On the Performance of Object Clustering Techniques

Manolis M. Tsangaris
Jeffrey F. Naughton

Technical Report #1090

June 1992

On the Performance of Object Clustering Techniques

Manolis M. Tsangaris and Jeffrey F. Naughton
{mt,naughton}@cs.wisc.edu
Department of Computer Sciences
University of Wisconsin–Madison

Technical Report # 1090 - 1992

May,1 1992

Abstract

We investigate the performance of some of the best-known object clustering algorithms on four different workloads based upon the Tektronix benchmark. For all four workloads, stochastic clustering gave the best performance for a variety of performance metrics. Since stochastic clustering is computationally expensive, it is interesting that for every workload there was at least one cheaper clustering algorithm that matched or almost matched stochastic clustering. Unfortunately, for each workload, the algorithm that approximated stochastic clustering was different. Our experiments also demonstrated that even when the workload and object graph are fixed, the choice of the clustering algorithm depends upon the goals of the system. For example, if the goal is to perform well on traversals of small portions of the database starting with a cold cache, the important metric is the per-traversal expansion factor, and a well-chosen placement tree will be nearly optimal; if the goal is to achieve a high steady-state performance with a reasonably large cache, the appropriate metric is the number of pages to which the clustering algorithm maps the active portion of the database. For this metric, the PRP clustering algorithm, which only uses access probabilities achieves nearly optimal performance.

---

* This work was supported by NSF grant number IRI–9157357, and partially by a NATO Fellowship. A shorter version of this report appears in the proceedings of the International Conference on Management of Data - SIGMOD 1992, San Diego, CA.

1 Introduction

In recent years a number of clustering algorithms for object-oriented databases have appeared in the literature. These algorithms attempt to improve the performance of object-oriented database systems by placing on the same page related sets of objects, thus attempting to avoid the performance penalty of one disk I/O per object access. For the most part, these algorithms have each been presented in isolation, with some experimental data illustrating how these algorithms perform when compared to no clustering or "random" clustering. In this paper we investigate the relative performances of a number of these clustering algorithms on four different workloads based upon the Tektronix [And90] benchmark. Our results apply directly to object bases with similar object structure, properties, and usage as in the Tektronix Benchmark.

The algorithms we compared were BFS, DFS, and WDFS [Sta84], Placement Trees [BD90], Cactis [DK90], PRP [YW73] and stochastic clustering [TN91]. Of these algorithms, BFS and DFS depend only on the structure of the object graph, while the other algorithms depend in addition on a information gleaned from a training trace representative of some workload. In more detail, these algorithms are "trained" by letting them gather statistics from a trace representative of the workload; they then use these statistics and the structure of the object graph to decide upon a good clustering. To evaluate the quality of the resulting clustering, one runs another trace, different from the training trace yet still representative of the given workload, and gathers statistics about page fault rates and numbers of pages touched.

We found that for all four workloads tested, stochastic clustering gave uniformly the best results by a variety of performance metrics. Stochastic clustering works by postulating that the workload is generated by some stochastic process, then gathering statistics from the training trace to estimate the parameters of this hypothetical stochastic process, and finally mapping objects to pages so as to minimize the probability that a pair of consecutive object accesses in the reference stream crosses a page boundary. The results of these experiments were an important confirmation of the utility of the ideas behind stochastic clustering, since before performing these experiments, it was not obvious that stochastic clustering would perform this well. In particular, a number of the assumptions made by stochastic clustering are only approximately true – references in the workloads in the Tektronix benchmark are not generated by stochastic processes, and it was not immediately obvious that minimizing the probability that consecutive object accesses cross page boundaries maximizes performance.

Stochastic clustering, while highly effective in these experiments, is prohibitively computationally expensive to be applied directly in many situations. In view of this fact, it is important to find lower-cost algorithms that approximate the performance of stochastic clustering. Our results were encouraging in that for each workload tested, there was at least one computationally less expensive algorithm that approximated the performance of stochastic clustering. However, unfortunately the algorithm that approximated stochastic clustering was different for each workload. This suggests that a practical clustering strategy may be a set of clustering strategies, each appropriate for a different class of workload, rather than a single monolithic strategy.

