it is a basis for the recursion. The final result of the Unfold is the union of these sets as described in Figure 4. The final result contains 7 occurrences.



Unfold(<[],Desk>) = $S_{1,1} \cup S_{1,2} \cup$ {<[3], Top>}

S_1 = map($CA_{[\,]}$)(Desk) = { <[1], Pedestal>, <[2], Pedestal>, <[3], Top>}

$S_{1,1}$ = map($CA_{[1\,]}$)(Pedestal) = { <[1,1], Drawer>, <[1,2], Drawer>, <[1,3], Above(d_1,d_2)>}

$S_{1,2}$ = map($CA_{[2\,]}$)(Pedestal) = { <[2,1], Drawer>, <[2,2], Drawer>, <[2,3], Above(d_1,d_2)>}

Figure 4: Unfolding of a HierSet describing a Desk.

In the Desk HierSet, we see an example of a primitive relationship *Above* which relates all occurrences of the d_1 and d_2 abstractions of the drawer objects. These primitive relationships are used to describe relationships between objects in the HierSet. They are folded and unfolded in the same way as the other primitives in the HierSet.

We define a function $Attr : \Omega, String \rightarrow \delta$ which maps an occurrence and an attribute name to an attribute. This function is used to access values for attributes of unfolded objects. For example, in Figure 4, *Value(Attr(<[1,2], Drawer>, size))= 5*. We can update attributes of unfolded objects via an update function: *Update*(Ω, String, N). This function permits the update (personalization) of an unfolded object in an unfolded view. For example, we can update the size of one of the four occurrences of the Drawer object in the desk. *Update(<[1,2], Drawer>, size, 10)* implies that *Value(Attr(<[1,2], Drawer>, size))= 10*. Recall that all other occurrence values remain unchanged under this update.

Having defined the unfolding and update operations, we can define what it means for a HierSet to

be folded. A HierSet containing primitive HSets *h* is folded with respect the attribute *d* named *s* if:

$$\text{for every } c \in \chi \text{ and every primitive } h \in \eta, \; Attr(< c, h >, s) =_{id} Attr(h, s). \quad (5)$$

This is to say that for an HSet, if all occurrences containing the same primitive HSet *h* have the same (i.e. same identity) attribute *d*, then the HierSet is fully folded. From this definition in Equation 5, we make the following observations about possible implementations of the Attr() function:

1. If an HSet is fully folded with respect to the attribute *s*, then we may store the attribute with the primitive HSet that terminates the occurrence path. This permits the attribute to be shared by all occurrences that contain the HSet.
2. If attributes are shared as described in item 1, an update to a folded attribute results in the update to all occurrences which share the attribute. We call this update an *out-of-context* update. This follows from the definition of fully folded as presented in Equation 5.
3. An update to an unfolded attribute cannot be shared as described in item 1, because there is no space allocated in an HSet in which to store the values for unfolded attributes.

These observations form the foundation and motivation for providing the implicit unfolding of hierarchical complex objects while permitting personalization of the structures. In the remainder of this paper, we explore implementations of the `Attr()` function that optimize performance for the kinds of queries commonly performed on complex hierarchical structures.

As we will describe in Section 4, the queries that we consider for our performance evaluation are aggregation queries and queries that traverse hierarchical graphs described by a HierSet (both nodes and edges are primitives). These classes of queries are representative of the kinds of queries that are frequently performed on hierarchically specified design objects and hierarchical graphs. We do not consider indexed queries because we have not yet developed index structures for all materialization strategies to permit us to compare them fairly.

3 Materialization Methods

In this section, we describe three different approaches to materializing a personalized unfolded view. First, we present explicit unfolding (EU), a fully materialized approach which stores personalization data in each explicitly and fully unfolded object. Second, we describe a dictionary personalization (DP) approach that uses a B+ tree structure to manage personalized values for implicitly unfolded object occurrences. We chose the B+ tree over other dictionary methods because of its availability in existing database systems and for its uniform performance characteristics. Third, we also present our own partial materialization approach that uses *partial unfolding* (PU) to personalize attributes in an auxiliary HierSet structure. We designed the PU approach to represent an approach near the middle of

the materialization spectrum.

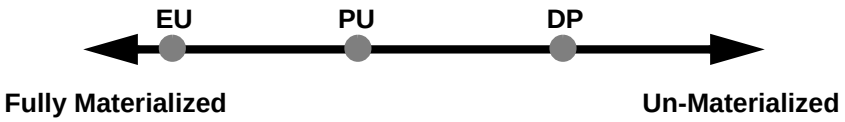


Figure 5: The Materialization Spectrum for Unfold

3.1 Parameters

Because the performance of these proposed methods depends upon the characteristics of the data being personalized, we briefly describe the characteristics of the HierSets and other parameters that we will be discussing before describing the personalization approaches.

- h : height of the HierSet DAG. This quantity describes the length of the longest path from the root HSet to a leaf (i.e., primitive HSet). Our generated HierSets also contain paths shorter than h.
- d : density of the HierSet. This quantity describes the average number of abstractions contained in each HSet of the design. In our generated HierSets these abstractions refer to both primitive and non-primitive HSets.
- n : the number of occurrences in the HierSet. This is the same as the number of paths in the HierSet DAG. Because of uniformity in our generated HierSets, we express n in terms of h and d as:

$$n \approx d^h . \tag{6}$$

- v_a : the number of personalized values for the attribute a.
- We denote the size (on secondary storage) of an object pointer, an object ID, an Abstraction, and a non-primitive HSet in our model by s_{ptr}, s_{id}, s_a, *and* s_h respectively.
- We denote the size of a primitive HSet and all of it attributes by s_p.

3.2 Explicit Unfolding (EU)

While HierSet structures are maintained as fully-folded structures, they are frequently explicitly unfolded before being queried by applications needing personalization data [23]. This practice is accomplished by storing the values that distinguish the occurrences in a separate structure and merging the personalizations into the explicitly unfolded HierSet for use by tools that require the unfolded representation. This process is shown in Figure 6, where the folded structure in Figure 6(a) is joined with the personalization values and transformed into a fully materialized unfolded structure shown in Figure 6(b). In this example, the folded structure in our example includes one adjacency (**Adj**) relationship that

is unfolded in the result to relate two pairs of occurrences.

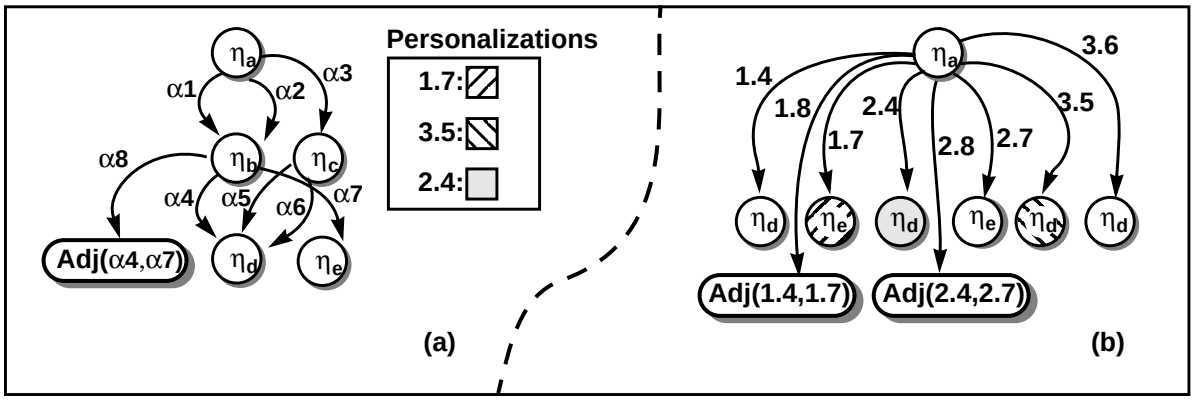


Figure 6: (a) Folded and (b) Unfolded Object with Objects, Personalizations and Relationships.

