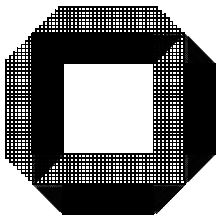


UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK

Postfach 6980, D-7500 Karlsruhe 1

Clustering in Object Bases

Carsten Gerlhof[*] *Alfons Kemper*[*] *Christoph Kilger*[†] *Guido Moerkotte*[†]

[*]RWTH Aachen
Lehrstuhl für Informatik III
5100 Aachen, Germany
[*gerlhof*|*kemper*]@informatik.rwth-aachen.de

[†]Universität Karlsruhe
Fakultät für Informatik
7500 Karlsruhe, Germany
[*kilger*|*moer*]@ira.uka.de

Interner Bericht Nr. 06/92 * Juni 1992

Abstract

We investigate clustering techniques that are specifically tailored for object-oriented database systems. Unlike traditional database systems object-oriented data models incorporate the application behavior in the form of type-associated operations. This source of information is exploited for clustering decisions by *statically* determining the operations' access behavior applying dataflow analysis techniques. This process yields a set of weighted access paths. The statically extracted (syntactical) access patterns are then matched with the actual object net. Thereby the inter-object reference chains that are likely being traversed in the database applications accumulate correspondingly high weights. The object net can then be viewed as a weighted graph whose nodes correspond to objects and whose edges are weighted inter-object references. We then employ a newly developed (greedy) heuristics for graph partitioning—which exhibits moderate complexity and, thus, is applicable to object bases of realistic size. Extensive benchmarking indicates that our clustering approach consisting of static operation analysis followed by greedy graph partitioning is in many cases superior to traditional clustering techniques—most of which are based on dynamically monitoring the overall access behavior of database applications.

1 Introduction

The fate of object-oriented database systems will largely depend on their performance compared to currently employed database technology, e.g., relational database systems. To increase the performance, clustering has been considered for all data models since the early days of databases, e.g., an optimal clustering strategy has been developed for the hierarchical model [Sch77]. Contrary to other data models, it is one of the main characteristics of object-oriented databases that the object nets tend to be recursive. Consequently, several clustering strategies have been developed which can also deal with recursive object nets, e.g. [Sta84, CDRS86, KCB88, HK89, BD90, TN91, CH91]. Except for [KCB88, CDRS86, BD90], all these approaches rely on statistical data *dynamically* collected from former application runs, i.e., monitoring. Thus, they are applicable to object-oriented databases but they are not necessarily specifically tailored for those since they ignore the object types' associated behavior as a source of information for clustering decisions.

In this paper, we propose to exploit the knowledge about the objects' behavior—which is an integral part of the object base schema—for clustering decisions. For this purpose, we developed a technique, called *decapsulation*, that extracts the access patterns from the operations' implementation. Thus, decapsulation is a *static* approach to determining the application characteristics. The outcome is a set of weighted path expressions which model the potentially traversed reference chains at the syntactical level. The weights may either constitute the *traversal frequency*—if the goal is to optimize overall throughput—or the *criticality* of the operation—if one wants to give priority to certain particularly important operations. The static analysis of operations' access behavior exhibits major advantages over monitoring the overall database usage characteristics:

- decapsulation does not induce any run-time penalty which has to be tolerated when monitoring the database applications;
- monitoring generally requires an extensive (non-optimized) "training" phase during which the statistics have to become consolidated;
- static operation analysis enables us to emphasize particularly critical operations in the clustering decisions which is not possible when we monitor the anonymous, overall database applications.

Following decapsulation, the statically derived weighted path expressions are matched with the actual object net. Thereby the (inter-object) reference chains that are likely to be traversed in a database application accumulate accordingly high weights. This results in a net whose nodes correspond to objects and whose edges are weighted inter-object references.

Deriving the access patterns of database applications—statically or dynamically—is, however, only one dimension of the clustering problem. The other dimension is the mapping of the objects to pages under the consideration of the derived access characteristics. In this dimension, we distinguish between *sequence based* and *partitioning based* approaches. The former subsumes a wide variety of algorithms that sequentialize the object net into a total order and map the resulting object sequence to pages in a straight-forward manner. It was shown before by Tsangaris and Naughton [TN91] that the partitioning based approach, where subgraphs that exhibit a high degree of inter-object connectivity

are grouped, is usually superior to sequence based mappings. Unfortunately, the hitherto employed heuristics for partitioning the object net based on Kernighan and Lin's graph partitioning heuristics [KL70] is prohibitively expensive (in run time) for object bases with realistically large numbers of objects. We, therefore, developed a greedy partitioning heuristics which has a moderate run time complexity while yielding an even better clustering quality—according to our extensive benchmarks. Further, if we consider some operations to be critical, the heuristics combined with decapsulation is often superior to other approaches.

The rest of the paper is organized as follows. In Section 2 we first introduce the classification of clustering strategies and then develop our new greedy graph partitioning algorithm for mapping objects to pages. Section 3 introduces the decapsulation technique for statically extracting information on the access behavior of an operation from its implementation. Section 4 describes our simulation environment that we used for benchmarking a large variety of clustering strategies. In Section 5 we present the benchmark results, and Section 6 concludes the paper. Appendix A describes the decapsulation technique in more detail, which could only be overviewed in Section 3.

2 Sequence Based and Partitioning Based Clustering

Most of the clustering algorithms found in the literature transform the object net into a *clustering sequence* which is then stored on pages such that all objects on one page form a subsequence of the clustering sequence. Section 2.1 describes a generic clustering algorithm which subsumes all known sequence based clustering algorithms.

The main drawback of sequence based clustering algorithms is that objects being closely related may be separated by some page border even if they are highly related neighbors in the clustering sequence. An alternative—and more promising—approach to clustering is to apply graph partitioning algorithms to the object graph. Thereby, the object graph is partitioned, such that for each subgraph all objects can be stored on one page. Unfortunately, optimal algorithms for the partitioning problem require exponential run time. Therefore, heuristics must be applied. Section 2.2 presents our new graph partitioning heuristics which exhibits very good performance, i.e., quality of the partition, while having a moderate run time complexity. Section 2.3 classifies related work.

2.1 A Generic Sequence Based Cluster Algorithm

The generic algorithm is denoted in the style of a UNIX pipe consisting of the two steps *PreSort* and *Traversal*:

PreSort | *Traversal*

In the first step, all objects to be clustered are sorted applying a method *PreSort*, resulting in the input sequence for the second step. During the second step the object net is traversed using the method *Traversal*. *Traversal* is initialized by the first non-visited object of the input sequence and then traverses all objects reachable from this one—this is repeated until all objects have been visited. The objects are stored on pages in the order in which they are visited, resulting in a sequence of pages. As we will show in Section 2.3, this algorithm subsumes all known sequence based clustering strategies.

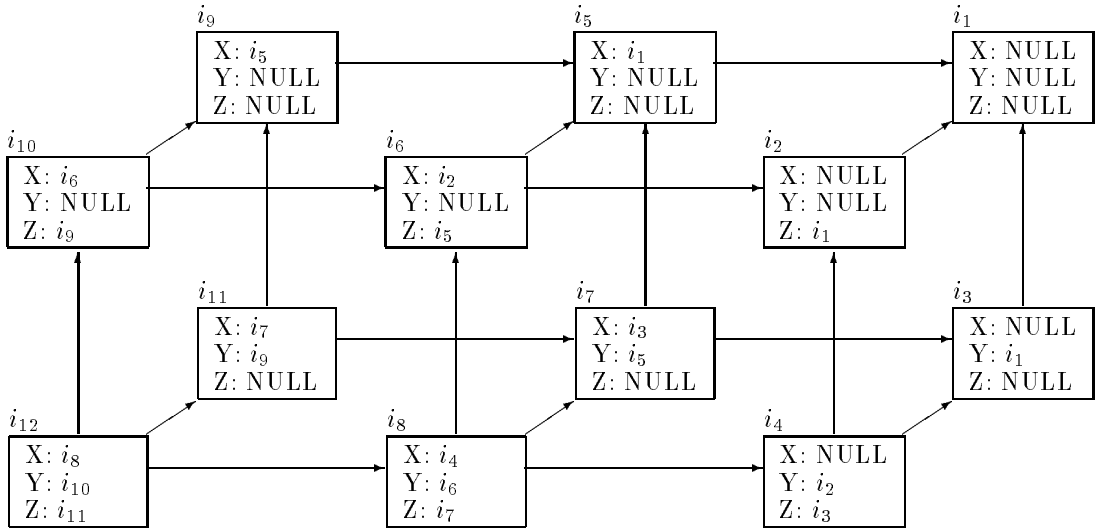


Figure 1: A simple database extension

To obtain a good clustering sequence the *Traversal* component should exhibit a similar access behavior as the applications. Thus, the importance of the *Traversal* method is obvious. Since the impact of the *PreSort* order may be less obvious, it is highlighted by means of an example. Consider the database extension shown in Fig. 1. It consists of a number of grid points with X, Y and Z coordinates constituting a regular grid. The goal is to cluster these objects for the following sequence of operation invocations:

i_9.xmove(3);	i_{10}.xmove(3);	i_{11}.xmove(3);	i_{12}.xmove(3);
(i_9, i_5, i_1)	(i_{10}, i_6, i_2)	(i_{11}, i_7, i_3)	(i_{12}, i_8, i_4)

where $xmove(n)$ follows a trajectory of length n in X direction. The access trace for each invocation is written underneath it.
One traversal strategy proposed for clustering is depth-first traversal (see, e.g., [BKKG88]). It is applied to our example where the entry points to the objects are i_1, i_2, i_3, ..., i_{12} (in this order). Let us assume that this is the order in which the objects have been created, i.e., the order is by the objects' time of creation (abbreviated by *toc*). Instantiating the generic clustering algorithm this strategy can be denoted by *toc* | *depth-first*.

If exactly four grid objects fit on one page, *toc* | *depth-first* clusters as follows:

$$[i_1, i_2, i_3, i_4] \quad [i_5, i_6, i_7, i_8] \quad [i_9, i_{10}, i_{11}, i_{12}].$$

Assuming a page buffer of size 1[1], executing the above application leads to 12 page faults which happens to be the worst case.

For comparison, let us assume another presort order. The objects of the grid are now sorted by increasing static reference counts[2] (abbreviated *src*). Then, the traversal starts at the root object i_{12} (i_{12} has a static reference count of 0). The clustering algorithm *src* | *depth-first* yields the clustering sequence

$$[i_{12}, i_8, i_4, i_2] \quad [i_1, i_3, i_6, i_5] \quad [i_7, i_{10}, i_9, i_{11}]$$

leading to only 8 page faults for the above application. Thus, besides the traversal strategy whose importance has been emphasized in the literature, the presort order has a major impact on the result of a clustering strategy.

[1] This unrealistically low number is chosen to compensate for the extremely low database volume.

[2] The static reference count of an object o is given by the number of times o is referenced in the object base [Sta84].

```
PageList := ⟨ ⟩;
Store each object on a new page and insert this page into PageList;
Let ConnectList be a list containing all tuples of the form (o1, o2, w_{o1,o2})
    where w_{o1,o2} is the total weight of all references from o1 to o2 or vice versa;
Sort ConnectList by descending weights;
foreach (o1, o2, w_{o1,o2}) in ConnectList do begin
  Let P1, P2 be the pages containing objects o1, o2;
  if P1 ≠ P2 and all objects of P1 and P2 fit on one page then begin
    Move all objects from P2 to P1;
    Remove P2 from PageList;
  end if;
end foreach;
```

Figure 2: The greedy graph partitioning based clustering algorithm

2.2 Clustering Based on Greedy Graph Partitioning

Partitioning based clustering algorithms partition the object net into a set of subgraphs such that for each subgraph all objects fit on one page. The problem is then to minimize the sum of the weights of all references crossing the page borders (which is known to be NP-complete).

In the literature graph partitioning was considered for clustering only in the work of Tsangaris and Naughton [TN91], where the algorithm by Kernighan and Lin [KL70] was applied. But this algorithm appears to be prohibitively expensive because of its asymptotic run time complexity of $O(n^{2.4})$, where n is the number of objects stored in the object base. In this section, we present a new partitioning based clustering algorithm—based on greedy graph partitioning—with a rather moderate asymptotic run time of $e \log e$, where e is the number of references stored in the object base. For realistic object bases $e \log e$ will be significantly lower than $n^{2.4}$ as the number of references emanating from an object is not nearly as high as n. For example, if we consider an object base with 100.000 objects, and assume that—in the average—100 references emanate from each object, $n^{2.4} = 10^{12}$, whereas $e \log e \approx 2.3 * 10^8$.

Partitioning based clustering is strongly related to subset optimization problems (e.g., the minimum spanning tree problem [PS82]) for which greedy algorithms often find good solutions. Our greedy clustering algorithm shown in Fig. 2 performs as follows: First, all objects are stored on a new page; these pages are inserted into a *PageList*. Then, for all objects o_1, o_2 connected by some reference in the object base, the connectivity w_{o_1,o_2} of o_1 and o_2 is computed and a tuple (o_1, o_2, w_{o_1,o_2}) is inserted into the list *ConnectList*. The *connectivity* of two objects o_1 and o_2 is given as the total weight[3] of all references stored in the object base that lead from o_1 to o_2 and, vice versa, from o_2 to o_1.

All tuples of *ConnectList* are visited in the order of descending connectivity values. Let (o_1, o_2, w_{o_1,o_2}) be the current tuple and let P_1, P_2 be the pages on which the objects o_1 and o_2 are stored. If $P_1 \neq P_2$ and if all objects stored on P_1 and P_2 fit on one page the two pages are joined.

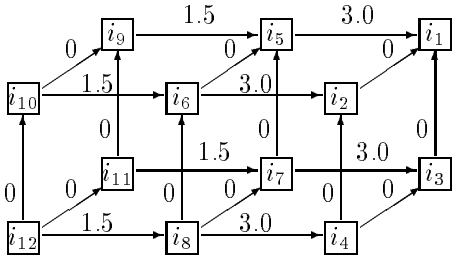
The run time complexity of the algorithm is $O(e \log e)$ because of sorting the list *ConnectList* and—assuming logical object identifiers—object lookup in pages.

[3]The weights being determined either by dynamic monitoring or by our decapsulation based approach.

Example: Suppose for the references of the database extension of Fig. 1 the weights shown in the illustration below (we will show in Section 3 how these weights could be determined by static operation analysis). For these weights, the greedy graph partitioning algorithm computes the clustering scheme[4]

$$[i_{12}, i_8, i_4] \quad [i_{11}, i_7, i_3,] \quad [i_{10}, i_6, i_2] \quad [i_9, i_5, i_1]$$

This clustering scheme leads to 4 page faults and is optimal for the application described at the beginning of Section 2.1.