Another fact that became clear in our experiments is that even if you fix the object base and the workload, which clustering algorithm is best depends in an important way on the performance goals of the system. For example, if the goal of the system is to perform well on traversals of a small portion of the database starting with a cold cache, the important metric is the ratio of the number of pages a traversal touches to the smallest number of pages in which the objects touched by the traversal could be stored. On the other hand, if the goal of the system is to perform well in steady state with a fairly large cache, the important metric is to how many pages the clustering algorithm maps the active portion of the database. An algorithm that performs well by one metric will not necessarily perform well by the second. A particularly interesting result is that for the large cache, steady state case, the PRP (Probability Ranking Partitioning) algorithm is nearly optimal. This is surprising since the PRP algorithm makes no use of the object graph at all, clustering solely on the basis of statistics gathered from the training trace.

A final result of this study is that like high performance race horses, high performance clustering algorithms can be temperamental. That is, the "bad" clustering algorithms are relatively insensitive to differences between the training trace and the testing trace, whereas the "best" clustering algorithms show dramatic drops in performance when the testing workload contains elements of a workload that was not included in the training trace. This suggests that the highest performing algorithms may not be desirable if the reference patterns in the system vary markedly over small intervals of time.

The remainder of this paper is structured as follows: in Section 2 we describe our model of an OODBMS and the clustering problem that arises in this model. Section 3 describes our simulation environment, which we used to run the experiments. In Section 4 we present the results of our experiments, and we conclude in Section 5.

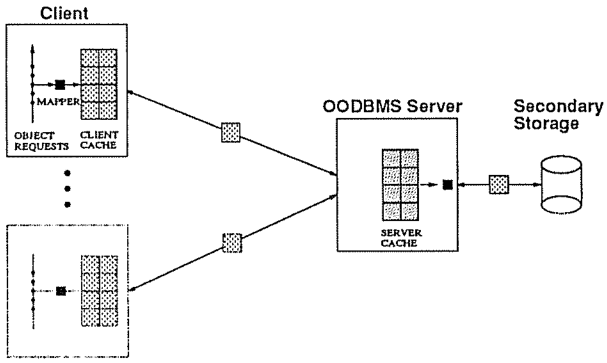

Figure 1: A simplified OODBMS Architecture

2 Clustering in OODBMS

A general Object Oriented Data Base Management System architecture is shown in Figure 1. The system consists of a) a data base server serving one or more clients, b) the system's secondary memory only accessible to the server, and c) the clients, programs written in some object oriented programming language the OODBMS supports. The server supports some object abstraction for its clients, and every object in the system is uniquely identified by its Object Identifier (OID), a permanently assigned number. The object state representation is stored in the system secondary memory consisting of fixed uniform size pages. Each object is assigned to only one page through the *Clustering Mapping*. Both the server and the clients have a mechanism to find the page of an object given its OID.

Conceptually, objects can be considered vertices of a directed and possibly cyclic graph called the Object Graph. The directed edges of the object graph are just the object to object pointers. Typically, the client programs access objects *sequentially* during their execution (i.e. one at the time), by dereferencing object pointers and thus "walking on the object graph". The program requires access to the object representation, and therefore, the client process is suspended if the object is not available. The process is resumed when the missing object is obtained from the server. In the *Paged OODBMS Architectures* the server will not just return a single object, but will return all objects mapped to the same page as the requested object. After the client receives the page containing the requested object, the client resumes execution and can access all of the objects brought in with that page.

Because of the high overhead associated with servicing requests, most OODBMS's add caches on the client and on the server. The client cache reduces server requests, and the server cache reduces disk accesses. It is important to note that a page of the client cache contains the same objects as the corresponding page on the server. Redistributing objects to pages is usually avoided, because it can cause performance problems when writing back modified objects, and because it may force the client to acquire many fine granularity object locks instead of fewer coarse page locks.

The choice of placing objects to pages affects the performance of the system in terms of system load, server overhead, concurrency control, and recovery. For example, 100 different objects can be placed into as many as 100 different pages, or as few as 5 pages of $4k$ bytes each (assuming a 200 byte average object size). The latter clustering mapping will require 1/20 as many client server interactions, 1/20 as much client cache memory, 1/20 as many pages that might need logging during updates, 1/20 as many page locks that have be obtained, and smaller probability of conflict between different transactions. In this paper we will concentrate on the effect of clustering on memory utilization and system load.