3.2.1 Algorithm and Structures

The unfold operation [14] results in a new HierSet that contains only unfolded primitives. Because queries in our performance evaluation did not require maintenance of the hierarchical structure in the EU representation, we also flattened the HierSet during the unfolding process (see Figure 2(b)).

```
// The recursive unfold() operation unfolds the HierSet rooted at root
HSet::unfold(HSet *root, Path context) {

   for(each Abstr a owned by this HSet) {
   if (a->abstractionOf->isPrimitive) {
      Primitive *p = root->instantiatePrimitive(a, context);
      p->getAttributeValues(context + a->instID);
   } else {
      a->abstractionOf->unfold(root, context+a->instID);
   }
```

Figure 7: Unfolding a HierSet.

The algorithm shown in Figure 7 proceeds as follows:

For each abstraction in the HSet:

1. Create a new unfolded primitive if the abstraction refers to a primitive. Use the context and abstraction ID as the unique name for the primitive. Allocated space in the primitive to store ***all*** of the attributes for the primitive. Read attributes from the personalization source into the primitive.
2. Extend the current context using the abstraction *ID* and pass that to the recursive call to unfold if the abstraction refers to a non-primitive HSet.

3.2.2 Performance Issues

Once unfolded and flattened, a HierSet exists as a set of occurrences (tuples containing a context and a primitive (see Section 2.1)). It occupies space proportional to the number of unfolded occurrences in the HierSet. Recall from the characteristics of the folded HierSet described in Section 3.1 that the number of occurrences is given by n, and the size of a primitive is given by s_p. Therefore the total size

(i.e. storage cost) of the unfolded design is given by ns_p. Because each unfolded primitive contains space for all possible personalized values, the size of the EU representation is independent of the number of actual personalizations.

Given a specific occurrence in the design, the time to access an attribute in the occurrence is determined by the time to access the occurrence. This is because the unfolded attribute values are stored directly with the unfolded occurrences, and so, once the occurrence is obtained, the attribute values are immediately available. However, this does not necessarily indicate that the unfolded structure is the most efficient. The unfolding cost and size of a large unfolded structure contribute substantially to the cost of performing queries on unfolded structures. Additionally, there is no clear way to access the unfolded occurrences by name unless an index on the name (formed by the concatenation of the abstraction IDs) is used. Because we have not developed indices for all of the materialization strategies, we have reserved the evaluation of indexed queries for future work.

For queries that traverse other unfolded relationships between occurrences, it is difficult to predict the paging performance for the explicitly unfolded structure. The performance is dependent upon how the data has been clustered. The clustering that we impose upon the unfolded representation is the clustering dictated by the order in which the structure is unfolded. Clustering topologically on relationships such as Adj() in Figure 6 is likely to improve performance for traversal of the relationship. However, this kind of clustering requires a second pass after unfolding, and so was not considered.

3.3 Dictionary Personalization (DP) Using a B+ Tree

In the *dictionary personalization* (DP) approach, one or more dictionaries store the values for personalized attributes. We considered several possibilities for dictionary organization. By dividing up the personalized attributes in different ways, we vary the number and sizes of the dictionaries. These variations are listed in Table 2 with descriptions of the main distinguishing characteristics. The context portion of the occurrence (i.e., the path to the occurrence in the HierSet DAG) is used as the dictionary key and stored with the personalized attribute. Initial values for the un-personalized attributes are stored in the primitive HSets. In this way, the entire HierSet structure can remain fully folded. During implicit unfolding, the query processor maintains a context path that can be used as a dictionary key. Using this key, the query processor consults the dictionary to obtain values for personalized occurrence attributes.

Table 2: Characteristics of Different Approaches to Dictionary Personalization.

Single/Multiple	Attributes Sliced	Attributes Unsliced
Dictionary for each HierSet	• One separate dictionary for each attribute. • Attribute determines dictionary D = d(a). • Context used as key in each dictionary.	• Single dictionary for all attributes. • No search required to locate dictionary. • Context/Attribute used as dictionary key. • Least overhead; only single dictionary.
Dictionary for each Primitive of each HierSet	• Each primitive has a set of dictionaries. • Attribute/Primitive determines dictionary. • Context used as key in each dictionary. • Most number of dictionaries required.	• Each primitive has a single dictionary. • Primitive determines dictionary. • Context/Attribute used as dictionary key.

3.3.1 Algorithm and Structures

We chose a B+ tree to serve as the dictionary. To lookup a personalized value for the attribute δ where *Name*(δ) = *s* of the occurrence < *[x, y, z]*, *p* > in the HierSet *h*, we do the following:

1. Determine the dictionary $D = b(h, p, s)$ to search for the personalized value of attribute *s*. This lookup function is maintained by the HierSet and has little cost, since the number of distinct attribute names is likely to be small. For this reason we do not consider I/O cost to determine the dictionary to use.
2. Search for the key *x.y.z* in D. If the key is found, return the value stored with the key. Done.
3. Otherwise, return the attribute value stored with the primitive element *p* at the end of the path *x.y.z*..

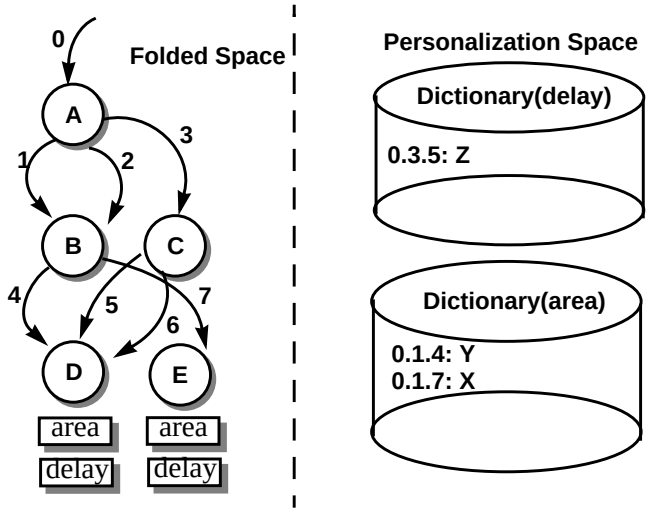


Figure 8: A Folded Description with Sliced Personalizations.

After some careful consideration we confined our performance evaluation to the DP approach that uses a single dictionary for each attribute represented in the HierSet. This approach is illustrated in Figure 8, with dictionaries for personalizing the attributes *area* and *delay.* The slicing of attributes into separate dictionaries has the desirable effect of limiting dictionary activity to only those attributes that are

relevant to a particular query. It also permits the most flexible support for different types of attributes.

3.3.2 Performance Issues

Because the personalization values are stored in a B+ tree structure indexed by the path in the HierSet DAG, a dictionary lookup is required for every access to a personalized attribute. The branching factor *b* can be expressed as:

$$b = \frac{D_{sz}}{s_{ptr} + s_{id}h + s_{attr}}. \quad (7)$$

where D_{sz} is the size of a disk page, s_{id} the size of an identifier, h the height of the HierSet DAG, s_{ptr} the size of a pointer, and s_{attr} the size of an attribute. For our simulations, $b = 100$.