◇

2.3 Related Work

Aside from the above cited work by Tsangaris and Naughton [TN91], all other related work is founded on sequence based clustering algorithms. In this section, these algorithms are classified according to the generic clustering algorithm. The related work is summarized in Table 1. The traversal component of the algorithms in Table 1 is denoted as *TraversalAlgorithm.OrderingOfSuccessors* denoting the order in which the successors of some object are visited. If *OrderingOfSuccessors* is omitted an arbitrary ordering is assumed. Note that the sequence based clustering algorithms investigated in the literature cover only a small part of all possible algorithms derivable from our generic algorithm. Subsequently, we describe pre-sort orders and traversal strategies separately.

Pre-Sort Orders. In [BKKG88] *user* hints are exploited to select the starting points of the traversal—leading to a partial order on the objects in which selected objects are sorted before non-selected. A whole class of orders found in the literature is based on an order of types. All objects of one type are arranged before all objects of another type if the former precedes the latter in the given type order. This results in a partial order on objects only—its completion to a full order is arbitrary. Type based pre-sort orders have been examined by [Sta84, HZ87, BD90]. In [Sta84] the pre-sort order *trace* has been proposed to obtain a near optimal clustering sequence. *Trace* sorts objects according to their first appearance in the access trace of a specific application. This requires monitoring the trace of the application. The clustering strategy incorporated in the Cactis prototype [HK89] sorts objects by descending dynamic reference counts (abbreviated *drc*) before starting the traversal. The *drc* value of an object is given by the number of times the object is referenced in the trace of a former application. In [CH91] the Kruskal algorithm [Kru56] is utilized to compute the maximum weight spanning tree of the object net. This spanning

[4]The algorithm starts with some triple whose weight is maximal, say $(i_8, i_4, 3.0)$. Thus, i_8 and i_4 are placed on the same page by merging the pages they were initially assigned. Eventually, the triple $(i_{12}, i_8, 1.5)$ is considered—before any triple with weight 0—and i_{12} is placed on the same page as i_8 and i_4.

Sequence based clustering algorithms:

[Sta84]	λ | depth-first	λ | breadth-first	λ | id
	λ | depth-first.src	λ | breadth-first.src	trace | id
	λ | depth-first.drc	λ | breadth-first.drc	type | id
[HZ87]	type | id		
[BKKG88]	user | depth-first	user | breadth-first	
	user | children-depth-first		
[HK89]	drc | best-first.dtc		
[BD90]	type | placement-trees		
[CH91]	kruskal.dtc | id		

Partitioning based clustering algorithms:

[TN91]	Kernighan & Lin graph partitioning [KL70]
This paper	greedy graph partitioning

λ	Arbitrary input sequence
src	Stat. ref. counts (increasing)
drc	Dyn. ref. counts (decreasing)
dtc	Dyn. transition counts (decr.)

Table 1: Related work

tree is then transformed into an object sequence which is stored on consecutive pages. If an arbitrary pre-sort order is applied, we denote this by λ.

Traversal Strategies. For the traversal algorithm many different alternatives like depth-first, breadth-first, children-depth-first, best-first, etc. can be found in the literature. In the Cactis prototype [HK89] a best-first traversal is utilized: all references from an object on the current page to a not yet stored object are ordered according to their traversal frequencies, and the object referenced by the highest ranked reference is added to the current page. We denote this traversal strategy by *best-first.dtc* (*dtc* abbreviates *dynamic transition count*).

Beside *dtc*'s, reference counts for objects can be used to guide the traversal. Here, the successors of objects are sorted by increasing *src* or decreasing *drc* values [Sta84].

In [BD90] *placement trees* were introduced to describe the access patterns of applications. The nodes of placement trees represent types and the edges represent attributes. The placement trees are then matched onto the object net and each matching set of objects is stored on a single page.

The simplest traversal strategy is to copy the input sequence to the clustering sequence (denoted by *id*). This strategy has been applied by [Sta84, HZ87, CH91].

3 Clustering by Static Method Analysis

Obviously, all clustering algorithms need information about the access patterns of applications that are to be supported by clustering. In this section, we focus on techniques to determine these access patterns in terms of *reference counts* for objects and *transition counts* for inter-object references, accumulating the access patterns of several applications. Reference and transition counts can either be determined by monitoring real applications or by analyzing the applications' schema and the object instances of the object base. We

refer to the former as *dynamically* and to the latter as *statically* derived reference and transition counts, respectively.

Dynamically derived reference and transition counts exhibit three disadvantages. First, monitoring real applications on the real object net is required which imposes a run-time penalty on the applications during the sampling phase. Second, dynamically derived reference and transition counts reflect only the average load during the sampling phase. Especially, they cannot account for the *criticality* of operations which may vastly differ from the relative invocation frequencies, e.g., in the case of emergency operations which seldom occur but must be executed very fast. Third, in object-oriented databases dynamic transition counts are by no means trivial to determine. Note, that it is not sufficient to count the number of accesses to an attribute referencing another object since there is no guarantee that the object will actually be accessed via the read reference.

Contrary to database systems founded on classical data models, e.g., the relational model, object-oriented database systems also store the behavior of objects, i.e., the code of the operations associated with the objects. This lead to the idea to analyze the operations' code to derive the access patterns statically. Obviously, such a static code analysis avoids the above mentioned disadvantages of dynamically gathered statistics. The outcome of such an analysis could be placement trees as proposed in [BD90], or, as in our case, a set of weighted path expressions. These can then be matched against the current object net and their weights can be accumulated for each successful match to derive so-called *static weighted reference counts* (abbreviated *swrc*) and *static weighted transition counts* (abbreviated *swtc*) representing the access behavior of the analyzed operations.

The rest of this section is organized reflecting the above outline. In Section 3.1 our code analysis method called *decapsulation* is briefly introduced. Section 3.2 defines the static weighted reference and transition counts for a given set of weighted path expressions, and elucidates the exploitation of *swrc* and *swtc* values for clustering.

```
type Gridpoint is
  body [ X, Y, Z: Gridpoint;
         Data: ...;
         ... ]
  operations
    declare YX*move: → Gridpoint;
    ...
end type Gridpoint;
```

```
define YX*move is
  var g: Gridpoint;
begin
S1:  g := self.Y;    !! Assume self.Y ≠ NULL
S2:  while g.X ≠ NULL do
S3:    g := g.X;
S4:  return g;
end define YX*move;
```

Figure 3: Definition of the type *Gridpoint*

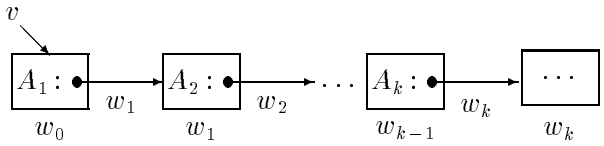
3.1 Decapsulation

The input of the decapsulation process is some operation with its weight denoting the invocation frequency or the criticality of the operation. The output is a set of *weighted path expressions* of the form

$$v[w_0] \,.\, A_1[w_1] \,.\, A_2[w_2] \,.\, \cdots \,.\, A_k[w_k]$$

Every 'dot' in the above expression denotes — at the object level — a reference to some object. The weight factors w_i indicate the importance of these references for clustering

decisions. In order to give a first understanding of the subsequent usage of weighted path expressions, the mapping of the weights of the above path expression to objects and references is visualized by the following illustration:



The squares denote objects and the arrows references. Their corresponding weights are written underneath. In order to avoid too many technical details we do not give the complete formal definition of the decapsulation process which is based on term rewriting and data flow analysis techniques as defined in [ASU87]. Instead, we introduce the main idea of decapsulation by means of an example.[5]

Consider the definition of the operation YX^*move associated with the type *Gridpoint* shown in Fig. 3. Verbally, the access behavior of the operation YX^*move can be described as follows: Starting at **self** — i.e., the object on which YX^*move is invoked — go one step in Y direction and then follow the reference chain in X direction as far as possible. This access pattern is characterized by the expression **self**.$Y.X^*$ where the star indicates the presence of the loop. Taking into account the weight of YX^*move, say 1.5, the outcome of the decapsulation process should be

$$\textbf{self}[1.5]\,.\,Y[1.5]\,.\,(X[1.5])^*$$

(E.g., 1.5 could denote the weight of YX^*move as assigned by the user.)

To capture the behavior of loop-statements the result of decapsulation is a set of weighted *regular* path expressions, i.e. regular expressions over the alphabet $(Var \cup Attr) \times W$ where Var denotes the set of all (persistent) variables, $Attr$ denotes the set of all attributes and W denotes the set of all weights, i.e., a finite subset of the real numbers. Given an operation op and its weight w_{op}, the decapsulation process works as follows:

1. Propagate the operation weight to expressions. Path expressions — denoting the access to objects — only occur within the expressions of op. Hence, to compute a set of weighted path expressions describing the access behavior of op, the weight w_{op} has to be propagated to the expressions of op. This propagation starts by assigning the weight w_{op} to the statement comprising the body of op, and proceeds by assigning to every statement the weight of its directly enclosing statement or its predecessor statement, respectively.

The only exception are **if**-statements: given some **if**-statement "**if** e **then** S_1 **else** S_2" with weight w_{if} and probability $prob(e)$ of the boolean expression e being true, the weights w_1 and w_2 of the statements S_1 and S_2 are defined as $w_1 = w_{if} * prob(e)$ and $w_2 = w_{if} * (1 - prob(e))$, respectively. In the above example all expressions receive the weight 1.5, as YX^*move does not contain any **if**-statements.

2. Extract weighted path expressions from expressions. In step 1 weights are assigned to all expressions of op. These weights are now used to compute for every expression e with weight w_e the set of weighted path expressions $\mathcal{P}(e)$ occurring in e.

[5] The formal definition of the decapsulation process is given in Appendix A. The decapsulation process presented here is a refinement of the decapsulation process applied in [KKM91].

If e is some constant $\mathcal{P}(e) = \emptyset$ or if e is some variable v , $\mathcal{P}(e) = \{\, v[0] \,\}$. If e is of the form $e = v.A_1.\cdots.A_k$

$$\mathcal{P}(e) = \{\, v[w_e] \,.\, A_1[w_e] \,.\, \cdots \,.\, A_{k-1}[w_e] \,.\, A_k[0] \,\}$$

Here, all weights are set to w_e except for the last one that is set to 0 — due to the fact that every 'dot' but the last one in the path expression $v.A_1.\cdots.A_k$ denotes an access to some object. For compound expressions of the form $e = e_1 \cdot e_2$ ('$\cdot$' denotes some binary operator) the set $\mathcal{P}(e)$ is defined as the union of the sets $\mathcal{P}(e_1)$ and $\mathcal{P}(e_2)$.

In the above example $\mathcal{P}(\textbf{self}.Y) = \{\, \textbf{self}[1.5] \,.\, Y[0] \,\}$, and $\mathcal{P}(g.X) = \{\, g[1.5] \,.\, X[0] \,\}$.

3. Rewrite according to assignments. In the operation YX^*move, the assignment statement S_1 stores the value of the expression **self**.Y in the local variable g and the statements S_2 and S_3 access the X attribute of the object referenced by g. From a naive analysis of these statements one could deduce that the operation YX^*move traverses paths of the form **self**.Y and $g.X$ for arbitrary values of g and **self**. But, obviously, the access behavior is not represented well by the separate path expressions **self**.Y and $g.X$. The solution is to replace the variable g with its "value" **self**.Y in the path expression $g.X$ — resulting in the expression **self**.$Y.X$. Performing the indicated rewriting with the weighted path expressions results in the weighted path expression **self**$[1.5] \,.\, Y[1.5] \,.\, X[0]$ which models the access behavior of YX^*move much better.

4. Consider loops. So far, the weighted path expression deduced for statement S_3 is **self**$[1.5] \,.\, Y[1.5] \,.\, X[0]$. Thus, there is no account for S_3 being contained within a loop. To remedy this problem, cycles within the assignments of the loop's body are searched, and for each cyclic assignment of some variable v the path expression corresponding to one iteration is computed. This path expression is labeled by a star '*' and is used to replace v in expressions of the loop's body.

In our example, there exists the simple cyclic assignment $g := g.X$ in the body of the loop statement S_2. Here, the weighted path expression corresponding to one iteration is $g[1.5] \,.\, X[1.5]$. Hence, the expression corresponding to an arbitrary number of iterations is $g[1.5] \,.\, (X[1.5])^*$. This expression is used to replace g in the expressions occurring in S_2, leading to the weighted path expression $g[1.5] \,.\, (X[1.5])^* \,.\, X[0]$. Finally, replacing g by its value from the assignment of S_1 yields the desired weighted path expression **self**$[1.5] \,.\, Y[1.5] \,.\, (X[1.5])^* \,.\, X[0]$. In the final result, the last component $X[0]$ will be discarded because of its weight 0.

The steps 1 to 4 listed above give only a short outline of decapsulation. For clustering according to a mix of operations all operations are decapsulated in the above fashion, variables are replaced by their corresponding types, and all resulting sets of weighted path expressions are merged to one set with weights of matching expressions being accumulated.

Index structures like access support relations [KM90] or generalized materialization relations [KKM91] have a significant impact on clustering. If there is an access support relation, i.e., an index supporting a path expression p of the operation op, then p should not be considered for clustering and thus its weights have to be set to 0. If the results of the operation op are materialized and stored in the corresponding generalized materialization relation, an invocation of op does not induce any access to objects in the object base (assumed the precomputed results are still valid). Thus, materialized functions should not be decapsulated at all — as they need not to be considered for clustering.

3.2 Decapsulation Based Clustering Strategies

Remember that our goal is to compute for each object of the object base the static weighted reference count and for each inter-object reference the static weighted transition count according to some set of weighted path expressions. Every path expression describes — on the type level — a set of paths, i.e., reference chains, of the object net. These paths are called *instantiations* of the path expression. For all path expressions their weights are propagated to the objects and references of their instantiations. All weights propagated to some object (resp. reference) are accumulated yielding its *swrc* (resp. *swtc*) value. Subsequently, this intuitive notion of *swrc* and *swtc* values is formalized.

For a given set of operations, let P be the set of weighted path expressions extracted from these operations. The path expressions in P are regular expressions over the alphabet $(Type \cup Attr) \times W$ where *Type* denotes the set of all types, *Attr* denotes the set of all attributes and W denotes the set of all weights, i.e., a finite subset of the real numbers. To match the path expressions in P with the object net, we proceed in three steps:

1. All regular path expressions in P are merged to one regular path expression PE_P.
2. PE_P is transformed into a deterministic, finite automaton denoted by DFA_P.
3. The words of the language represented by DFA_P are matched with the object net in order to compute the *swrc* and *swtc* values.