2.1 Clustering Performance Measures

As the previous discussion motivates, the "packing capability" of clustering algorithms is a simple way to measure their performance. If we view objects as records and client requests as queries of records, the *Expansion Factor* (EF) can be used as such a metric. When the client requests a set of objects Q that maps to $N(Q)$ distinct pages, EF is defined as:

$$EF(Q) = \frac{N(Q)}{\left\lceil \frac{||Q||}{L} \right\rceil}$$

where the denominator is the size of ideal packing of $||Q||$ objects to pages of L objects each. In the above example, EF can be as low as 1 and as high as 20, but in general EF ranges between 1 and L. If the client may issue any one of $Q_1, Q_2, ...Q_r$ queries, each with probability $P(Q_i)$, it makes sense to define the average EF:

$$\overline{EF} = \sum_{i=1}^{r} P(Q_i) EF(Q_i)$$

For a given set of queries and their probabilities, the ideal clustering mapping minimizes $\overline{EF}$. Although it is easy to avoid bad clustering mappings of $\overline{EF} = L$, in general it is very hard or impossible to find a clustering mapping of $\overline{EF} = 1$ [YSLS85].

Although this definition makes sense for records and queries, EF alone is not an adequate metric for clustering. The EF measures the distribution of objects to pages, but does not take into account the order and frequency with which each object is needed. There are a large number of possible clustering mappings[1] that have the same EF. However, not all of

[1] $\frac{n!}{(L!)^N}$ to be exact

them achieve the same performance on small cache sizes. For example, suppose we map 6 objects $\{a, b, c, d, e, f\}$ to 3 pages of size 2:

$$c_1 = \{[a, b], [c, d], [e, f]\}$$
$$c_2 = \{[a, f], [b, d], [c, e]\}$$

Both mappings c_1, c_2 achieve the same $EF = 1$ for all queries involving all 6 objects. Both perform well with caches of size 3 pages or more. However on a cache of 2 pages they may perform very differently. A query using objects in the following sequence:

$$t = (abc)^*(def)^* = abc\ abc\ abc\ \dots\ def\ def\ def\ \dots$$

will thrash under c_2, but it will work fine under c_1 only producing 3 page faults. It is easy to see that c_1 outperforms c_2 because it maps the frequently needed objects $\{a, b, c\}$ and $\{d, e, f\}$ to 2 pages whereas c_2 maps them to 3 pages.

In general, suppose we are given a clustering mapping such that a set of objects Q maps to $N = N(Q)$ pages. If the client cache is at least N pages big, the client will only experience an initial startup delay proportional to N, and after that, its computation will proceed at full speed. However, if the cache cannot hold N pages "thrashing" (a series of page faults) will occur, especially if some of Q objects are requested repeatedly. If the cache has size $C < N$, then every traversal through the objects will generate at least $N - C$ page faults, and the computation will proceed at a much lower speed (requiring at least $N - C$ server requests per iteration).

It is well known that the performance of caches depends on the "locality" of page requests. Clustering maps many objects to each page and thus it increases the page locality. On small caches mappings that achieve better locality will perform better. The average working set size can be used to measure locality [Den68], the lower the working set size the higher the locality. For a sequence of requests $(X_n) = x_1, x_2, \dots x_n$, the working set at time t is defined as the cardinality of the set of the last w requests:

$$WSS(w, t) = ||\{x_{t-w+1}, x_{t-w+2}, \dots x_t\}||$$

By representing access patterns as stochastic processes, clustering can be formulated as an optimization problem, and optimal clustering corresponds to the clustering mapping that minimizes the expected working set size. The complexity of "optimal clustering" in general (which has been shown to be NP-complete in [TN91]) makes it very hard to find the optimal clustering mapping. However, this formulation gives a new view to the problem, helps to

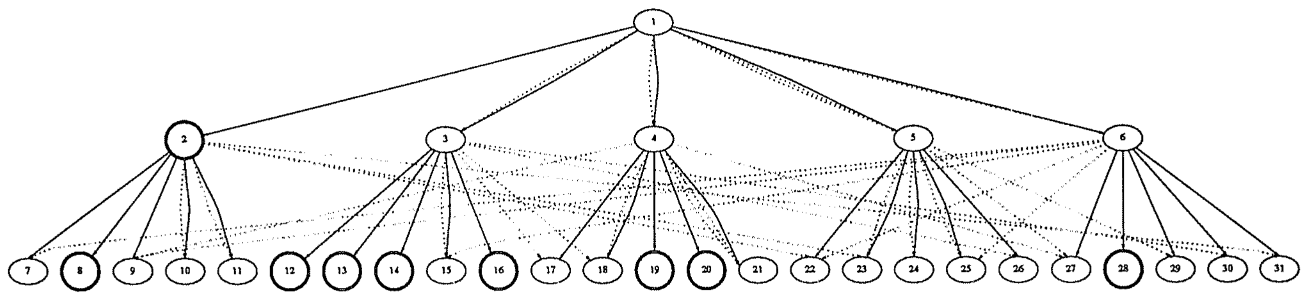

Figure 2: The Tektronix Benchmark Object Graph

come up with some more reasonable (close to "optimal") clustering algorithms, and also gives a practical limit to how much clustering can help performance.