The access time to an attribute can be expressed in terms of the bounds on the B+ tree performance. The number of pages accessed when looking up a personalization value for the attribute *a* with v_a total personalizations is given by:

$$cost = log_b(v_a). \quad (8)$$

3.3.3 Optimizing DP Access

One of the drawbacks of any dictionary personalization approach is that the lookup cost for an attribute value is dependent upon the number of personalizations – whether or not a given occurrence has any personalized attributes. Under the DP approach explained thus far, the query manager is always obligated to consult the personalization structure to determine if an occurrence has a personalized value for a given attribute. Here we consider how to reduce these compulsory consultations.

It is common design practice for personalizations to be localized to sub-DAGs of the HierSet DAG. This occurs because a designer typically refines sub-modules of a design one at a time. By providing a means to determine whether or not a sub-DAG contains personalizations, we can improve performance when processing queries on designs that contain localized personalizations.

Two approaches to this problem that we can identify are:

- **Bit-Vector Pruning.** Information about personalizations is stored in the HierSet. This requires maintaining a bit-vector in each abstraction (1 bit per attribute) that is used to record whether or not there are personalizations in a particular sub-DAG. The query processor determines if there are personalizations for the attribute *s* in a sub-DAG identified by the path $[a_1, a_2, \ldots, a_n]$ if:

  $$(\forall a_i, a_i.bit[s] = 1), \text{ where } i = 1..n \quad (9)$$

  Personalization updates propagate the bit vector setting to the root of the HierSet DAG, or until a *bit=1* is encountered.

- **Prefix Pruning.** A query prevents excessive lookups in the B+ tree by "peeking" into the B+ tree using a prefix whenever traversing downward in the HierSet. If the prefix lookup indicates that there are no personalizations in the B+ tree for that particular sub-DAG, then all subsequent lookups while traversing the sub-DAG can be "pruned", and thus prevented.

As we describe in our results, we evaluated both schemes and found the bit vector approach to be the most efficient. The algorithm to determine a personalized value under bit vector pruning is as follows:

1. During the traversal of the HierSet DAG (i.e. moving up or down the DAG hierarchy), maintain a stack of state variables that identify if there are personalizations along the current context path. One stack is maintained for each attribute considered in the query. The stack is kept up to date by consulting the abstraction to obtain the personalization bit for each attribute, combining it (via logical *and*) with the current top of the stack and pushing the result on the stack.
2. If the top value on the stack is 0, return the attribute value stored in the primitive.
3. If the top of the stack indicates possible personalizations in the sub-DAG, consult the B+ tree for all personalization lookups within the sub-DAG.
4. If the key is not found, always return the attribute value stored with the primitives in the sub-DAG associated with the prefix.

3.3.4 Performance Improvements from Pruning

The performance improvement from prefix pruning depends upon the distribution of personalizations. For the most common pattern of personalizations, we expect prefix pruning to improve retrieval performance. Under certain conditions, prefix pruning may perform worse that a non-pruned approach, because the benefits of pruning are offset by the additional size of the bit vector storage.

If we assume an attribute a has v_a personalizations out of n total possible, we can make probabilistic bounds estimates on the effectiveness of the pruning. We express the DP bit vector access cost to retrieve r attribute values as:

$$r \cdot \left(\left(1 - \frac{v_a}{n}\right) \cdot \log_b(v_a)\right) < \text{cost} < r \cdot \log_b(v_a) \tag{10}$$

where $\left(1 - \frac{v_a}{n}\right)$ represents the fraction of lookups not prevented by pruning.

3.4 The Partial Unfolding (PU) Strategy

Our *partial unfolding* (PU) strategy represents a middle ground (in terms of materialization) between the DP and EU approaches. In partial unfolding, the *base HierSet* – the original folded structure – is maintained in the same way as the DP approach (i.e., remains folded), but personalized

attributes are stored in separate HierSet structures (PU HierSets). PU HierSets are initially isomorphic to the base HierSet, but each contains all primitive attributes with the same name. In order to personalize a PU HierSet, we *partially unfold* it to create paths to new leaf attribute values. In this way the PU HierSet acts as a dictionary for personalization values. After personalization, the PU structure diverges from the form of the base HierSet, but it still contains the same paths. For example, a base HierSet is shown in Figure 9(a) with a PU HierSet, in Figure 9(b). In Figure 9(c), we show the PU HierSet with a single personalized occurrence [3,5]. The new occurrence path is shown as a dashed line.

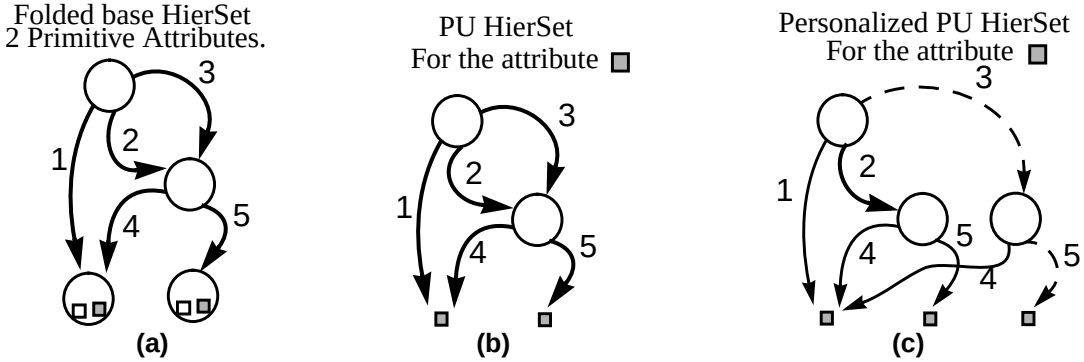


Figure 9: (a) Base HierSet and (b) PU HierSet, (c) PU HierSet with [3,5] Personalized.

The duplication of labels in the HierSet is permitted, since the uniqueness within a parent is still met, and so unique paths to each leaf are still guaranteed. The partial unfolding method has several possible advantages over unfolding the entire HierSet:

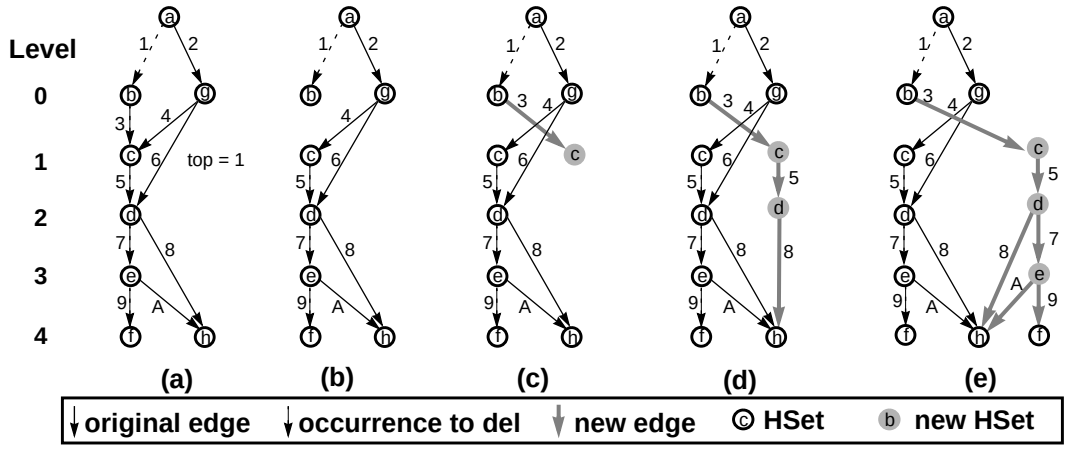
1. The PU HierSets may be smaller than an EU HierSet, and so more of them are likely to fit into the allocated page buffers of the database system.
2. Only those attributes that are personalized need a personalization structure, so some attributes in the primitive cells will still benefit from being completely folded.
3. The personalization structure exhibits similar spatial and temporal locality to its base HierSet, possibly improving paging performance.