In step 1 all regular path expressions of the set $P = \{ p_1, p_2, \ldots, p_n \}$ are combined to one regular path expression $p_1 \,|\, p_2 \,|\, \ldots \,|\, p_n$. This expression is then transformed using algebraic identities for regular expressions; the resulting expression is called PE_P.

In the second step PE_P is transformed into a deterministic finite automaton DFA_P over the alphabet $(Type \cup Attr) \times W$ such that for each word $p \in |PE_P|$, all prefixes of p are in $L(DFA_P)$. Here, $|PE_P|$ is the set of weighted path expressions represented by the regular expression PE_P, and $L(DFA_P)$ is the language accepted by the automaton DFA_P. For example, if $PE_P = Gridpoint[1.5].Y[1.5].(X[1.5]^*)$ then

$$L(DFA_P) \;=\; \{\, Gridpoint[1.5],\; Gridpoint[1.5].Y[1.5],\; Gridpoint[1.5].Y[1.5].X[1.5],\; \ldots \}$$

Finally, the words, i.e., the weighted path expressions, of $L(DFA_P)$ are matched with the object net. We denote the set of all objects of the object net by *Obj*; if $o \in Obj$, $type(o)$ is the type of o.

For some object $o \in Obj$ we define $swrc(o, P, l)$ as the sum of all weights w_i, such that o lies on a path described by some path expression $p \in \{\, p \in L(DFA_P) \mid length(p) \leq l \,\}$[6] and w_i is the weight of the subexpression of p directly referencing object o. More precisely, we define

$$swrc(o, P, l) \;=\; \sum_{w \,\in\, W_{ref}(o,P,l)} w$$

where $W_{ref}(o, P, l)$ is the multi-set containing all weights w_i of subexpressions of the set $\{\, p \in L(DFA_P) \mid length(p) \leq l \,\}$ directly referencing object o:

$$W_{ref}(o, P, l) \;=\; \{ w_n \mid \exists\, o' \in Obj,\; t[w_0].A_1[w_1].\cdots.A_n[w_n] \in L(DFA_P),\; 0 \leq n \leq l :\; type(o') = t \;\wedge\; o'.A_1.\cdots.A_n = o \}$$

[6] For some path expression $p = t[w_0].A_1[w_1].\cdots.A_k[w_k]$ and $k \geq 0$, $length(p) = k + 1$.

For two objects $o_1, o_2 \in Obj$ being connected by some reference (o_1, o_2) the static weighted transition probability $swtc(o_1, o_2, P, l)$ is the sum of the weights w_i, such that the reference (o_1, o_2) is part of a path described by some weighted path expression $p \in \{\, p \in L(DFA_P) \mid length(p) \leq l \,\}$ and w_i is the weight of the subexpression of p directly referencing the "terminal" object o_2 of the edge (o_1, o_2). More precisely, we define

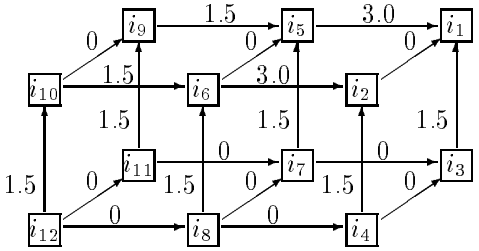
$$swtc(o_1, o_2, P, l) \;=\; \sum_{w \in W_{trans}(o_1, o_2, P, l)} w$$

where $W_{trans}(o_1, o_2, P, l)$ is the multi-set containing all weights w_i of subexpressions of $\{\, p \in L(DFA_P) \mid length(p) \leq l \,\}$ describing some reference from o_1 to o_2:

$$\begin{aligned} W_{trans}(o_1, o_2, P, l) \;=\; \{ w_n \mid \exists\, o' \in Obj,\; & t[w_0] \,.\, A_1[w_1] \,.\, \cdots \,.\, A_n[w_n] \in L(DFA_P), \\ & 1 \leq n \leq l : \; type(o') = t \;\wedge\; o'.A_1.\cdots.A_{n-1} = o_1 \\ & \wedge\; o'.A_1.\cdots.A_n = o_2 \} \end{aligned}$$

For sequence based clustering algorithms, *swrc* will be used as a presort order and *swtc* is used for ordering of successors. For partitioning based algorithms, the edges of the object graph will be labeled with their according *swtc* value.

Example: Consider the database extension shown in Fig.1, and let the set P equal $\{\, Gridpoint[1.5] \,.\, Y[1.5] \,.\, (X[1.5])^* \,\}$. The illustration on the right hand side shows the $swtc(o_1, o_2, P, 3)$ values for all references of the example extension. The computation of the $swrc(o, P, 3)$ values is analogous. ◇



4 Simulation Environment

Although clustering strategies are incorporated into almost all object base management systems—research prototypes as well as products—no system we know of provides for the replacement of the implemented clustering mechanism. Therefore, we have developed the extensible access simulator for object bases (called TEXAS). Even though TEXAS is based on our object-oriented database system GOM [KMWZ91] the results carry over to other object models. Every experiment is divided into three phases: the training phase to collect statistics for the dynamic methods, the clustering phase where static access patterns are determined and the objects are mapped to pages, and the benchmark phase to evaluate the generated mapping of objects to pages.

We implemented two benchmarks on top of TEXAS: the Sun Engineering Benchmark [CS90] and the Grid Benchmark. The Sun benchmark is widely known for benchmarking object-oriented database systems. Although providing a good first understanding of the performance of different clustering strategies, this benchmark exhibits several disadvantages. First, because of its simplicity, the schema of the Sun Benchmark does not well reflect real-world applications (e.g., in GOM, the schema of the Sun Benchmark consists of only two tuple-structured types having four attributes in total, one set-structured type, and one operation). Contrary to the Sun benchmark schema, real-world applications, especially engineering applications, often contain more complex types and several operations

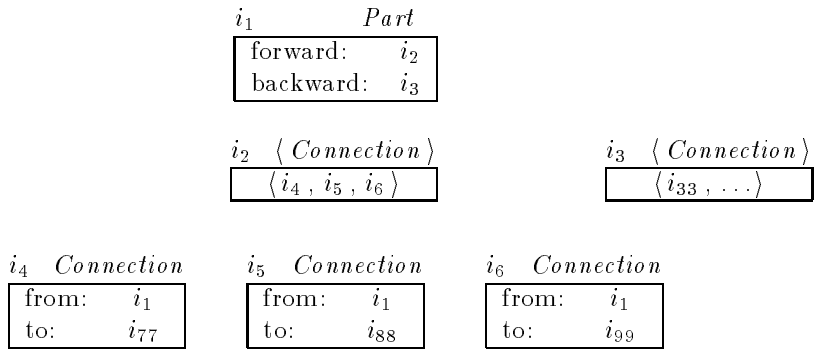


Figure 4: The structure of the Sun benchmark database

having diverging access patterns. Second, the behavior of the benchmark operations—and the benchmark results—are difficult to "understand" because of the random structure of the database extension. All connections between objects are chosen randomly, preventing any regular structure of the object net.

Therefore, we created the Grid benchmark having the following properties: (1) It provides a high degree of recursiveness which is a challenge to any clustering strategy, (2) it provides a (rather) complex schema with several operations exhibiting different access behaviors, and (3) the structure of the benchmark database includes regular patterns as well as random connections between objects.

4.1 The Sun Benchmark

The database of the Sun benchmark consists of N objects of type *Part*, each of them being connected to exactly K children (also of type *Part*). The database is built as follows: N objects of type *Part* are created. For each of these objects, the K children are randomly selected with probability 0.9 from the objects being "close" to the parent object (within 1% of N) in terms of OID distance, and with probability 0.1 from all objects. Every child keeps back-references to all its parents. All references between *Part* objects are realized via separate objects of type *Connection*. The structure of the database created by the Sun benchmark is shown in Fig. 4 (for $K = 3$).

Two access operations, *Lookup* and *Traversal*, are defined for the Sun benchmark. *Lookup* randomly selects K *Part* objects and visits each of them separately. This operation cannot be used for evaluating clustering algorithms because of its totally random access behavior. *Traversal* performs a recursive depth-first traversal along the forward pointers of one randomly selected *Part* object. (The *forward* pointers connect parents with their children.) The depth of the traversal is limited by some non-negative integer D. *Traversal* accesses $\sum_{i=0}^{D} K^i$ *Part* objects with possible duplicates.

4.2 The Grid Benchmark

The database of the Grid benchmark consists of N objects of type *Gridpoint*, each of them having 8 attributes X, Y, Z, X_{inv}, Y_{inv}, Z_{inv}, *random*, and *data*. The *Gridpoints* constitute a regular grid via their X, Y, and Z references—as shown in Fig. 1 on Page 4. The total number of *Gridpoint* objects equals $N = NG * NX * NY * NZ$ where NG is

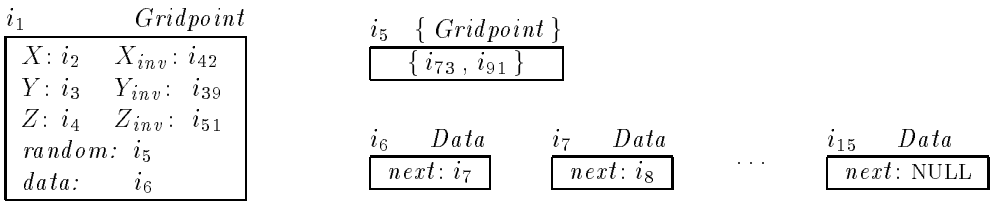


Figure 5: The structure of the Grid benchmark database

Operation	Decapsulation Results[7]
Traversal	{ *Gridpoint*.(*random*.$)^*$ }
ShortPaths	{ *Gridpoint*.*X*, *Gridpoint*.*Y*, *Gridpoint*.*Z*, *Gridpoint*.X_{inv}, *Gridpoint*.Y_{inv}, *Gridpoint*.Z_{inv} }
LinearPath	{ *Gridpoint*.*X*.*Y*.*Z*.Y_{inv}.X_{inv}.*data*.*next* }
YX^*move	{ *Gridpoint*.*Y*.$(X)^*$ }
$Y^*ZZ_{inv}move$	{ *Gridpoint*.$(Y)^*$.*Z*, *Gridpoint*.$(Y)^*$.Z_{inv} }
$next^*move$	{ *Gridpoint*.*data*.$(next)^*$ }

Table 2: Operations of the Grid benchmark

the number of grids, and NX, NY, and NZ give the lengths of the grids in X, Y, and Z dimension. Every *Gridpoint* also keeps back-pointers (X_{inv}, Y_{inv}, Z_{inv}) to its neighbor objects in X^{-1}, Y^{-1}, and Z^{-1} direction, respectively.

Besides the regular grid patterns of the database, there are also randomly selected object references. For each *Gridpoint*, these are stored in a separate set object referenced by the attribute *random*. This set contains between 0 and K references to objects equally selected among all *Gridpoints* (in the average $K/2$). The last component of *Gridpoint* is the attribute *data* referencing a linear list of L objects of type *Data*. The objects of this list are not shared between several *Gridpoints*. The structure of the grid database for $K = 4$ and $L = 10$ is illustrated by Fig. 5.

We have defined six operations each of them performing different access patterns. The operations together with the path expressions obtained from their decapsulation are summarized by Table 2. The first operation of the operation mix, *Traversal*, is similar to the *Traversal* operation of the Sun benchmark: the grid is traversed depth-first following *random* pointers all the time. Again, the depth of the traversal is limited by some non-negative integer D. In the path expression *Gridpoint*.$(random.\$)^*$ extracted from the operation *Traversal* the set-iterator symbol \$ denotes the access to all elements of some set-structured object.

The operations *ShortPaths* and *LinearPath* merely traverse a set of predefined paths as specified in Table 2. The next two operations, YX^*move and $Y^*ZZ_{inv}move$, traverse a path in X direction or in Y direction, respectively, until the end of the grid is reached. YX^*move access the Y attribute of the receiver object before starting the traversal in X direction, and $Y^*ZZ_{inv}move$ accesses for each object o encountered along its traversal in Y direction the attributes $o.Z$ and $o.Z_{inv}$. The last operation of the mix, $next^*move$, accesses all elements of the *data* list associated with some *Gridpoint*.

5 Benchmark Results

In this section, we present a selection of all experiments we have carried through. This section is divided into two subsections. In Section 5.1, the Sun benchmark is used to evaluate nine different combinations of page mapping algorithms and methods for deriving access patterns of the applications. From these results, the best two page mapping algorithms and the best two methods for deriving the access patterns are determined and further evaluated using the Grid benchmark. The results for the corresponding four combinations are shown in Section 5.2.

5.1 Sun Benchmark

The parameters of the Sun benchmark are $N = 1300$, $K = 3$, and $D = 5$ (7800 objects in total). The object size for *Part* and *Connection* is 52 bytes, and for $\langle$ *Connection* $\rangle$ 8 bytes per element. During the training phase 4000 *Traversal* operations were performed ensuring that—in the average—three traversals where started at each *Part* object. We measured nine combinations of page mappings and methods for deriving access patterns as shown by the following table:[8]

		derivation of access behavior	
		dynamic	static
page mapping	partitioning based	*kern.dtc* *ggp.dtc*	*kern.swtc* *ggp.swtc*
	sequence based	*drc* | *bstf.dtc* *kruskal.dtc* | *id*	*swrc* | *bstf.swtc* *kruskal.swtc* | *id* *type* | *pt*

Some of these clustering strategies were already examined in previous work, or realized in OODBMS prototypes: *kern.dtc* was examined in [TN91], *kruskal.dtc* | *id* was examined in [CH91], *drc* | *bstf.dtc* is realized in Cactis, and *type* | *pt* is realized in O_2. The remaining five combinations are new using the decapsulation based access information or the greedy graph partitioning algorithm (or both).

From the decapsulation of the operation *Traversal* the weighted path expression

$$Part[1] \,.\, (forward[1] \,.\, \$[1] \,.\, to[1])^*$$

was determined and was used for computing the *swrc* and *swtc* values. For the clustering strategy *type* | *pt*, we used the (linear) placement tree described by the path expression

$$Part.forward.\$.to.forward.\$.to.forward$$

A similar placement tree was determined as the best possible placement tree for the traversal operation in a subset of the Tektronix benchmark described by Benzaken et al. [HBD91].

[7] The weights of all paths expressions equal the weights of the corresponding operations.

[8] Partitioning based clustering algorithms are denoted by *PartAlgo.EdgeLableAlgo*. We use *kern* as an abbreviation for the Kernighan & Lin graph partitioning algorithm, and *ggp* for our greedy graph partitioning algorithm.