3.4.1 Algorithms and Structures

The algorithm illustrated in Figure 10 shows the steps required to accomplish the personalization of a single occurrence in a PU HierSet. The general procedure for this process is as follows:

1. Identify the region of the DAG which must be unfolded. An HSet along the occurrence path is identified as the `top` – the closest HSet to the root with a fanin > 1.
2. For each HSet along the path between top and the leaf in the path inclusive, clone the HSet, and link it to its parent. All abstractions in the HSet are cloned except the abstraction corresponding to the portion of the path being personalized.

3. Personalize the newly cloned leaf.



a. Locate top by traversing the path from the root.
b. Remove edge incident on top HSet (from b to c).
c. Clone HSet c and change edge from b to new cloned c.
d. Clone HSet d (including its edge to h) and add edge from c to d.
e. Clone HSet e (including its edge to h) add edge from e to f.
   Update value in f with new personalization value.

Figure 10: Personalization of occurrence <[1,3,5,7,9], f> in HierSet.

The sequence of frames in Figure 10 shows the operations required for personalizing the occurrence <[1,3,5,7,9], f> in the HierSet. The figure shows the removal of all paths containing the edge 3 incident on the top HSet, followed by the systematic reconstruction of all paths that contain the edge 3, but end at a new primitive HSet for the path [1,3,5,7,9]. This reconstruction corresponds to the creation of copies (versions with small changes in terms of which Abstr they own) of all HSet between and including the *top* marker and the leaf. A detailed description of the steps is included in the figure.

3.4.2 Performance Issues

A personalization value is obtained from a PU structure by retrieving the path from the root of the PU HierSet to the appropriate leaf-level primitive, using the components of the occurrence path to guide the traversal. Because the abstractions are clustered together with each HSet, the number of pages that must be retrieved from disk is likely to be bounded above by *h* for an occurrence path of length *h*. This is because PU HSets and primitives are lightweight objects, and many fit on a disk page.

We designed the PU HierSet under the expectation that it would exhibit complimentary paging behavior to the HierSet for which it provides personalization values. It also avoids the storage of long keys such as those used in the DP approach. This is because the keys are distributed throughout the PU HierSet DAG and are also folded and re-used by many different personalization values. Because the PU HierSet keeps attributes folded as much as possible, it represents a middle ground in the materialization strategies.

4 Performance Evaluation Experiments

We performed an extensive evaluation of the three personalization approaches by varying operating conditions and data characteristics. We performed four different queries on the data, namely (1) queries that traverse the complete HierSet DAG, (2) queries that traverse sub-DAGs of the HierSet, (3) queries that traverse single folded relationships, and (4) queries that traverse paths of folded relationships. The first, an *`Aggregation`* query, computes an aggregate over the entire HierSet on a single attribute. The second, a *`Sub-Aggregate`* computes an aggregate on a sub-DAG. These are the two traversal queries considered in the DAG clustering performance evaluation described in [4]. Because HierSets are often used to describe hierarchical graphs, we added queries which traverse the HierSet along the folded relationships. Out third query, a *`Neighbors`* query traverses unfolded relationships to all neighbor primitives, and finally, a *`Paths`* query, traverses a path of unfolded relationships across the HierSet. These four queries are representative of the types of queries commonly performed on hierarchical graphs and hierarchical sets. Queries involving searching for indexed values were not included in our evaluation because we have not implemented the necessary indexing structures.

4.1 Experimental Setup

In our performance evaluation, we performed the following steps:

1. Generated a HierSet with variable height (*h*) and a constant density (*d*). We used generated data so that we could control these characteristics.
2. Personalized an attribute with a variable number of personalizations. We used two different personalization patterns to personalize attributes. In one pattern, the personalizations were randomly distributed throughout the HierSet, while in the other, the personalizations were localized to a few sub-DAGs in the HierSet (to simulate personalization as a designer might do it).
3. Clustered the design data (and personalization data, if relevant) using two different clustering algorithms. The first algorithm clustered the DAG in depth-first order. The second employed a hybrid version of depth-first and breadth-first clustering [4]. We refer to these as depth-first (DF) and children-depth-first (CDF), respectively. Because breadth-first clustering has demonstrated inferiority [4] to DF and CDF, we did not consider it in our evaluations.
4. Executed four different queries against the personalized design object. In all cases, the queries accessed personalized attributes during execution.
5. Monitored the I/O costs during the query evaluation.

4.2 Specific Characteristics

4.2.1 Height and Density

We selected a constant density of 12 and varied the height of the test HierSet DAG between 2 and 6. Table 3 shows the sizes for folded and unfolded HierSets, including the number of occurrences in the unfolded HierSet.

Table 3: Size (in 1k disk pages) of Randomly Generated DAG with Density = 12.

Height	Folded Size	UnFolded Size	Number of Occurrences
2	57	25	250
3	96	100	1000
4	148	400	4000
5	214	1600	16000
6	229	6300	63000

4.2.2 Other Parameters

In the course of our simulations, we varied a number of parameters. However, for the results presented in this paper, we fixed many of these to focus on the most relevant characteristics. For example, we fixed the size of a disk page (and buffer page) at 4096 bytes. The sizes for HierSet building blocks were fixed as shown in the table in Table 4.

Table 4: Fixed Simulation Parameters.

Symbol	Description	Value
D_{psz}	HierSet Primitive Size (10 attributes)	48 bytes
D_{hsz}, P_{hsz}	HSet Size (both HierSet and PU HierSet)	8 bytes
D_{asz}, P_{asz}	Abstraction Size (both HierSet and PU HierSet)	8 bytes
P_{psz}	PU Primitive Size (1 attribute)	8 bytes
D_{sz}	Buffer page size	4096 bytes
K_{sz}	DP key size (concatenation of 4 byte IDs)	40 bytes

5 Results

We conducted a number of focused experiments to determine the storage and I/O costs for the different personalization approaches. In the following subsections, we present the goal of each experiment, a description of the conditions of the experiment, a plot showing the results, and a brief interpretation of these results.

5.1 HierSet Sizes on Secondary Storage

5.1.1 Design Size vs. HierSet DAG Height

One clear justification for keeping HierSets folded is the possible storage savings. To measure these savings, we constructed HierSets (EU, DP, and PU) – both unpersonalized and 50% personalized on one attribute. The resulting savings are shown in the plots of Figure 11.

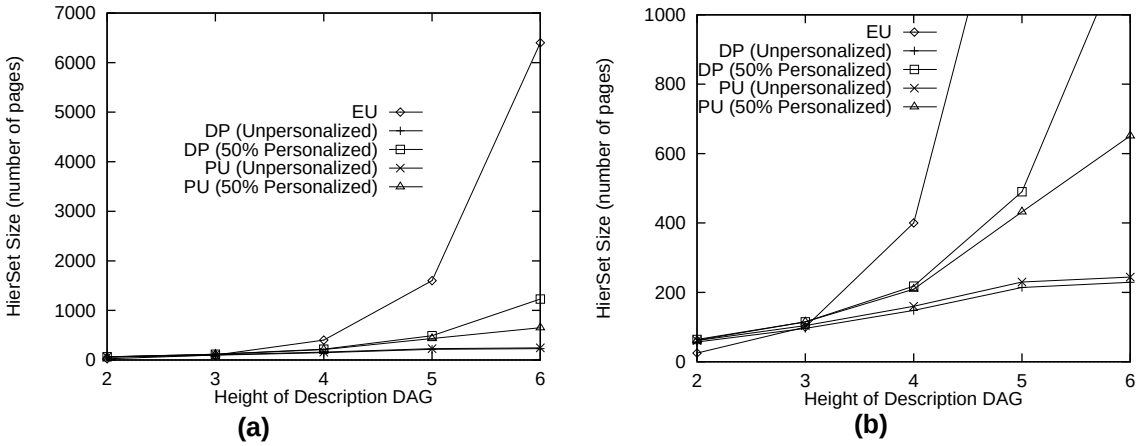


Figure 11: (a) HierSet size vs. DAG Height. (b) Magnified region of plot.