5.1.1 Single Page Buffer

In the first benchmark, we evaluated the performance of the nine algorithms in terms of page faults under varying page sizes, employing a page buffer of size 1 (i.e., the number of logical page references of the application was measured). When using a larger buffer of size > 1 not only the clustering algorithm but also the choice of the replacement strategy influences the number of page faults—thereby diluting the effect of good clustering. As we wanted to omit this influence, we used a single page buffer for most of the experiments described in this paper. A second reason for using a single page buffer is that clustering aims at grouping strongly related objects on one page. To measure this grouping ability of algorithms in terms of page faults, a single page buffer must be used. The results of the first benchmark are visualized in Fig. 6 with the page size varying from 1 K to 4 K. 10 *Traversals* with randomly selected start objects were performed during this benchmark producing 22736 object accesses.

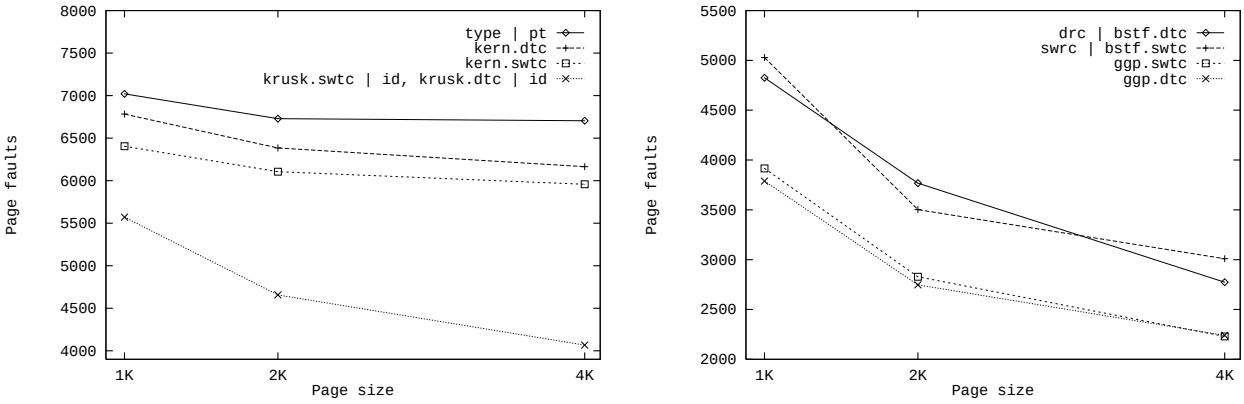


Figure 6: *Traversal* under varying page sizes

Fig. 6 shows that all clustering algorithms achieved a performance gain of 70 % or better compared to random placement of the objects. Interestingly the plots of the clustering algorithms employing the same page mapping algorithm run very close—indicating that the statically derived access information (*swrc*, *swtc*) performs as well as the dynamically derived access information (*drc*, *dtc*)—without inducing the disadvantages of monitoring. Note that computing the *drc* and *dtc* values for 4000 training operations lasted 2 days on a Sun Sparc 2 workstation, whereas the time for computing the *swrc* and *swtc* values was below 10 minutes—which can even be done off-line, i.e., does not have to be done during regular database use.

Our greedy graph partitioning algorithm applied to both dynamically and statically derived statistics performed best among the tested algorithms reducing the number of page faults by about 90 % (for 4 K pages). The clustering algorithms using the best first algorithm and the Kruskal algorithm also achieved good clustering results. The worst performance exhibited the algorithms based on the *kern* algorithm and on placement trees. The result of the *kern* algorithm is surprising as we performed 300 iterations over all possible pairs of pages running about 12 hours on a Sun Sparc 2 workstation. However, this algorithm failed to achieve a good partitioning of the object graph because of (1) the fixed number of pages determined by the initial partition and (2) the varying object sizes of the Sun benchmark (objects of different size cannot be swapped between nearly full

pages). Contrary to *kern* the *ggp* algorithm is a constructive graph partitioning algorithm and, thus, avoids the problems mentioned above.

The algorithm *type* | *pt* exhibited the worst performance of the nine algorithms. Interestingly, the plot of this algorithm also shows the smallest decline upon the increase of the page size. This can be explained as follows: as the traversal component of the algorithm is specified by the given placement tree, only small groups of objects are visited during each traversal. Hence, an increase of the page size increases the number of groups per page, but does not necessarily lead to an improvement of the clustering scheme because objects of different groups stored on a page might not be related at all.

5.1.2 Varying the Buffer Size

Since the size of the buffer could have an impact on the relative performance of the clustering algorithms, we compared the algorithms under varying buffer sizes. Four experiments were conducted with buffer sizes varying from 1 to 8 pages for a page size of 4 K (8 pages of 4 K are about 10 % of the database volume). A LRU replacement strategy was employed. The results of this benchmark are visualized in Fig. 7. Again, we conducted 10 *Traversals* with 22736 object accesses.

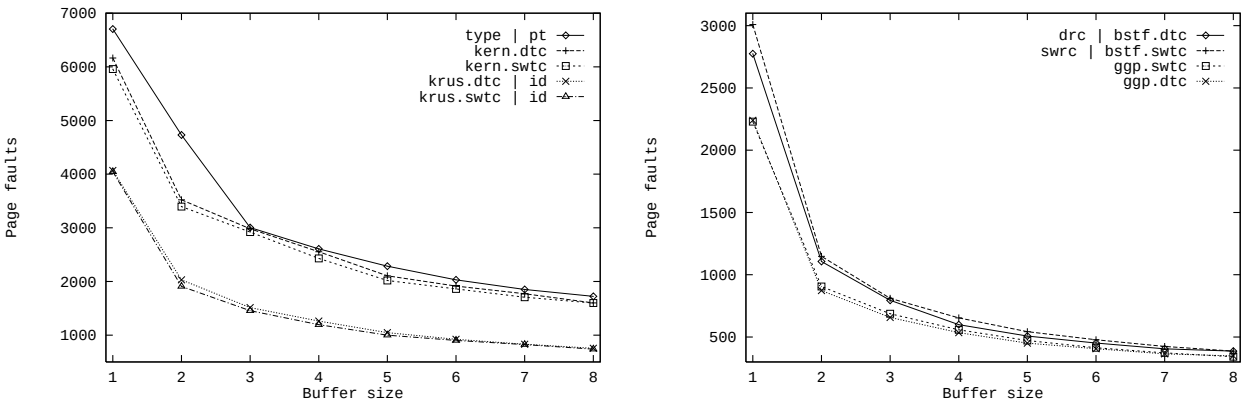


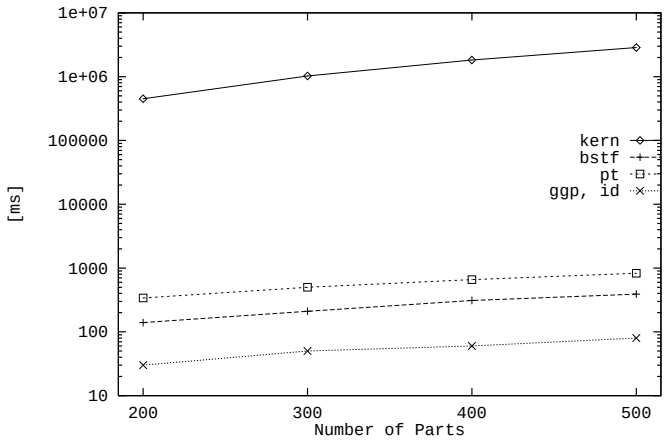
Figure 7: *Traversal* under varying buffer sizes

Although the differences among the clustering algorithms decrease as the buffer size increases, the relative order among the algorithms remains invariant. As in the previous benchmark, the algorithms can be divided into three classes according to their performance. Again the *bstf* and *ggp* based algorithms performed best, with the latter being slightly superior on large buffers and essentially superior on small buffers. The static and dynamic versions of the Kruskal algorithm together with the identical page mapping represent the second group. The last group contains the algorithms *kern.dtc*, *kern.swtc* and *type* | *pt* performing about a factor $3-4$ worse than the *ggp* based algorithms. In [TN92] similar results were observed for a buffer size of 10 % of the database volume and a desired LRU hit ratio of 90 %. However, in their work, all objects were of equal size leading to much better results for the *kern* algorithm.

5.1.3 Run Time of Clustering Algorithms

In order to verify the algorithm's value for practical use, i.e., for clustering object bases of realistic size, their run time has been evaluated in the next benchmark. The number

of *Parts* of the object base was varied from 200 to 500. Fig. 8 shows the results. The run time of the algorithms is plotted against the logarithmically scaled y-axis.[9] Except for *kern* all algorithms exhibited a very low run time of 1 sec or below. The run time of the *kern* algorithm varied from 7.5 min to about 50 min due to the algorithm's complexity of $O(n^{2.4})$. This—for the rather small database volume—tremendous run time together with the rather poor clustering results especially for objects of various sizes—as illustrated by the first benchmark—disqualify the *kern* algorithm for clustering of realistically large databases.



Page Size (KB)	4
No. of *Parts*	200, 300, 400, 500
# ops	10
# accessed objects	≈ 22000

Figure 8: Run time of clustering algorithms

5.2 Grid Benchmark

In this subsection, we further examine the "winners" from the previous benchmarks—the page mappings *ggp* and *bstf* together with the dynamic and static reference and transition counts—using our Grid benchmark. The goals of these experiments are: (1) to compare the quality of static and dynamic access information utilizing the same page mapping algorithms, and (2) to compare the *ggp* with the *bstf* algorithm if both utilize the same access information.

The database of the Grid benchmark consists of two separate grids with $NX = 8$, $NY = 8$, $NZ = 16$ (2048 grid objects), $K \in \{0, \ldots, 4\}$ (mean 2), and $L = 4$ constituting 11893 objects in total. The object size for the types *Gridpoint* and *Data* is 300 bytes, and the size of objects of type { *Gridpoint* } is 8 bytes per element. The operation mix includes all operations listed in Table 2.[10] For each experiment, the operation mix is described by a set of quadruples (*op*, W, Tr, Be) with W denoting the weight of the operation *op* (as utilized by the decapsulation process), and Tr and Be denoting the probability that *op* is chosen from the set of operations during the training phase or the benchmark phase, respectively. During the training phase 4000 operations were performed, randomly selected according to the Tr probabilities. During the benchmark phase, 1000 operations were performed (selected according to the Be probabilities). For the reasons explained

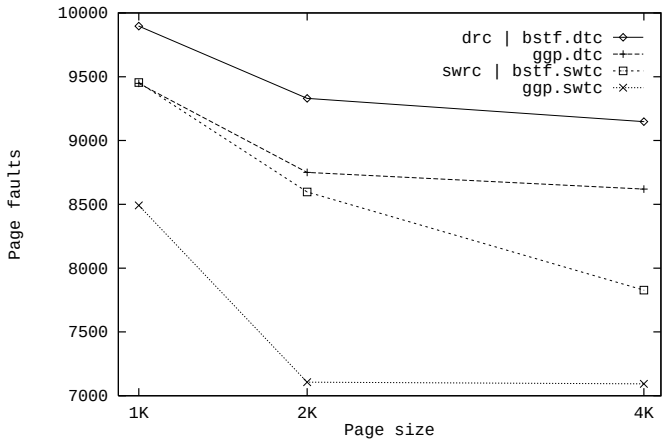
[9] Note, that we measured only the run time of the graph partitioning algorithms and the traversal algorithms, respectively. The run time for pre-sorting the objects and computing the *dtc* and *swtc* values was not included.

[10] The depth of the *Traversal* operation was $D = 2$.

above, we describe here only those experiments that were carried out with a single page buffer.

5.2.1 Uniformly Distributed Operation Mix

The objective in the first benchmark is to examine the performance of clustering algorithms under a broad operation mix with strongly diverging access patterns. 1000 operations were executed uniformly selected among all six operations—inducing 11759 object accesses. To model uniform importance of all operations within the operation mix, the weight factors W of all operations were set to 1. The results of this benchmark are visualized in Fig. 9.



Operation	W	Tr	Be
$Traversal$	1	1/6	1/6
$ShortPaths$	1	1/6	1/6
$LinearPath$	1	1/6	1/6
$YX^*\,move$	1	1/6	1/6
$Y^*ZZ_{inv}\,move$	1	1/6	1/6
$next^*\,move$	1	1/6	1/6
# ops (training)	4000		
# ops (benchm.)	1000		
# accessed objects	11759		

Figure 9: Uniformly distributed operation mix

It is intuitively known that clustering strategies are less efficient under a mix of operations exhibiting different access behavior and working on the same set of objects. Indeed, the plots in Fig. 9 show a ratio between page faults and object accesses ranging from 78 % (*drc | bstf.dtc*, 4 K pages) to 60 % (*ggp.swtc*, 4 K pages)—constituting a much smaller performance gain with respect to random object clustering than the clustering results obtained from the Sun benchmark.

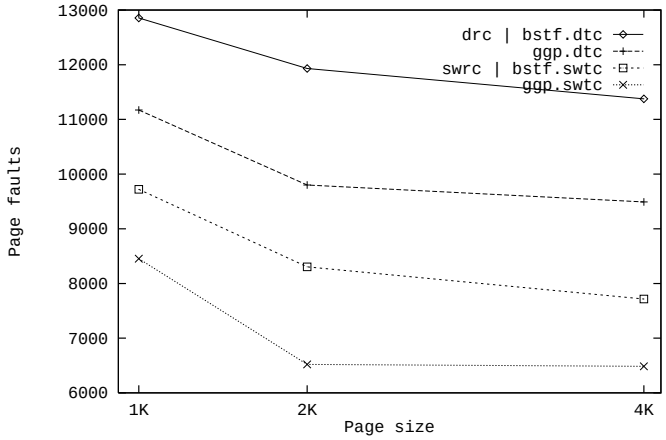
The results of this benchmark indicate further that the decapsulation based reference and transition counts provide a better model of the complex access behavior of the operation mix than the dynamic reference and transition counts. For both page mapping algorithms—*ggp* and *bstf*—the ratio of page faults of the decapsulation based version and the dynamic version is about 85 %. Although this result strongly depends on the actual operation mix and database structure, it was also backed by further experiments where decapsulation based reference and transition counts showed comparable or even better quality for clustering decisions than dynamic reference and transition counts.

The third result of this benchmark is the performance of the *ggp* algorithm compared to the *bstf* algorithm. *ggp* out-performed *bstf* independently of the utilized access information. This result is also backed by a large variety of experiments we carried through on both the Sun and the Grid benchmark.

5.2.2 Changing the Operation Mix

Although object bases are usually loaded with a broad mix of applications with strongly diverging access patterns—as measured in the previous benchmark—there may be a few distinguished particulary critical operations that are to be specifically supported by clustering. As the access patterns of these critical operations may be totally submerged by the total load of the object base they cannot be detected by dynamic monitoring under normal database use. Of course the access patterns of these operations could be "learned" during a specific training phase—but, this is only possible if the operations have no side effects and, furthermore, it would induce an additional load on the object base system. Nevertheless, the access patterns of critical operations can be deduced by decapsulating the operations' implementation.

We designed the following benchmark in order to measure the performance of the four algorithms from the prior benchmark based on an operation mix containing the critical operations *Traversal* and YX^*move. The weights of these operations were set to 100—denoting their criticality—and the weights of the remaining operations were set to 1. During the training phase, all operations were invoked with uniform probability 1/6 (representing a mixed workload during normal operation of the object base), whereas during the benchmark phase only the critical operations *Traversal* and YX^*move were invoked with probability 1/2.



Operation	W	Tr	Be
$Traversal$	100	1/6	1/2
$ShortPaths$	1	1/6	0
$LinearPath$	1	1/6	0
YX^*move	100	1/6	1/2
$Y^*ZZ_{inv}move$	1	1/6	0
$next^*move$	1	1/6	0
# ops (training)	4000		
# ops (benchm.)	1000		
# accessed objects	15620		

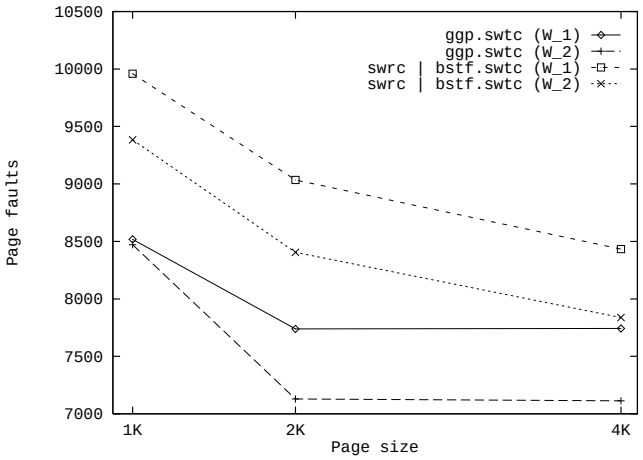
Figure 10: Operation mix with critical operations

The results of this benchmark are shown in Fig. 10. As expected, the clustering algorithms based on the *swrc* and *swtc* values performed much better than their corresponding dynamic versions. Whereas in the previous benchmark, the ratio of page faults of the static and the dynamic version of clustering algorithms employing the same page mapping was about 85 %, it declined to 70 % in this benchmark. Please note that independent of the utilized access information, the *ggp* algorithm achieved better clustering results than the *bstf* page mapping.

We conducted a second experiment in order to measure the penalty that is imposed on the "normal" use of the database by adapting the operation weigths according the operations' criticality. In this experiment, the "normal" use of the database was modelled by an uniformly distributed operation mix. We compared the *ggp* and *bstf* algorithms using the operations' criticality as operation weights $(100, 1, 1, 100, 1, 1)$ with the *ggp*

and *bstf* algorithms using equal operation weights corresponding to the operation mix $(1,1,1,1,1,1)$. The operation mix is a set of quadruples (Op, W_1, W_2, Be) with W_1 and W_2 denoting the criticality or the weights corresponding to the invocation probabilities, respectively. The result are visualized in Figure 11.

As expected, the algorithms using the operations' criticality (W_1) perform worse than the algorithms using the "correct" access patterns of the benchmark (W_2). Nevertheless, the algorithms using the weights W_1 perform comparable or even better than the algorithms based on dynamic access information that were measured in the benchmark of the preceding subsection (see Fig. 9). Please note, that by adapting the weights of the critical operations (which were set to 100 in the above benchmark) the algorithms could be further tuned to reach a better performance during "normal" use of the database.



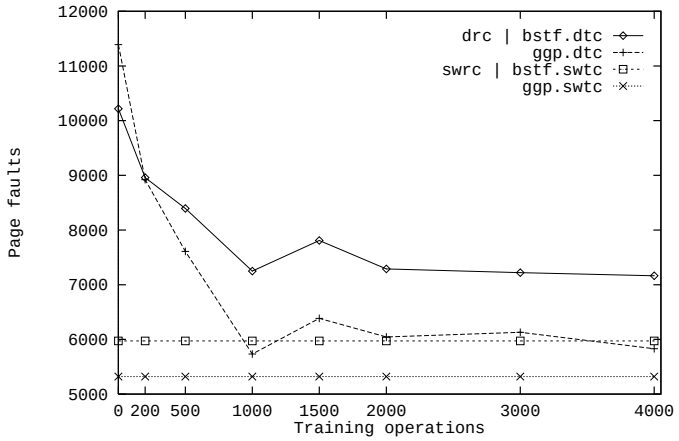
Operation	W_1	W_2	Be
Traversal	100	1	1/6
ShortPaths	1	1	1/6
LinearPath	1	1	1/6
YX^*move	100	1	1/6
$Y^*ZZ_{inv}move$	1	1	1/6
$next^*move$	1	1	1/6
# ops (training)	–		
# ops (benchm.)	1000		
# accessed objects	11759		

Figure 11: Effect of adapting the operation weights

5.2.3 Length of Training Phase

The inferior clustering quality of algorithms relying on dynamic reference and transition counts in the case of critical operations (as demonstrated by the previous benchmark) indicates that a specific training phase for monitoring the access patterns of the critical operations is necessary. As the training phase induces an additional load upon the object base system, it should be as short as possible. The next benchmark was designed to examine the impact of the length of the training phase on the clustering results of dynamic clustering algorithms. The workload consisted of a mix of four operations whose weights and invocation probabilities are shown in Fig. 12. During the benchmark phase, 1000 operations were executed. The number of operations executed during the training phase varied from 0 (no access information) to 4000.

The graph of Fig. 12 shows the number of page faults as a function of the length of the training trace. Obviously, the static algorithms *swrc* | *bstf.swtc* and *ggp.swtc* are not affected by the length of the training trace, whereas the plots for the dynamic algorithms *drc* | *bstf.dtc* and *ggp.dtc* show a steep decline between 0 and 1000 training operations and a slow decline between 1500 and 4000 operations. For a training trace of length 0, no access information is available. In this case, the page mapping algorithms randomly group (structurally) related objects together. As the number of training operations increases better access information becomes available, leading to a reduction of page faults. But



Operation	W	Tr	Be
$Traversal$	0	0	0
$ShortPaths$	20	0.2	0.2
$LinearPath$	10	0.1	0.1
YX^*move	0	0	0
$Y^*ZZ_{inv}move$	60	0.6	0.6
$next^*move$	10	0.1	0.1
# ops (training)	0 - 4000		
# ops (benchm.)	1000		
# accessed objects	14288		
page size	4K		

Figure 12: Importance of the training phase for dynamic sampling

even after 4000 training operations the dynamic versions of the clustering algorithms do not reach the performance of the corresponding static versions. Interestingly, the number of page faults of the dynamic algorithms increases between 1000 and 1500 training operations. Thus, increasing the length of the training trace does not always lead to better clustering results.

6 Conclusion

In this paper we have distinguished two dimensions that are crucial for clustering decisions: (1) determining the access patterns of object base applications and (2) mapping objects to pages based on the characteristics obtained in (1). Along the first dimension, we distinguish dynamic monitoring of the object base from static operation analysis, i.e., decapsulation. Along the second dimension, we separate the algorithms into sequence based and partitioning based mappings of objects to pages.

The contribution of this paper is two-fold. Along the first dimension, i.e., obtaining the access behavior, we developed a new technique, called *decapsulation*, for static operation analysis. The *decapsulation* yields a set of weighted path expressions where the weights represent traversal frequency or "criticality". The weighted path expressions are matched with the actual object net to obtain static weighted reference counts for objects and static weighted transition counts for inter-object relationships. Along the second dimension, i.e., mapping objects to pages, we proposed a new heuristic partitioning based algorithm called *greedy graph partitioning (ggp)*. Whereas former proposals for graph partitioning are prohibitively expensive, our *ggp* algorithm exhibits a moderate run time complexity that makes it suitable even for (realistically) large object bases.

Our extensive benchmarking—of which only a small fraction could be presented in the paper—indicates that the *ggp* page mapping algorithm is superior to the various sequence based mappings. Furthermore, the benchmarks substantiate that in many cases reference and transition counts obtained from static operation analysis (decapsulation) are superior to those obtained from dynamic monitoring, especially if certain operations are particularly *critical* and have to be given priority in clustering decisions. Even under the objective of maximizing the overall throughput of a wide variety of operations, static analysis performs comparable to dynamic monitoring—without inducing the additional

overhead of monitoring.

Acknowledgements

This work was supported by the German Research Council DFG under contracts SFB 346 and Ke 401/6-1. Peter C. Lockemann's continuous support of our research is gratefully acknowledged. Christoph Müller and Heiner Spies carried out the implementation of the decapsulation process.

References

[ASU87] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1987.

[BD90] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. In A. Dearle, G. Shaw, and S. Zdonik, editors, *Implementing Persistent Object Bases*, pages 403–412, Martha's Vineyard, Sep 1990. Morgan Kaufmann.

[BKKG88] J. Banerjee, W. Kim, S. J. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE Trans. on Software Engineering*, 14(11):1684–1699, Nov 1988.

[CDRS86] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 91–100, Kyoto. Japan, Aug 1986.

[CH91] J. R. Cheng and A. R. Hurson. Effective clustering of complex objects in object-oriented databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 22–31, Denver, CO, May 1991.

[CS90] R. Cattell and J. Skeen. Engineering database benchmark. Technical report, Database Engineering Group, Sun Microsystems, Mountain View, Ca., April 1990.

[HBD91] G. Harrus, V. Benzaken, and C. Delobel. Measuring performance of clustering strategies: The CluB-0 benchmark. Technical Report 66-91, Altair, GIP-Altair-INRIA, BP-105, 78153 Le Chesnay, France, Jan 1991.

[HK89] S. E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. on Database Systems*, 14(3):291–321, Sept 1989.

[HZ87] M. Hornick and S. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Trans. on Office Information Systems*, 5(1):70–95, Jan 1987.

[KCB88] W. Kim, H. T. Chou, and J. Banerjee. Operations and implementation of complex objects. *IEEE Trans. on Software Engineering*, 14(7):985–996, Jul 1988.

[KKM91] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 258–267, Denver, CO, May 1991.

[KL70] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, Feb 1970.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 364–374, Atlantic City, NJ, May 1990.

[KMWZ91] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. of the German Conf. on Databases in Office, Engineering and Science (BTW)*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik Fachberichte Nr. 270.

[Kru56] J. B. Kruskal. On the shortest spanning subgraph of a graph and the travelling salesman problem. *Proc. of the Amer. Math. Soc.*, 7:48–49, 1956.

[LNS90] R. Lipton, J. Naughton, and D. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–11, 1990.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 256–276, 1984.

[Sal69] A. Salomaa. *Theory of Automata*, volume 100 of *Int. Series of Monographs in Pure and Applied Mathematics.* Pergamom Press, Oxford, 1969.

[Sch77] M. Schkolnick. A clustering algorithm for hierarchical structures. *ACM Trans. on Database Systems*, 2(1):27–44, Mar 1977.

[Sta84] J. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[TN91] M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 12–21, Denver, CO, May 1991.

[TN92] M. Tsangaris and J. Naughton. On the performance of object clustering techniques. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 144–153, San Diego, CA, June 1992.

A Decapsulation

This section summarizes the principles of decapsulation. We define the decapsulation process only for a subset of the GOM data model, but it can easily be extended to cover the full model. Without loss of generality, we may assume that there are no superfluous assignments in the operations, i.e., assignments where a variable is set and subsequently overwritten before the value is used elsewhere. (Statements of this kind may be removed without altering the semantics.) The input of the decapsulation process is some operation op with its associated weight factor w_{op}. The output is a set of weighted path expressions — denoted by $paths(op)$ — describing the access behavior of op. Weighted path expressions are of the form $v[w_0].A_1[w_1].\cdots.A_k[w_k]$ where v is a variable referencing some object, and the A_i are attribute names.[11] The weight factors w_i indicate how often paths of this kind probably will be traversed during an invocation of op. If all weight factors of some path expression p equal 0 the weights can be omitted (in this case p is called a *non-weighted* path expression).

The definition of the decapsulation process proceeds in five steps:

1. For some expression e we define the set $\mathcal{P}(e)$ of all weighted path expressions occurring in e.
2. For each statement S the execution probability of S is computed. The execution probabilities of statements are used to weight the path expressions in the set $paths(op)$ (see step 5).
3. Variables are replaced by their "value" according to preceding assignments.
4. The semantics of loop-statements are considered.
5. The final result — the set $paths(op)$ — is computed by merging the path expressions extracted from the expressions and re-weighting these expressions using the weight of the operation and the execution probabilities of the statements.

This section is organized reflecting the above outline.

A.1 Expressions

The set of all path expressions occurring in some expression e is denoted by $\mathcal{P}(e)$. The following definition introduces the mapping $\mathcal{P}$.

Definition A.1 (Path Extraction for Expressions)
Let c be some constant, v some variable, $A_1, \ldots, A_k$ attributes, and $\cdot$ some binary operator, e.g., $+$, $-$, $<$. Further, let e_1, e_2 be two expressions. The mapping $\mathcal{P}$ is defined by the following equations:

$$\begin{aligned}
\mathcal{P}(c) &= \emptyset \\
\mathcal{P}(v) &= \{\, v[0] \,\} \\
\mathcal{P}(v.A_1.\cdots.A_k) &= \{\, v[1].A_1[1].\cdots.A_{k-1}[1].A_k[0] \,\} \\
\mathcal{P}(e_1 \cdot e_2) &= \mathcal{P}(e_1) \uplus \mathcal{P}(e_2)
\end{aligned}$$

[11] Additionally, weighted path expressions may contain operation invocations and the set iterator symbol \$ denoting the access to all elements of some set- or list-structured object. However, in the remainder of this section we consider only path expressions containing neither operation invocations nor the iterator symbol \$.

Statement S	*Data-flow equations for S*		
$v := e$	$w_{out}(S)$	$=$	$w_{in}(S)$
return e	$w_{out}(S)$	$=$	0
if e **then** S_1 **else** S_2	$w_{in}(S_1)$	$=$	$prob(e) * w_{in}(S)$
	$w_{in}(S_2)$	$=$	$(1 - prob(e)) * w_{in}(S)$
	$w_{out}(S)$	$=$	$w_{out}(S_1) + w_{out}(S_2)$
while e **do** S_1	$w_{in}(S_1)$	$=$	$prob(e) * w_{in}(S)$
	$w_{out}(S)$	$=$	$(1 - prob(e)) * w_{in}(S) + w_{out}(S_1)$
$S_1; S_2$	$w_{in}(S_1)$	$=$	$w_{in}(S)$
(S_2 must not be a	$w_{in}(S_2)$	$=$	$w_{out}(S_1)$
sequence of statements)	$w_{out}(S)$	$=$	$w_{out}(S_2)$

Table 3: Data-flow equations for computing execution probabilities for statements

□

The operator $\uplus$ computes the union of two sets P_1, P_2 of weighted path expressions. If there are two path expressions $p_1 \in P_1$ and $p_2 \in P_2$ such that the variable and attributes of p_1 equal those of p_2 then p_1 and p_2 are joined to one path expression in $P_1 \uplus P_2$ with corresponding weights being added.