Notice that the DP size with 50% personalizations is proportional to the size of the HierSet under EU, though smaller. We also see that the PU structure shows similar size characteristics to the DP, except that the 50% personalized HierSet under PU is considerably smaller than the DP implementation for HierSets with DAG height greater than 5.

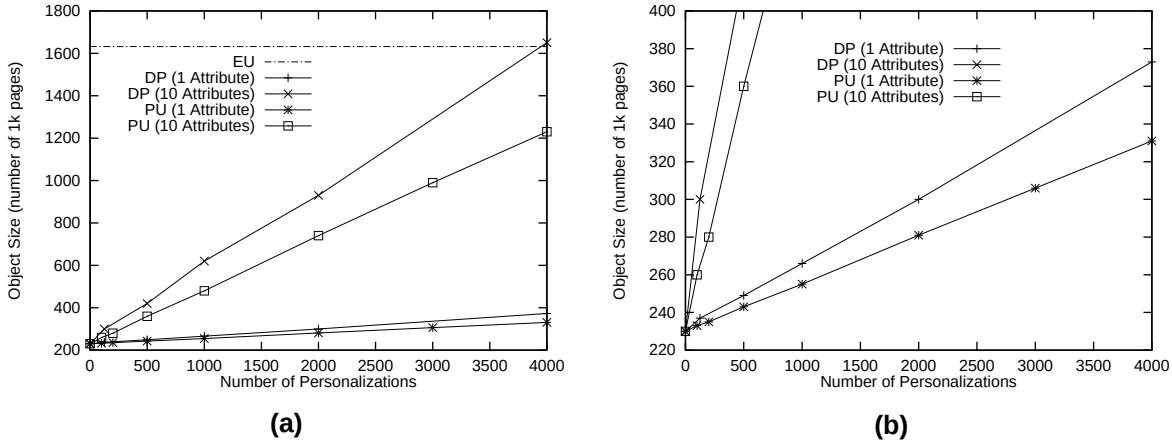


Figure 12: (a) Design Size vs. Number of Personalizations (b) Magnified.

5.1.2 Design Size vs Number of Personalizations

5.1.2.1 Personalization Schemes Compared

The second factor contributing to the HierSet size is the number of personalizations. Because both DP and PU grow with the number of personalizations, we consider these effects when evaluating the

approaches. The plot in Figure 12 shows size advantages under DP for several attributes personalized up to 25%. PU compares favorably for up to 10 attributes when compared to EU (the horizontal line), containing 10 attributes. DP grows linearly with the number of personalizations. The slope of the line is determined by the size of the keys used to index into the B+ tree. Use of key compression techniques should reduce the size of these keys, and thus reduce the slope of the curve.

5.1.2.2 The Effect of Personalization Distribution on PU Size

We observed that the PU approach is sensitive to the way personalizations are distributed throughout the occurrences in the HierSet. We compared the sizes of the PU structures under localized and randomly dispersed personalizations, as illustrated in Figure 13.

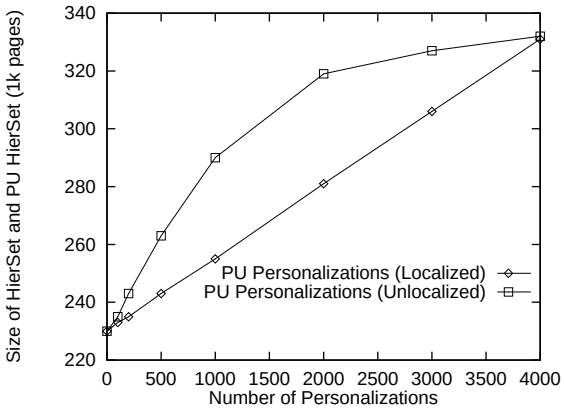


Figure 13: Size of PU for Random and Localized Personalizations.

When updates are localized to sub-DAGs in the HierSet, the resulting PU structure remains mostly folded, and as a consequence remains compact. On the other hand, if personalizations are randomly scattered throughout the occurrences in the HierSet, the resulting PU structure is larger, having been mostly unfolded. The sizes reconverge as the representations approach complete unfolding. It is common in design to refine localized portions of a design, so we expect the linear behavior to occur in practice.

5.2 I/O Performance Comparison

5.2.1 Contribution of I/O Buffer Allocation to Performance

We tested the sensitivity of our four queries to the number of allocated buffer pages by executing the query on a HierSet with 4000 personalizations. Figure 14 shows the DP and PU structures performance improving substantially as the buffer allocation approaches the representation size. In the region in which these two representation are “starved” for pages, they perform very badly. Fortunately, both the DP and PU representations are relatively small compared to the EU representation, so an adequate amount of buffer space is likely to be available.