A.2 Execution Probabilities

Path expressions only occur within expressions, and expressions are embedded in statements. Our goal is to assign to the each statement S the probability that S will be executed during some invocation of *op*. The execution probability of S is then propagated to the path expresssion occuring in S.

We define two probability factors for each statement S: $w_{in}(S)$ denotes the probability that statement S will be executed, i.e., $w_{in}(S)$ is the *execution probability* of S. $w_{out}(S)$ denotes the probability, that after the execution of S terminated the statement succeeding S will be executed. $w_{out}(S)$ equals $w_{in}(S)$ if there are no **return** statements in S. The computation of the probabilities w_{in} and w_{out} are defined by the data-flow equations of Table 3. The computation starts with the statement comprising the body of *op* denoted by S_0 with $w_{in}(S_0) = 1$. Then, the computation of w_{in} and w_{out} proceeds with subsequent and directly enclosed statements.

In the equations of Table 3 $prob(e)$ equals the probability of some boolean expression e to be *true*. This probability can be estimated as follows:

- Statistics about the state of the database may contain information to estimate selectivity factors. The statistics can be determined either statically by inspecting the database or dynamically by keeping result traces for boolean expressions (e.g., see [LNS90, PSC84]).
- If only a rough branching probability is needed the values of $prob(e)$ can be set to a constant, e.g., 0.5, for all expressions e.
- The database programmer can be asked to set the values $prob(e)$.

Subsequently, we abbreviate the execution probability $w_{in}(S)$ of some statement S by w_S.

A.3 Statements

Based on the mapping $\mathcal{P}$ from expressions to sets of path expressions and the execution probabilities of statements we can now define the extraction of weighted path expressions from statements. In this section, we do not consider the effect of loop-statements, e.g., **while** statements. These are discussed in Section A.4.

For the extraction of path expressions from statements the effect of variable assignments occurring in preceding statements has to be considered. This is done by replacing variables by their current "value". We will use *term rewriting systems* to represent the state of variable assignments. The rules of these rewriting systems are of the form $(v \leftarrow p, W)$ with v being a variable, p being a non-weighted path expression, and $0 \leq W \leq 1$. The rule $(v \leftarrow p, W)$ indicates that v references the same object as p with probability W. Rules of the above form are called *assignment rules*. Subsequently, the following definitions are needed.

Definition A.2 (Auxiliary definitions)
Let R, R_1, R_2 be a set of assignment rules and let V be a set of variables. Then the sets $defs(R)$, $R - V$, and $R_1 \uplus R_2$ are defined as

$$\begin{aligned}
defs(R) &= \{\, v \mid \exists\, (v \leftarrow p, W) \in R \,\} \\
R - V &= \{\, (v \leftarrow p, W) \in R \mid v \notin V \,\} \\
R_1 \uplus R_2 &= \{\, (v \leftarrow p, W_1 + W_2) \mid (v \leftarrow p, W_1) \in R_1,\ (v \leftarrow p, W_2) \in R_2 \,\} \\
&\quad \cup \{\, (v \leftarrow p, W_1) \in R_1 \mid \nexists (v \leftarrow p, W_2) \in R_2 \,\} \\
&\quad \cup \{\, (v \leftarrow p, W_2) \in R_2 \mid \nexists (v \leftarrow p, W_1) \in R_1 \,\}
\end{aligned}$$

Let $p = v[w_0].A_1[w_1].\cdots.A_k[w_k]$ be a path expression, P a set of path expressions, R a set of assignment rules, and $u \in [0,1]$.

$$\begin{aligned}
p * u &= v[w_0 * u].A_1[w_1 * u].\cdots.A_k[w_k * u] \\
P * u &= \{\, p * u \mid p \in P \,\} \\
R * u &= \{\, (v \leftarrow p, W * u) \mid (v \leftarrow p, W) \in R \,\}
\end{aligned}$$

□

A set of assignment rules is applied to a set of path expressions (or to the path expressions of a second set of assignment rules) using the operator $\circ$ which is defined by Definition A.3.

Definition A.3 (Rewriting of path expressions)
Let P be a set of path expressions, R, R' two sets of assignment rules, and $0 \leq w \leq 1$.

$$\begin{aligned}
P \circ_w R' &= \{\, p' * W' \mid \exists\, p \in P : (p, R', w) \longrightarrow (p', W') \,\} \\
R \circ_w R' &= \{\, (v \leftarrow p', W * W') \mid \exists\, (v \leftarrow p, W) \in R : (p, R', w) \longrightarrow (p', W') \,\}
\end{aligned}$$

The predicate $(p, R', w) \longrightarrow (p', W')$ holds for some set R' of assignment rules, some path expressions p and p' and some w, $W' \in [0,1]$ iff

1. $p = v_1[w_0].A_1[w_1].\cdots.A_k[w_k]$, *there is some rule* $(v_1 \leftarrow v_2.B_1.\cdots.B_l, W') \in R'$, *and* $p' = v_2[0].B_1[0].\cdots.B_{l-1}[0].B_l[w_0].A_1[w_1].\cdots.A_k[w_k]$ *or*

2. *there is no rule in* R' *applicable to* p, $p = p'$, *and* $W' = w$.

w is the default weight if there is no rule applicable. We write $P \circ R'$ for $P \circ_1 R'$ and $R \circ R'$ for $R \circ_1 R'$. □

To compute for some statement S the set of assignment rules valid at the beginning of S — denoted by $in(S)$ — and the set of assignment rules valid at the end of S — denoted by $out(S)$ — we analyze the dataflow of the operation *op* similar to [ASU87, Chapter 10]. Before defining the sets $in(S)$ and $out(S)$ we introduce the sets $gen(S)$ and $kill(S)$ as follows: For some statement S the set $gen(S)$ contains rules modelling all assignments generated by S, and the set $kill(S)$ contains the identifiers of all variables whose former assignments were killed by S. If S assigns a new value to some variable v then $v \in kill(S)$ holds.

Note the difference between the sets *gen* and *out*: For some statement S, $gen(S)$ contains all assignments that reach the end of S independently of the assignments reaching the beginning of S, whereas $out(S)$ is the set of assignments that reach the end of S with respect to the assignments valid at the beginning of S. Similarly, the weight assigned to some rule $(v \leftarrow p, W) \in gen(S)$ denotes the probability that p is assigned to v when executing S, whereas the weight assigned to some rule $(v' \leftarrow p', W') \in out(S)$ denotes the probability of p' being assigned to v' if S is executed with probability w_S, i.e., during some invocation of the operation *op*.

The equations in Table 4 give an inductive definition of the sets *gen*, *kill*, *in*, and *out*. First the sets *gen* and *kill* are computed for all statements of *op* starting with the innermost statements, i.e., statements without nested substatements. To compute $gen(S)$ and $kill(S)$ for some statement S the sets *gen* and *kill* must already be computed for all substatements of S. Then, the sets *in* and *out* are computed top-down starting at the statement representing the body of *op* (for that statement the set *in* is empty). For all other statements S the set $in(S)$ is inherited from the statement preceding or directly enclosing S.

Subsequently, the dataflow equations of Table 4 are discussed separately for each kind of statement.

Empty statement, return statement. Neither the empty statement ϵ nor the **return** statement produce any assignment rules. Therefore, for some empty statement or **return** statement S the sets $gen(S)$ and $kill(S)$ are empty, and $out(S) = in(S)$.

Assignment. If S is an assignment statement $v := e$ the set $gen(S)$ contains one assignment rule $(v \leftarrow p, 1)$ for each $p \in \mathcal{P}(e) * 0$. The non-weighted path expressions $\mathcal{P}(e) * 0$ are used to rewrite path expressions extracted from subsequent statements. In these statements the evaluation of v does not induce the traversal of any path — that is why the weights of the path expressions are set to 0 in the corresponding assignment rules. The set $out(S)$ for some assignment statement S contains all assignment rules generated by S after being rewritten by the assignment rules valid at the beginning of S using the default weight w_S $(gen(S) \circ_{w_S} in(S))$, together with all assignment rules valid at the beginning of S but not killed by S $(in(S) - kill(S))$.

Statement S	*Data-flow equations for S*		
return e, ϵ	$gen(S)$	$=$	$\emptyset$
(ϵ *denotes*	$kill(S)$	$=$	$\emptyset$
the empty statement)	$out(S)$	$=$	$in(S)$
$v := e$	$gen(S)$	$=$	$\{ (v \leftarrow p, 1) \mid p \in \mathcal{P}(e) * 0 \}$
	$kill(S)$	$=$	$\{ v \}$
	$out(S)$	$=$	$(gen(S) \circ_{w_S} in(S)) \uplus (in(S) - kill(S))$
if e **then** S_1 **else** S_2	$gen(S)$	$=$	$gen(S_1) * prob(e) \ \uplus \ gen(S_2) * (1 - prob(e))$
			$\uplus \{ (v \leftarrow v, w_{S_1}) \mid v \in defs(gen(S_2)) \setminus defs(gen(S_1)) \}$
			$\uplus \{ (v \leftarrow v, w_{S_2}) \mid v \in defs(gen(S_1)) \setminus defs(gen(S_2)) \}$
	$kill(S)$	$=$	$kill(S_1) \cup kill(S_2)$
	$in(S_1)$	$=$	$in(S)$
	$in(S_2)$	$=$	$in(S)$
	$out(S)$	$=$	$out(S_1) \uplus out(S_2)$
			$\uplus \{ (v \leftarrow v, w_{S_1}) \mid v \in defs(gen(S_2)) \setminus defs(gen(S_1)) \}$
			$\uplus \{ (v \leftarrow v, w_{S_2}) \mid v \in defs(gen(S_1)) \setminus defs(gen(S_2)) \}$
while e **do** S_1	$gen(S)$	$=$	$gen^*(S_1)$
	$kill(S)$	$=$	$kill(S_1)$
	$in(S_1)$	$=$	$(gen(S) \circ_{w_S} in(S)) \uplus in(S)$
	$out(S)$	$=$	$out(S_1)$
$S_1; S_2$	$gen(S)$	$=$	$(gen(S_2) \circ gen(S_1)) \uplus (gen(S_1) - kill(S_2))$
	$kill(S)$	$=$	$kill(S_2) \cup kill(S_1)$
(S_2 *must not be a*	$in(S_1)$	$=$	$in(S)$
sequence of statements)	$in(S_2)$	$=$	$out(S_1)$
	$out(S)$	$=$	$out(S_2)$

Table 4: Data-flow equations for reaching and rewriting assignments

Conditional Statement. The set $gen(S)$ for a statement S : **if** e **then** S_1 **else** S_2 contains all assignment rules generated either by S_1 or by S_2, i.e., the set $gen(S_1) * prob(e) \cup gen(S_2) * (1 - prob(e))$. Further, *identity rules* of the form $v \leftarrow v$ are inserted into $gen(S)$ for all variables to which a new value is assigned in only one of the branches — because the use of the old values of v has to be re-weighted according to the branching probability. For example, consider the conditional statement

```
  S :  if g.X ≠ NULL
  S1:     then g := g.X;
[ S2:     else ;            ]
```

where the **else** branch is empty. Let the branching probability of S be 0.7. Clearly, the rules $(g \leftarrow g.X, 0.7)$ and $(g \leftarrow g, 0.3)$ have to be inserted into $gen(S)$ to reflect the state of variable assignments independent of the outcome of the condition $g.X \neq NULL$.

Contrary to [ASU87] we define the set $kill(S)$ as the union of $kill(S_1)$ and $kill(S_2)$ to compensate for the additional identity rules inserted into the set $gen(S)$. (In [ASU87] $kill(S) = kill(S_1) \cap kill(S_2)$.)

Sequence. A sequence of statements $S_1; S_2$ is analyzed from left to right according to the sequence of execution — here, S_2 must not be a sequence of statements. The set of

assignment rules generated by S_1; S_2 is given by all statements generated by S_2 rewritten by the assignment rules generated by S_1 ($gen(S_2) \circ gen(S_1)$) and all assignment rules generated by S_1 that are not killed by S_2 ($gen(S_1) - kill(S_2)$).

A.4 Loops

Our desire is that recursive assignments generated by the body of a loop are reflected by the extracted path expressions. For example, consider the loop statement S_2 of the operation YX^*move (also shown in Fig. 3):

S_2: **while** g.X $\neq$ NULL **do**
S_3: g := g.X;

The assignment statement S_3 generates the rule $gen(S_3) = \{ (g \leftarrow g.X, w_{S_3}) \}$. But which rules are generated by repeatedly executing S_3 inside the loop? Obviously, in each iteration the variable g is recursively defined using the value of g from the last iteration (or its initial value). The corresponding assignment rules are

$$\{ (g \leftarrow g, w_{S_3}),\ (g \leftarrow g.X, w_{S_3}),\ (g \leftarrow g.X.X, w_{S_3}),\ \ldots \}$$

which can be abbreviated by $\{ (g \leftarrow g(.X)^*, w_{S_3}) \}$ — using the "$*$"-notation known from regular expressions. We call this set the *loop closure* of the statement S_3 and denote it $gen^*(S_3)$. In the dataflow equations of Table 4 for some loop statement S : **while** e **do** S_1 the loop closure $gen^*(S_1)$ is used to define the set of assignment rules $gen(S)$ generated by the loop S.

To compute the loop closure $gen^*(S)$ for an arbitrary statement S we proceed in three steps:

1. From statement S we compute the set $gen(S)$ of assignment rules generated by S using the dataflow equations of Table 4.

2. Then, the rules of $gen(S)$ are transformed into a deterministic finite automaton (DFA), whose states represent the variables appearing in $gen(S)$ and whose transitions represent the rules of $gen(S)$. This automaton is denoted $\mathcal{A}(S)$.

3. Finally, the rules of the loop closure $gen^*(S)$ are defined by the regular language accepted by $\mathcal{A}(S)$.

The goal of transforming the set $gen(S)$ into a DFA $\mathcal{A}(S)$ is to yield a DFA for a language representing all path expressions that may be traversed by the sequence

$$S;\ S;\ S;\ \ldots$$

Obviously, the transition function of $\mathcal{A}(S)$ must directly reflect the rules of $gen(S)$. For reasons of simplicity we do not consider the weights of the rules in $gen(S)$ when constructing the automaton $\mathcal{A}(S)$.[12] It is convenient to take the variables occuring in $gen(S)$

[12] Considering the weights of the rules would mean to get a probabilistic automaton [Sal69]. We plan to extend the decapsulation process described in this work to include also weights of assignment rules inside the body of loop statements. By ommitting these weights, the decapsulation process described in this work cannot capture the effect of branching statements inside loop statements.

as states and the rules of $gen(S)$ as transitions in $\mathcal{A}(S)$. Some rule $v_2 \leftarrow v_1.Q \in gen(S)$[13] corresponds to a transition from state v_1 into state v_2 with the input Q. To specify the domain of the transition function in the subsequent definition of $\mathcal{A}(S)$ we abbreviate the set of all attribute chains Q occuring in the rules of $gen(S)$ by $\mathbf{E}$. (The DFA $\mathcal{A}(S)$ will be defined over the alphabet $\mathbf{E}$.)

The variables appearing on the right hand side of some rule in $gen(S)$, e.g., the variable v_1 in the above example, are called the *initial variables* of S as they represent the input to the execution of S; S; S; ...; all paths traversed by S; S; S; ... start at one of the initial variables of S. Because of that, the initial variables of S are taken as initial states of $\mathcal{A}(S)$.

Definition A.4 (Computation of $\mathcal{A}(S)$)
Let S be some statement generating the assignment rules $gen(S)$. The DFA $\mathcal{A}(S) = (\boldsymbol{S}, \boldsymbol{S}_0, \delta)$ is defined by the following condition:

$$v_1 \in \boldsymbol{S}_0,\ v_2 \in \boldsymbol{S},\ \delta(v_1, Q) = v_2 \quad \textit{iff} \quad v_2 \leftarrow v_1.Q \in gen(S)$$

where $\boldsymbol{S}$ is the set of states, $\boldsymbol{S}_0 \subseteq \boldsymbol{S}$ is the set of initial states and $\delta : \boldsymbol{S} \times \boldsymbol{E} \rightarrow \boldsymbol{S}$ is the transition function of $\mathcal{A}(S)$. □

The language that is represented by some initial state $v_1 \in \mathbf{S}_0$ and some state $v_2 \in \mathbf{S}$ in the DFA $\mathcal{A}(S)$ is

$$L(\mathcal{A}(S), v_1, v_2) \quad = \quad \{\, Q_1 Q_2 \ldots Q_k \mid \delta(v_1, Q_1 Q_2 \ldots Q_k) = v_2 \,\}$$

Here, the domain of the transition function δ is extended to $\mathbf{S} \times \mathbf{E}^*$ in the canonical way.

Now we must show that $\mathcal{A}(S)$ accepts exactly those words representing assignments generated by the sequence S; S; S; ...

Lemma A.1 *The variable v_2 may be assigned the value of $v_1.Q_1.Q_2.\cdots.Q_k$ during the execution of S; S; S; ... iff $v_1 \in \boldsymbol{S}_0$, $v_2 \in \boldsymbol{S}$, and $Q_1 Q_2 \ldots Q_k \in L(\mathcal{A}(S), v_1, v_2)$ hold.*

Proof The following equivalences hold according to the definition of $\mathcal{A}(S)$:

$$\begin{aligned}
&& Q_1 Q_2 \ldots Q_k &\in L(\mathcal{A}(S), v_1, v_2) \\
&\iff & \delta(v_1, Q_1 Q_2 \ldots Q_k) &= v_2 \\
&\iff & \delta(\delta(\ldots(\delta(v_1, Q_1), \ldots), Q_{k-1}), Q_k) &= v_2 \\
\text{(P2)} \quad &\iff & \exists\, v^{(1)}, v^{(2)}, \ldots, v^{(k)}, v^{(k+1)} &\in \mathbf{S} : \\
& & v^{(1)} = v_1 \ \wedge\ v^{(k+1)} &= v_2 \ \wedge\ v^{(i+1)} \leftarrow v^{(i)}.Q_i \in gen(S) \qquad (1 \le i \le k)
\end{aligned}$$

By considering the equivalence (P1) $\iff$ (P2) and the equation

$$gen(\underbrace{S;\ S;\ \ldots;\ S}_{k\ times}) \quad = \quad \underbrace{gen(S) \circ gen(S) \circ \ldots \circ gen(S)}_{k\ times} \tag{1}$$

the result of the lemma is easily being seen. Equation (1) is derived from the dataflow equations of Table 4 for sequences and the fact that $gen(S) - kill(S) = \emptyset$ holds for all statements. ■

Now we can define the loop closure $gen^*(S)$ of some statement S in terms of the languages represented by the DFA $\mathcal{A}(S)$:

[13] Subsequently, let Q denote a sequence of attributes.

Definition A.5 (Computation of $gen^*(S)$)

$$gen^*(S) \;=\; \{\, (v_2 \leftarrow v_1.\alpha_{1,2}, w_S) \;:\; |\alpha_{1,2}| = L(\mathcal{A}(S), v_1, v_2) \,\}$$

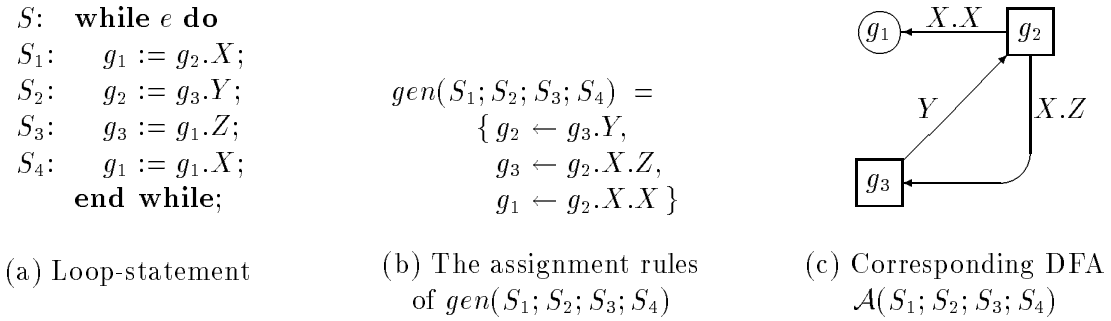
where $\alpha_{1,2}$ is a regular expression over the alphabet $\boldsymbol{E}$ and the auxiliary alphabet $\{\,|, *, (,)\,\}$, and $|\alpha_{1,2}|$ denotes the language represented by $\alpha_{1,2}$. □

An $O(n^3)$ algorithm to compute a regular expression α to some DFA $\mathcal{A}$ accepting the regular language L with $L = |\alpha|$ can be found in [Sal69] (where n is the number of states in $\mathcal{A}$).

Example: Consider the body of the **while**-statement shown in Fig. 13 (a). The assignment rules generated by $S_1; S_2; S_3; S_4$ are $\{\, g_1 \leftarrow g_2.X.X,\ g_2 \leftarrow g_3.Y,\ g_3 \leftarrow g_2.X.Z \,\}$. The DFA $\mathcal{A}(S_1; S_2; S_3; S_4)$ of the loop's body is shown in Fig. 13 (c); the initial states of this DFA are marked by squares, the remaining states by circles. The set $gen(S) = gen^*(S_1;\ S_2;\ S_3;\ S_4)$ is given by

$$\left\{\begin{array}{ll} g_1 \leftarrow (g_2(.X.Z.Y)^*.X.X & \mid g_3(.Y.X.Z)^*.Y.X.X) \\ g_2 \leftarrow (g_2(.X.Z.Y)^* & \mid g_3(.Y.X.Z)^*.Y) \\ g_3 \leftarrow (g_2(.X.Z.Y)^*.X.Z & \mid g_3(.Y.X.Z)^*) \end{array}\right\}$$

◇



(a) Loop-statement

(b) The assignment rules of $gen(S_1; S_2; S_3; S_4)$

(c) Corresponding DFA $\mathcal{A}(S_1; S_2; S_3; S_4)$

Figure 13: Computation of the loop closure of a sequence of assignments

A.5 Collecting All Path Expressions of an Operation

To compute the set $paths(op)$ containing all weighted path expressions traversed by the operation op we first define for each statement S of op the set $paths(S)$ containing all paths being traversed by S (or any of its substatements). Then $paths(op)$ equals $paths(S_0)$ where S_0 is the body of op. The path expressions occurring in some statement S are rewritten by the rules of $in(S)$ and multiplied by the weight $w_S * w_{op}$ before they are inserted into $paths(S)$. If S is a loop statement, $in(S) \cup out(S)$ is used for rewriting the path expressions of S because also assignments generated by the execution of the loop's body may be valid during the next iteration of the loop. The equations of Table 5 define the set $paths(S)$ for an arbitrary statement S.

Statement S	*Equations for* $paths(S)$
ϵ	$paths(S) = \emptyset$
$v := e$	$paths(S) = \{ p * w_{op} \mid p \in \mathcal{P}(e) \circ_{w_S} in(S) \}$
return e	$paths(S) = \{ p * w_{op} \mid p \in \mathcal{P}(e) \circ_{w_S} in(S) \}$
if e **then** S_1 **else** S_2	$paths(S) = \{ p * w_{op} \mid p \in \mathcal{P}(e) \circ_{w_S} in(S) \} \uplus paths(S_1) \uplus paths(S_2)$
while e **do** S_1	$paths(S) = \{ p * w_{op} \mid p \in \mathcal{P}(e) \circ_{w_S} (in(S) \cup out(S)) \} \uplus paths(S_1)$
$S_1; S_2$	$paths(S) = paths(S_1) \uplus paths(S_2)$

Table 5: Definition of the set $paths(S)$

B SUN-Benchmark

B.1 Definition of PartType

```
persistent type PartType is
  body [
          oid : int;
          to_connect : ConnectionListType;
          from_connect : ConnectionListType;
        ]

  operations

    declare PartType : int → void;
    declare access : → void;
    declare traversal_fwd : int → void;
    declare traversal_bwd : int → void;

  implementation

    define PartType(i) is
      begin
        self.oid := i;
        self.to_connect.create.persistent;
        self.from_connect.create.persistent;
        self.persistent;
    end define PartType;

    define access is
    var oid : int;
    begin
      oid := self.oid;
    end define access;

    define traversal_fwd(depth) is
    var ii : int;
        p : PartType;
        pl : PartListType;
        dl : IntListType;
```

```
    begin
      pl.create.insert(self);
      dl.create.insert(depth);
      while (not pl.is_empty) begin
        depth := dl.remove_last - 1;
        p := pl.remove_last;
        if (depth ≠ 0) begin
          p.access;
          ii := 0;
          while (ii < p.to_connect.length) begin
            ii := ii + 1;
            dl.append(depth);
            pl.append(p.to_connect.n_th(ii).to);
          end while;
        end if;
      end while;
    end Traversal_fwd;

    define traversal_bwd(depth) is
    var ii : int;
        p : PartType;
        pl : PartListType;
        dl : IntListType;
    begin
      pl.create.insert(self);
      dl.create.insert(depth);
      while (not pl.is_empty) begin
        depth := dl.remove_last - 1;
        p := pl.remove_last;
        if (depth ≠ 0) begin
          p.access;
          ii := 0;
          while (ii < p.from_connect.length) begin
            ii := ii + 1;
            dl.append(depth);
            pl.append(p.from_connect.n_th(ii).from);
          end while;
        end if;
      end while;
    end Traversal_bwd;

end type PartType;
```

B.2 Definition of ConnectionType

```
persistent type ConnectionType is
  body [
          oid : int;
          from, to : PartType;
```

```
        ]

  operations

    declare ConnectionType : PartType, PartType → void;

  implementation

    define ConnectionType(from_part, to_part) is
      begin
        self.from := from_part;
        self.to := to_part;
        self.oid := 0;
        self.persistent;
    end define ConnectionType;

end type ConnectionType;
```

B.3 Creation of Database

```
persistent type PartListType is < PartType >;
persistent type ConnectionListType is < ConnectionType >;
persistent type IntListType is < int >;
persistent var all_part_objects : PartListType;
persistent var number_of_parts : int;
persistent var number_of_connections : int;

declare create_db : int,int → void;


define create_db(parts, connections) is
var ii,jj : int;
    p,q : PartType;
    c : ConnectionType;
    index : int;
begin
  for (ii := 0; ii < parts; ii := ii + 1) begin
    p.create(ii);
    all_part_objects.insert(ii) := p;
  end for ;
  for (ii := 0; ii < parts; ii := ii + 1) begin
    p := all_part_objects.n_th(ii);
      for (jj := 0; jj < connections; jj := jj + 1) begin
        if (rand_float > 0.10) begin
          index := ii + (rand_int % (parts / 100)) − parts / 200;
          if (index < (parts / 200)) begin
            index := index + parts / 200;
          end if ;
          if (index > (parts − parts / 200)) begin
```

```
            index := index - parts / 200;
          end if ;
        end if ;
        else begin
          index := rand_int % number_of_parts;
        end else ;
        q := all_part_objects.n_th(index);
        c.create(p,q);
        p.to_connect.insert(jj) := c;
        q.from_connect.insert(q.from_connect.length) := c;
      end for ;
    end for ;
  end define create_db;
```

B.4 The Benchmark

```
declare benchmark : int,int,int,float,float,float → void;

define benchmark(ops,n_look,n_trav,P_lookup,P_traversal_fwd,P_traversal_bwd) is
var p : part_type;
    r : float;
    ii,jj : int;
begin
  for (ii := 0; ii < ops; ii := ii + 1) begin
    r := rand_float;
  if (r ≤ P_lookup) begin
    for (jj := 0; jj < n_look; jj := jj + 1) begin
      p := all_part_objects.n_th(rand_int % number_of_parts);
      p.access;
    end for ;
    end if ;
    else begin
        if ((P_lookup < r) and
          (r ≤ (P_lookup + P_traversal_fwd))) begin
          p := all_part_objects.n_th(rand_int % number_of_parts);
          p.traversal_fwd(n_trav);
        end if ;
      else begin
        if ((P_lookup + P_traversal_fwd < r) and
          (r ≤ (P_lookup + P_traversal_fwd + P_traversal_bwd))) begin
          p := all_part_objects.n_th(rand_int % number_of_parts);
          p.traversal_bwd(n_trav);
        end if ;
      end else ;
    end else ;
  end for ;
end define benchmark;
```