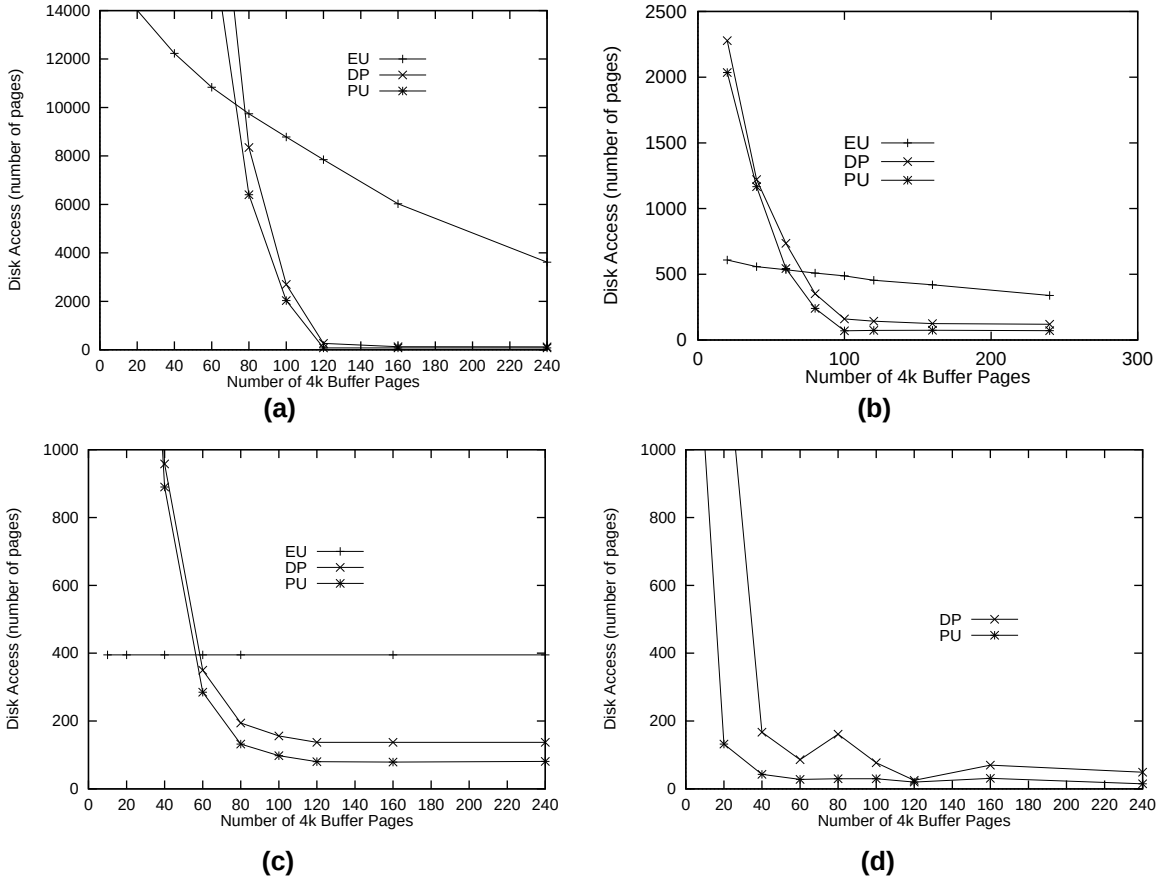


Figure 14: (a) Path (b) Neighbors (c) Aggregate (d) Sub-Aggregate queries.

5.2.2 Personalization I/O Compared

We also executed the queries while varying the number of personalizations. Figure 15 shows the relative performance of PU and DP for the Neighbor and Sub-Aggregate queries. The PU scheme outperforms the DP for the Sub-Aggregate, but there is no clear winner for the Neighbor query. Figure 15(a) demonstrates the non-uniform growth (and corresponding performance) of the PU representation.

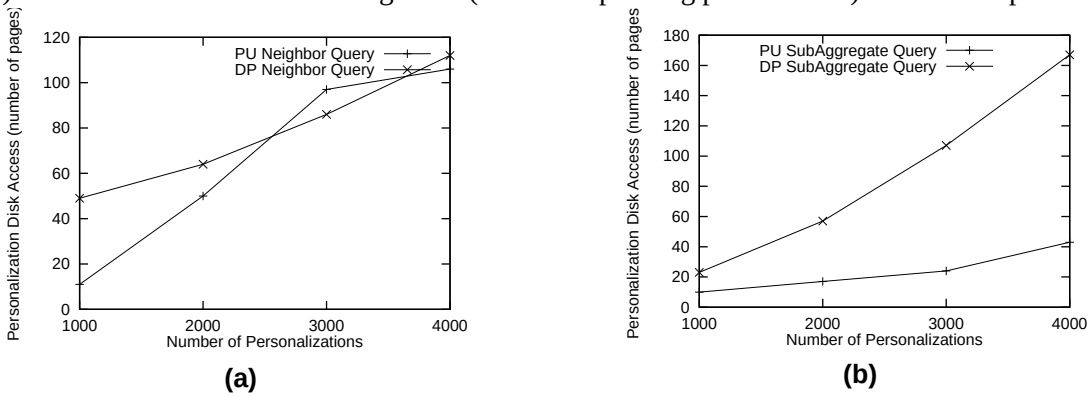


Figure 15: PU and DP Personalization I/O (a) Neighbors Query (b) Sub-Aggregate.

5.3 Optimizing DP Performance

5.3.1 Prefix Pruning and Bit Vector Pruning

As described in Section 3.3.3, the DP approach can benefit by using pruning to reduce the number of accesses to the B+ tree. We compared the performance of the bit vector and the prefix pruning for a number of queries. In general the bit vector performed better, primarily because of reduced activity in the B+ tree accesses. This slight performance edge is illustrated in Figure 16

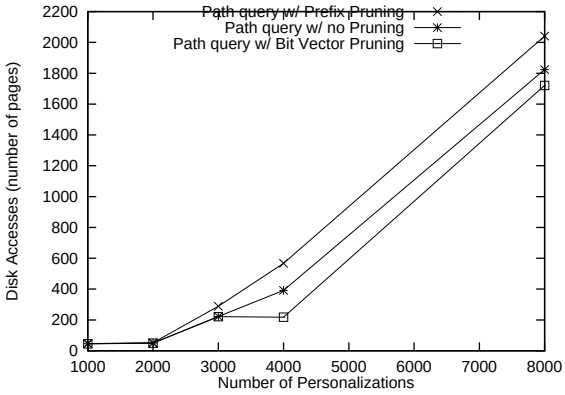


Figure 16: Comparison of Prefix and Bit Vector Pruning.

We measured the effectiveness of bit vector pruning on the various queries. We notice an improvement for the paths query, but no improvement for the aggregation query. The lack of improvement for the aggregation query is because the aggregation query corresponds to an in-order traversal of the paths in the HierSet DAG, and so parallels the in-order and very efficient traversal of the DP structure.

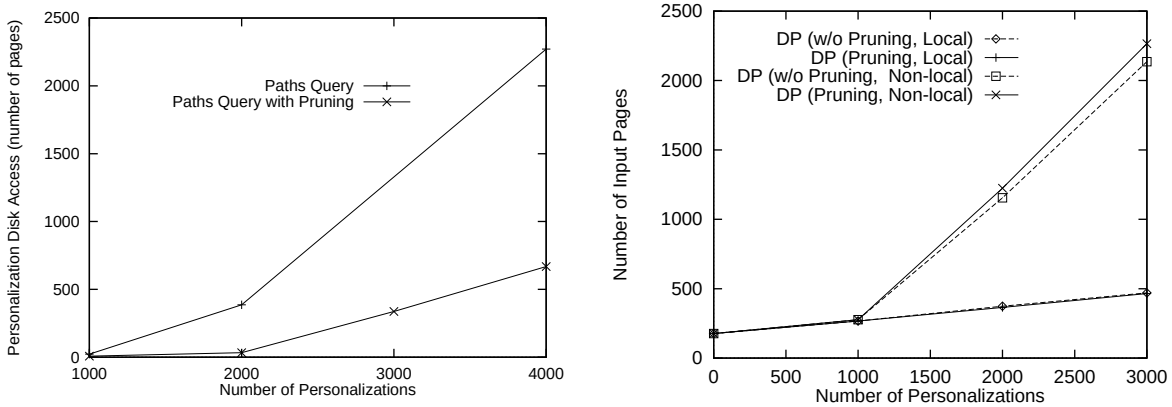


Figure 17: Effectiveness of Bit Vector Pruning for (a) Paths query and (b) Aggregation query.

5.3.2 The Effect of Personalization Update Patterns on Pruning Performance.

The pruning optimization offers some promise for improving the performance of queries that access personalized data. However, we noticed variations in the effectiveness of the pruning. We show

the variation in the plot of Figure 18. This plot shows clearly that the performance of the pruning technique is dependent on how well the personalizations are grouped into sub-DAGs of the HierSet.

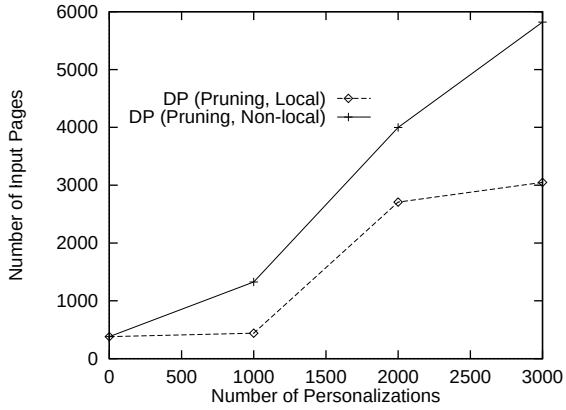


Figure 18: Paths Query with Local and Random Personalizations.