B.5 Main program

```
declare main : → void;

define main is
var ops : int;
    n_look,n_trav : int;
    P_lookup : float;
    P_traversal_fwd : float;
    P_traversal_bwd : float;
begin
  all_part_objects.create;

  print("Enter parts : ");
  number_of_parts := scanInt;
  print("Enter connections : ");
  number_of_connections := scanInt;
  print("Enter ops : ");
  ops := scanInt;
  print("Enter n_look : ");
  n_look := scanInt;
  print("Enter n_trav : ");
  n_trav := scanInt;
  print("Enter P_lookup : ");
  P_lookup := scanReal;
  print("Enter P_traversal_fwd : ");
  P_traversal_fwd := scanReal;
  print("Enter P_traversal_bwd : ");
  P_traversal_bwd := scanReal;

  print("\nparts : "); print(number_of_parts);
  print("\nconnections : "); print(number_of_connections);
  print("\nops : "); print(ops);
  print("\nn_look : "); print(n_look);
  print("\nn_trav : "); print(n_trav);
  print("\nP_lookup : "); print(P_lookup);
  print("\nP_traversal_fwd : "); print(P_traversal_fwd);
  print("\nP_traversal_bwd : "); print(P_traversal_bwd);
  print("\n");

  print("\nCreating database ...\n");
  create_db(number_of_parts,number_of_connections);
  all_part_objects.persistent;
  number_of_parts.persistent;
  number_of_connections.persistent;

  print("\nStarting benchmark ...\n");
  benchmark(ops,n_look,n_trav,P_lookup,P_traversal_fwd, P_traversal_bwd);
  print("\n\n");
end define main;
```

C GRID-Benchmark

C.1 Definition of GridType

```
persistent type GridType is
  body [ !! total size is 300 Byte, data entries omitted
           oid : int;
           x,y,x : GridType;
           x_inv,y_inv,x_inv : GridType;
           random : GridListType;
           data : ListType;
         ]

  operations

    declare GridType : int → void;
    declare xmove : int → GridType;
    declare ymove : int → GridType;
    declare zmove : int → GridType;
    declare ximove : int → GridType;
    declare yimove : int → GridType;
    declare zimove : int → GridType;
    declare YX*move : int → void;
    declare Traversal : int → void;
    declare ShortPaths : → void;
    declare LinearPath : → void;
    declare Y*ZZ_inv move : → void;
    declare next*move : → void;
    declare access_grid : → void;

  implementation

    define GridType(i) is
      begin
        self.oid := i;
        self.x := null(self.x);
        self.y := null(self.y);
        self.z := null(self.z);
        self.x_inv := null(self.x_inv);
        self.y_inv := null(self.y_inv);
        self.z_inv := null(self.z_inv);
        self.random := null(self.random);
        self.data := null(self.data);
        self.persistent;
    end define GridType;

    define access_grid is
    var oid : int;
    begin
      oid := self.oid;
```

```
end access_grid;

define xmove(n) is
var g : GridType;
begin
  g := self;
  while ((n := n−1) ≥ 0) begin
    if (not is_null(g.x)) begin
      g := g.x;
    end if ;
  end while ;
  return g;
end xmove;

define ymove(n) is
var g : GridType;
begin
  g := self;
  while ((n := n−1) ≥ 0) begin
    if (not is_null(g.y)) begin
      g := g.y;
    end if ;
  end while ;
  return g;
end ymove;

define zmove(n) is
var g : GridType;
begin
  g := self;
  while ((n := n−1) ≥ 0) begin
    if (not is_null(g.z)) begin
      g := g.z;
    end if ;
  end while ;
  return g;
end zmove;

define ximove(n) is
var g : GridType;
begin
  g := self;
  while ((n := n−1) ≥ 0) begin
    if (not is_null(g.x_inv)) begin
      g := g.x_inv;
    end if ;
  end while ;
  return g;
end ximove;
```

```
define yimove(n) is
var g : GridType;
begin
  g := self;
  while ((n := n−1) ≥ 0) begin
    if (not is_null(g.y_inv)) begin
      g := g.y_inv;
    end if ;
  end while ;
  return g;
end yimove;

define zimove(n) is
var g : GridType;
begin
  g := self;
  while ((n := n−1) ≥ 0) begin
    if (not is_null(g.z_inv)) begin
      g := g.z_inv;
    end if ;
  end while ;
  return g;
end zimove;

define ShortPaths is
begin
  self.xmove(1).access_grid;
  self.ymove(1).access_grid;
  self.zmove(1).access_grid;
  self.ximove(1).access_grid;
  self.yimove(1).access_grid;
  self.zimove(1).access_grid;
end ShortPaths;

define YX*move(x) is
begin
  self.ymove(1).xmove(x).access_grid;
end YX*move;

define Traversal(depth) is
var ii : int;
    g : GridType;
    gl : GridListType;
    dl : IntListType;
begin
  gl.create.insert(self);
  dl.create.insert(depth);
  while (not gl.is_empty) begin
    depth := dl.remove_last − 1;
    g := gl.remove_last;
```

```
        if (depth ≠ 0) begin
          if (not is_null(self.random))) begin
            ii := 0;
            while (ii < g.random.length) begin
              ii := ii + 1;
              dl.append(depth);
              gl.append(g.random.n_th(ii));
            end while;
          end if;
        end if;
      else begin
        g.data.access_list;
      end else ;
      end while;
    end Traversal;

    define LinearPath is
    begin
      self.xmove(1).ymove(1).zmove(1).yimove(1).ximove(1).data.lmove(1).access_list;
    end LinearPath;

    define Y*ZZ_inv move is
    var g : GridType;
    begin
      g := self;
      while (not is_null(g.y)) begin
        g.zmove(1).access_grid;
        g.zimove(1).access_grid;
        g := g.ymove(1);
      end while ;
      g.access_grid;
    end Y*ZZ_inv move;

    define next*move is
    var l : ListType;
    begin
      l := self.data;
      while (not is_null(l.next)) begin
        l:= l.lmove(1);
      end while ;
      l.access_list;
    end next*move;

end type GridType;
```

C.2 Definition of ListType

```
persistent type ListType is
  body [ !! total size is 300 Byte, data entries omitted
```

```
        oid : int;
        next : ListType;
      ]

  operations

    declare ListType : int → void;
    declare create_list : int → void;
    declare lmove : int → ListType;
    declare access_list : → void;

  implementation

    define ListType(i) is
      begin
        self.oid := i;
        self.next := null(self.next);
        self.persistent;
    end define ListType;

    define create_list is
    var l : ListType;
    begin
      l := self;
      while ((i := i−1) ≥ 0) begin
        l.next.create(l.oid);
        l := l.next;
      end while ;
    end create_list;

    define lmove(n) is
    var l : ListType;
    begin
      l:= self;
      while ((n := n−1) ≥ 0) begin
        if (not is_null(g.x)) begin
          l := l.next;
        end if ;
      end while ;
      return l;
    end lmove;

    define access_list is
    var oid : int;
    begin
      oid := self.oid;
    end access_list;

end type ListType;
```

C.3 Creation of Database

```
persistent type GridListType is < grid_type >;

declare create_grid_node : → GridType;
declare create_x_grid : int → GridType;
declare create_xy_grid : int,int → GridType;
declare create_xyz_grid : int,int,int → GridType;
declare create_grid_db : int,int,int → void;

persistent var all_grid_objects : GridListType;
persistent var root_grid_objects : GridListType;
persistent var number_of_objects : int;
persistent var number_of_grids : int;
persistent var number_of_lists : int;
persistent var x_dim,y_dim,z_dim : int;

define create_grid_node is
var o : GridType;
begin
  o.create(number_of_objects);
  all_grid_objects.insert(number_of_objects) := o;
  number_of_objects := number_of_objects + 1;
  return o;
end create_grid_node;

define create_x_grid(no_x) is
var root,o1,o2 : GridType;
    xx : int;
begin
  o1 := create_grid_node;
  root := o1;
  for (xx := 2; xx ≤ no_x; xx:= xx + 1) begin
    o2 := create_grid_node;
    o1.x := o2;
    o2.x_inv := o1;
    o1 := o2;
  end for ;
  return root;
end define create_x_grid;

define create_xy_grid(no_x,no_y) is
var root,o1,o2 : GridType;
    y_hold : GridType;
    xx,yy : int;
begin
  o1 := create_x_grid(no_x);
  root := o1;
```

```
  for (yy := 2; yy ≤ no_y; yy:= yy + 1) begin
    o2 := create_x_grid(no_x);
    y_hold := o2;
    for (xx := 1; xx ≤ no_x; xx:= xx + 1) begin
      o1.y := o2;
      o2.y_inv := o1;
      o1 := o1.x;
      o2 := o2.x;
    end for ;
    o1 := y_hold;
  end for ;
  return root;
end define create_xy_grid;

define create_xyz_grid(no_x,no_y,no_z) is
var root,o1,o2 : GridType;
    y_hold1,y_hold2,z_hold : GridType;
    xx,yy,zz : int;
begin
  o1 := create_xy_grid(no_x,no_y);
  root := o1;
  for (zz := 2; zz ≤ no_z; zz:= zz + 1) begin
    o2 := create_xy_grid(no_x,no_y);
    z_hold := o2;
    for (yy := 1; yy ≤ no_y; yy:= yy + 1) begin
      y_hold1 := o1;
      y_hold2 := o2;
      for (xx := 1; xx ≤ no_x; xx:= xx + 1) begin
        o1.z := o2;
        o2.z_inv := o1;
        o1 := o1.x;
        o2 := o2.x;
      end for ;
      o1 := y_hold1.y;
      o2 := y_hold2.y;
    end for ;
    o1 := z_hold;
  end for ;
  return root;
end define create_xyz_grid;

define create_grid_db(no_x,no_y,no_z) is
var candidate : GridType;
    r : float;
    ii,jj : int;
    oid : int;
    nr,count : int;
begin
  for (ii := 0; ii < number_of_grids; ii := ii + 1) begin
    root_grid_objects.insert(ii) := create_xyz_grid(no_x,no_y,no_z);
```

```
  end for ;
  /* create random */
  for (jj := 0; jj < number_of_objects; jj := jj + 1) begin
    candidate := all_grid_objects.n_th(jj);
    /* simple implementation of a histd-function */
    r := rand_float;
    if (r ≤ 0.20) begin
      nr := 0;
    end if ;
    if ((r > 0.20) and (r ≤ 0.40)) begin
      nr := 1;
    end if ;
    if ((r > 0.40) and (r ≤ 0.60)) begin
      nr := 2;
    end if ;
    if ((r > 0.60) and (r ≤ 0.80)) begin
      nr := 3;
    end if ;
    if (r > 0.80) begin
      nr := 4;
    end if ;
    if (nr > 0) begin
      candidate.random.create.persistent;
      for (ii := 0; ii < nr; ii := ii + 1) begin
        oid := rand_int % number_of_objects;
        candidate.random.insert(ii) := all_grid_objects.n_th(oid);
      end for ;
    end if ;
    candidate.data.create(jj);
    candidate.data.create_list(number_of_lists − 1);
  end for ;
end define create_grid_db;
```

C.4 The Benchmark

```
declare create_access_list : GridListType,int,float → void;
declare benchmark : GridListType,int,float,float, float, float,float,float, float → void;

define create_access_list(access,ops,P_root) is
var ii : int;
begin
  for (ii := 0; ii < ops; ii := ii + 1) begin
    if (rand_float ≤ P_root) begin
      access.insert(ii) := root_grid_objects.n_th(rand_int % number_of_grids);
    end if ;
    else begin
      access.insert(ii) := all_grid_objects.n_th(rand_int % number_of_objects);
    end else ;
  end for ;
```

```
end define create_access_list;

define benchmark(access,ops,P_gobble,P_jump,P_run,P_ramble,P_visit,P_scan) is
var g : GridType;
    ii : int;
    r : float;
    x,y,z : int;
begin
  x := x_dim;
  y := y_dim;
  z := z_dim;
  for (ii := 0; ii < ops; ii := ii + 1) begin
    g := access.n_th(ii);
    r := rand_float;
    if (r ≤ P_ramble) begin
      g.LinearPath;
    end if ;
    else begin
      if ((P_ramble < r) and (r ≤ (P_ramble + P_gobble))) begin
        g.ShortPaths;
      end if ;
      else begin
        if (((P_ramble + P_gobble) < r) and
          (r ≤ (P_ramble + P_gobble + P_jump))) begin
          g.Traversal(2);
        end if ;
        else begin
          if (((P_ramble + P_gobble + P_jump) < r) and
            (r ≤ (P_ramble + P_gobble + P_jump + P_run))) begin
            g.YX*move(x);
          end if ;
          else begin
            if (((P_ramble + P_gobble + P_jump + P_run) < r) and
              (r ≤ (P_ramble + P_gobble + P_jump + P_run + P_scan))) begin
              g.next*move;
            end if ;
            else begin
              g.Y*ZZ_inv move;
            end else ;
          end else ;
        end else ;
      end else ;
    end for ;
end define benchmark;
```

C.5 Main program

```
declare main : → void;
```

```
define main is
var access_list : GridListType;
ops : int;
rd : int;
P_gobble, P_run, P_jump, P_ramble, P_visit, P_scan, P_root : float;
ii : int;
begin print("\n");

access_list.create;

root_grid_objects.create;
all_grid_objects.create;
number_of_objects := 0;

print("Enter grids : "); number_of_grids := scanInt;
print("Enter lists : "); number_of_lists := scanInt;
print("Enter x dimension : "); x_dim := scanInt;
print("Enter y dimension : "); y_dim := scanInt;
print("Enter z dimension : "); z_dim := scanInt;
print("Enter operations : "); ops := scanInt;
print("Enter srand : "); rd := scanInt;
print("Enter P_root : "); P_root := scanReal;
print("Enter P_gobble : "); P_gobble := scanReal;
print("Enter P_jump : "); P_jump := scanReal;
print("Enter P_run : "); P_run := scanReal;
print("Enter P_ramble : "); P_ramble := scanReal;
print("Enter P_visit : "); P_visit := scanReal;
print("Enter P_scan : "); P_scan := scanReal;
print("\n");

print("\ngrids : "); print(number_of_grids);
print("\nlists : "); print(number_of_lists);
print("\nx dim : "); print(x_dim);
print("\ny dim : "); print(y_dim);
print("\nz dim : "); print(z_dim);
print("\nops : "); print(ops);
print("\nsrand : "); print(rd);
print("\nP_root : "); print(P_root);
print("\nP_gobble: "); print(P_gobble);
print("\nP_jump : "); print(P_jump);
print("\nP_run : "); print(P_run);
print("\nP_ramble: "); print(P_ramble);
print("\nP_visit : "); print(P_visit);
print("\nP_scan : "); print(P_scan);

print("\n\nCreating database ...\n");
create_db(x_dim,y_dim,z_dim);

set_rand(rd);
```

```
print("\nCreating access list ...\n");
create_access_list(access_list,ops,P_root);

all_grid_objects.persistent;
root_grid_objects.persistent;
number_of_objects.persistent;
number_of_grids.persistent;
number_of_lists.persistent;
x_dim.persistent;
y_dim.persistent;
z_dim.persistent;

print("\nStarting benchmark ...\n");
benchmark(access_list,ops,P_gobble,P_jump,P_run, P_ramble,P_visit,P_scan);

end define main;
```