5.3.3 The Effect of Personalization Update Patterns DP Performance

We found that the update patterns had a substantial impact on the performance of the DP approach. This is due to the way the personalizations are grouped in the B+ tree when they are inserted. A more localized set of personalizations results in a more compact and more "intelligently clustered" B+ tree. Note the personalization I/O as shown Figure 19 for both the *Neighbors* and *Sub-Aggregation* queries for the different patterns of updates.

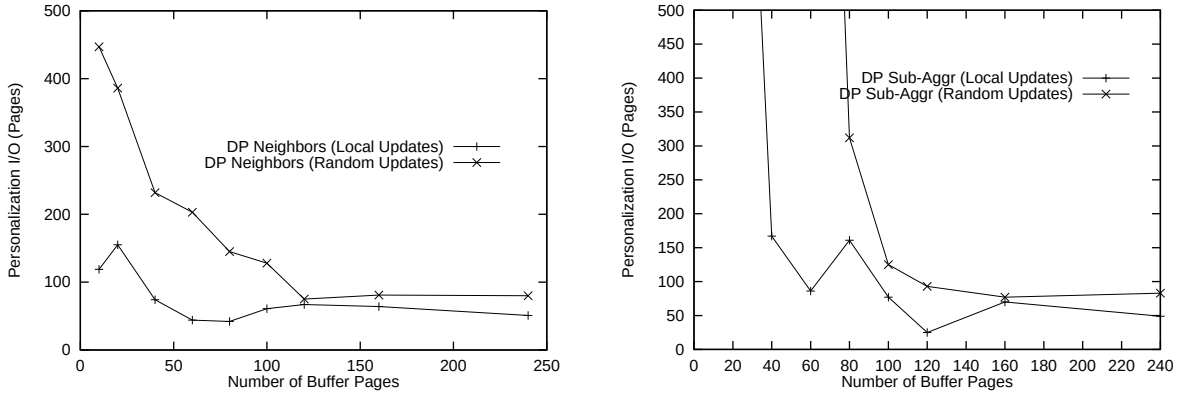


Figure 19: Sensitivity to Update Patterns for (a) Neighbors and (b) Sub-Aggregation Queries.

5.4 Optimizing the PU Approach

Because we had control over the way that the PU HierSet was clustered we were able to measure the effects of clustering on the query performance. In this experiment, we measured the performance variation of the PU approach for both the CDF and DF clusterings. The resulting plot is shown in Fig-

ure 20, indicating that the CDF clustering yields the best performance for the `Paths` query. However, the benefits of CDF clustering are diminished by randomly distributed personalizations.

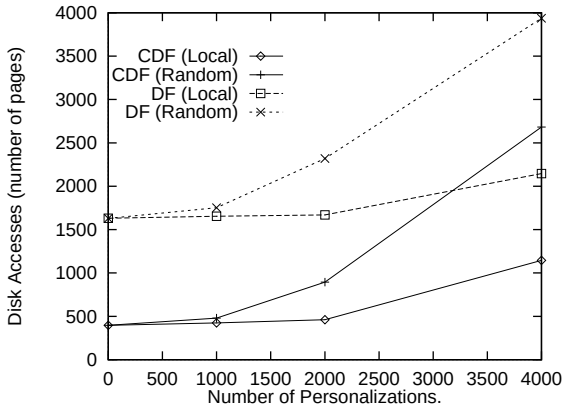


Figure 20: Paths Query Performance for CDF and DF Clustering.

6 Related Work

Relational Databases. As an alternative indirection solution, the *owns* and *owns-abstraction-of* relationships could be stored using relational tables. In this representation, the traversal of the OAO-DAG is accomplished via a join on the relationship. Traversal of the variable length paths extending from the root of the HierSet to the leaf primitive would be accomplished via a recursive join. Although a general recursive join could compute a set of primitives which are reachable from the root, the general join process does not retain the path information. Thus, casting the HierSet into a traditional relational model leaves us without the requisite path information. This path information is essential to solve the personalization problem, because the path is used to form the key into personalization indices. The path also imparts the requisite identity to the resulting occurrence, another important facet missing from a general recursive query.

Work in [10] presents extensions to SQL that provide a means to specify transitive closure queries while preserving the path information in a special PATH table. The PATH table can be queried and its contents can be accessed during the course of the query. Furthermore, this information could potentially be used to formulate keys to be used to access personalization data, though it has not been explored in [10]. All of these operations would have to be done explicitly, and all of the necessary provisions would also be required to provide access to unpersonalized values as well – as already provided for in the solution we present in this work. One key strength of our model is that captures the intended semantics of a commonly used structure and uses a highly optimized method to access both personalized and unpersonalized data stored in the structure. Additionally it does so via a query mechanism that can fully exploit the folded nature of the data to achieve superior performance.

Database Views. Earlier work in OODB database views has established data models, object algebras, update policies and materialization strategies, all motivated by the desire to provide either real or virtual restructuring of data for database applications [1,27]. Much of the formalization of complex data types can be found in [7].

Determining how to update materialized views is discussed in [5] in the context of relational views. View maintenance and materialization for a nested data model is presented in [17], along with log structures to increase the efficiency of updates. In both [5] and [17], the views are limited to select, project, and join (SPJ) views. View materialization is also implemented in the Multiview system [19, 20, 21]. However, the MultiView OODB view system [26] employs an object-preserving algebra as a query language for view definitions. Our `Unfold` operation is unique in that it is a complex view transformation requiring special support for both definition and materialization.

Recent work on view materialization has focused on applications to special view definitions and specific domains. View materialization for a deductive database used in temporal authorizations is presented in [11], while [12] examines views and view materializations appropriate for building graphical user interfaces. In our HierSet model, we address the complex restructuring employed in the design domain, such as flattening hierarchical graphs and deriving transitive relationships [15]. Furthermore, a view defined using the `unfold` operation is updatable via our solution to the personalization problem.

Electronic CAD. Work on the HS system [24] describes an API capable of implicitly flattening netlist data. Updates to the implicitly flattened data are limited, and require a re-initialization of the database. We support *in-context* updates (to implicitly defined occurrences) via our solution to the personalization problem. We also provide *out-of-context* updates (to all occurrences in a sub-DAG) via direct manipulation of the HierSet DAG. The HS system does not address the personalization problem.

Research on hierarchical attribute grammars [14] presented incremental update schemes to propagate changes from a folded representation to an explicitly unfolded representation. Our work supports such propagation without requiring the maintenance of an explicitly materialized unfolded representation. The FICOM system [3] maintains complex constraints across various abstraction domains, but also requires that the two distinct representations are fully materialized. FICOM addresses update propagation in both directions, but the same problems of space and performance overhead remain.

Recent research in enabling technology for electronic design frameworks has focused on information modeling of folded and unfolded design [6,8,28]. These models are used to define APIs, to develop data structure generators, or to formalize the exchange of data between systems. However, there is no published work relating how the information models can be used as the basis for sophisticated view definitions within an object-oriented database. With our HierSet model and personalizations, we pro-

vide principled and optimizable support for modeling the relationships mandated by these information models.

7 Conclusions

7.1 General Contributions

We have made the following contributions in this paper:

- We have identified and formalized the personalization problem for an unfolded view of folded hierarchical structures which commonly occur in many application domains.
- We proposed two approaches to solving the problem, our partial unfolding approach (PU), which tries to keep personalization values folded as long as possible, and the dictionary personalization approach (DP), which adapts existing index techniques from the literature to solve the problem.
- We implemented these two personalization strategies and a fully-materialized explicit unfolding (EU) strategy, as well as pruning and clustering techniques in a uniform test-bed implementation in order to provide a fair performance comparison of the approaches.
- Within the test bed we ran extensive experimental tests varying parameters such as data characteristics, database buffer sizes and percentage of personalization. A summary of results is given below.

7.2 Performance Evaluation Results

Our results indicate that either the PU or the DP approach are superior to the EU approach for full to partial personalizations of up to ten attributes. This means that either the PU or DP approach are preferred for use early in the refinement process of the data. The performance gap between EU and PU/DP is even more substantial when we consider that many queries may access unpersonalized attributes, and so would perform well even if other attributes are fully personalized.

Our results show the PU approach to be slightly better than the DP approach. Additionally, the PU approach offers substantially better performance when the personalizations are localized to a sub-DAG in the HierSet which is of practical significance since it is common in design practice to refine localized portions of a design. We improved the performance of the DP approach by using bit vector pruning to avoid unnecessary lookups during query processing. While, we found the effectiveness of the pruning was diminished when personalizations were randomly distributed, this is not a typical scenario for applications that use HierSets.

8 Future Work

Based upon our experimental results, we believe that the size of the keys used in the DP approach

limits the effectiveness of the DP. In the future, this could be addressed by using key compression techniques or perhaps even prefix B-trees to reduce the DP disk footprint and improve I/O performance.

In this work, we have evaluated the query retrieval costs for the personalization structures. A future study could include update/query workloads.

We are in the process of developing rewrite rules for other kinds of queries on the HierSet structure. These rewrite rules may present opportunities for optimization which benefit some personalization approaches more than others. We plan to consider these factors as we develop approaches to query optimization within the HierSet model.

9 Acknowledgments

The authors are grateful to Pedro Marron for the implementation of the partial unfolding algorithm. We would also like to thank the UM database group for reviewing earlier drafts of this work.

10References

[1] S. Abiteboul and A. Bonner, “Objects and Views,” in *Proc. of the ACM SIGMOD 91*, 1991.

[2] B. Amann and M. Scholl, “Gram: A Graph Data Model and Query Language”, *ECHT 92*.

[3] R. Armstrong and J. Allen, “FICOM: A Framework for Incremental Consistency Maintenance in Multi-Representation, Structural VLSI Databases,” in *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, 1992, pp. 336-343.

[4] J. Banerjee, W. Kim, S.-J. Kim and J. F. Garza, “Clustering a DAG for CAD Databases”, *IEEE Transactions on Software Engineering*, 14(11):1684-99, 1988.

[5] J. Blakeley, “Efficiently Updating Materialized Views”, SIGMOD Record, 15(2):61-71, June, 1986.

[6] A. Bredenfeld, “A Generator for Graph-Based Design Representations,” in *4th International Working Conference on Electronic Design Automation Frameworks*, (EDAF 94) 1994.

[7] P. Buneman, S. Naqvi, V. Tannen and L. Wong, “Principles of Programming with Complex Objects and Collection Types”, *Theoretical Computer Science*, 149(1995):3-48.

[8] CAD Framework Initiative, Inc., “Design Representation Electrical Connectivity Information Model and Programming Interface,” 121, October 23, 1991.

[9] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, “From Structured Documents to Novel Query Facilities,” in *Proc. ACM SIGMOD International Conference on Management of Data*, 1994.

[10] S. Dar and R. Agrawal, “Extending SQL with Generalized Transistive Closure,” in *IEEE Transactions on Knowledge and Data Engineering*, 5(5):799-812, 1993.

[11] E. Ferrari, E. Bertino, C. Bettini, A. Motta and P. Samarati, “On Using Materialization Strategies for a Temporal Authorization Model”, in *Proc. Workshop on Materialized Views: Techniques and Applications*, Montreal, CA, June 7 1996.

[12] N. Goyal, C. Hoch, R. Krishnamurthy, B. Meckler, M. Suckow, “(Active) View Materialization in GUI Programming,” in *Proc. Workshop on Materialized Views: Techniques and Applications*, Montreal, CA, June 7 1996, pp. 56-64.

[13] S. Heiler, “An Object-Oriented Approach to Data Management: Why Design Databases Need It”, in *Proc. IEEE/ACM Design Automation Conference (DAC)*, 1987, pp 335-40.

[14] L. G. Jones, “Fast Batch and Incremental Netlist Compilation of Hierarchical Schematics,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(7):922-31, 1991.

[15] M. Jones and E. A. Rundensteiner, “Extending View Technology for Complex Integration Tasks,” in *Proc. 4th Intl. Working Conf. on Electronic Design Automation Frameworks, (EDAF 94)*, pp. 71-80, 1994.

[16] M. Jones and E. A. Rundensteiner, “An Object Model and Algebra for the Implicit Unfolding of Hierarchical Structures”, Electrical Engineering and Computer Science Dept., University of Michigan, Ann Arbor, Tech. Rep. CSE-TR-251-95, July 1995.

[17] A. Kawaguchi, D. Lieuwen, I. Mumick and K. Ross, “View Maintenance in Nested Data Models,” in *Proc.*

Workshop on Materialized Views: Techniques and Applications, Montreal, CA, June 7 1996.

[18] Konomi, T. Furukawa, and Y. Kambayashi, "Super-Key Classes for Updating Materialized Derived Classes in Object Bases," in *Proc. DOOD Conference*, Dec. 1993.

[19] H. A. Kuno and E. A. Rundensteiner, "Materialized Object-Oriented Views in MultiView," in *Proc. Fifth Intl Workshop on Research Issues on Data Engineering: Distributed Object Management (RIDE-DOM '95)*, March 1995.

[20] H. A. Kuno and E. A. Rundensteiner, "Using Object-Oriented Principles to Optimize Update Propagation to Materialized Views", in *Proc. IEEE International Conference on Data Engineering (ICDE)*, 1996, pp. 310-317.

[21] H. A. Kuno and E. A. Rundensteiner, "The Multiview OODB View System: Design and Implementation", *Journal of Theory and Practice of Object Systems (TAPOS), Special Issue on Subjectivity in Object-Oriented Systems*, John Wiley, New York, 1996.

[22] D. Maier, "Making Database Systems Fast Enough for CAD Applications", in *Object-Oriented Concepts in Databases and Applications*, W. Kin and T. H. Luchovsky eds., ACM Press, 1989.

[23] Mentor Graphics, Inc. , "Design Viewpoint Editor", On-line Documentation, System 8.0, 1995.

[24] N. Parikh, C.-Y. Lo, N. Singhal, and K. Wu, "HS: A Hierarchical Search Package for CAD Data," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):1-5, 1993.

[25] E. A. Rundensteiner, "Design Tool Integration Using Object-Oriented Database Views," in *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, 1993, pp. 104-107.

[26] E. A. Rundensteiner, "MultiView: A Methodology for Multiple Views in OODBs," *in Proc. IEEE Intl. Conf. on Very Large Data Bases (VLDB)*, 1992, pp. 187-198.

[27] M. H. Scholl, C. Laasch, and M. Tresch, "Updatable Views in Object-Oriented Databases," in *Proc. DOOD Conference*, Germany, Dec. 1991.

[28] G. Scholz and W. Wilkes, "Information Modeling of Folded and Unfolded Design", in *Proc. European Design Automation Conference (EDAC)*, 1992.