The new formulation of clustering uses Markov processes to model "access patterns." A Markov process is a sequence of correlated random variables, which in our case models object requests; the probability to request some object y at time t strictly depends on the last object requested:

$$Pr\{X_{t+1} = y | X_t = x\} = P(x, y)$$

As a result, for a population of n accessible objects, this access pattern model requires at most n^2 parameters the values of the $P(x, y)$ matrix. Those parameters can be estimated from *sample sequences*, and in practice P is sparse, because of various access constraints.

3 Simulation Environment

In this study we have used cache simulations as a way to evaluate the performance of clustering algorithms. The evaluation of a clustering algorithm has three steps, the training, the clustering itself, and the testing. Training involves deriving a suitable access model for the algorithm (that is, a description of the access patterns) using the object graph and/or sample traces. Next, the clustering algorithms use their access model and the object graph as input and generate a mapping of objects to fixed uniform size pages as output. Finally, the generated clustering mapping is evaluated by running test traces over the object base using a cache simulator.

3.1 The Workload

Our experiments were selected from the CLUB-0 clustering benchmark for the O_2 system [HBD91]. CLUB-0 uses synthetic workloads based on a subset of the Tektronix Hyper-model

Benchmark [And90]. The object graph in CLUB-0 is a DAG, derived from a balanced 5-way tree with some additional edges.[2] Figure 2 illustrates the first levels of an actual tree we have used in our simulation. Each node of the tree has edges to 10 other nodes. Five of these nodes are reachable by "child" edges, hence are called children of the parent; the other 5 nodes are reachable by "part" edges and are called subparts of the parent. The subparts of a given node are chosen at random from among all the nodes at the same level as the children of the node. The depth of the tree used in CLUB-0 is 6 resulting in 3906 objects. A query is a collection of traversals, each traversal being a sequence of object accesses representing some hypothetical operation on the graph. A traversal starts from a node high in the tree, and at each step uses a fixed rule to select the next object to visit.

Structural operations on the graph are modeled by traversals denoted as AXB, which start from a randomly selected node at level B of the graph ($B = 0$ denotes the root of the tree). AnB denotes a Depth First Traversal (DFS) performed by exclusively following either the children hierarchy ($1nB$) or the parts hierarchy (MnB). We have added a new traversal AsB to model searches on the object graph. AsB locates some leaf level object by following a path on the graph, starting at level B and exclusively following the children hierarchy ($1sB$) or the parts hierarchy (MsB). The decision which of the parts (or children) to follow is taken randomly and independently on each level. We have also introduced *srnd*, for "skewed random", a modified *rnd* of the Tektronix Benchmark, that visits objects randomly with a probability that follows normal distribution. The variance in the skewed workload was set to 1/10 of the object base size, and the objects were selected based on their OID but using a random permutation first. As a result, hot objects are spread "uniformly over the object base", so there is no relationship between heat and the position in the object graph.[3] A query is just a sequence of traversals each one starting from a randomly selected node at the starting level B. A query trace is a concatenation of its traversal traces, and contains references to objects necessary to run the query. The traversal trace is obtained by trapping all object accesses occurring during the execution of that traversal, as if the code was running on an object oriented run time system. As a guide, we used the code produced by the non-swizzling E compiler [RC89] with all optimizations having to do with persistent objects turned off, so that object references appear every time the original traversal code requires access to the object state. Finally, for our purposes a workload is a trace obtained

[2]The object graph of the original Tektronix Benchmark contains one more relationship, the hyper links; those links are omitted here for simplicity as they were from the CLUB-0 benchmark.

[3]In addition to these queries, the CLUB-0/Tektronix benchmark includes several other queries. We did not present results from experiments on these workloads since they did not provide additional insight.

by mixing, concatenating, or interleaving query traces.

3.2 Clustering Algorithms

We have implemented and tested several algorithms based on heuristics and ideas discussed in the literature. The goal of all algorithms examined is to partition the object graph (OG) by assigning objects to uniform size pages. The object graph is formed considering objects as nodes and any reference from an object to some other object as a directed edge connecting them. Most clustering algorithms use as input a graph representation of the access patterns (called clustering graph or CG), assumed to be characteristic of the client behavior. The representation is usually derived from the object graph and/or from sample traces (the training traces).

In general, three types of CGs are used:

- The OG, i.e. the object graph itself; a rudimentary representation of access patterns that does not take advantage of any other knowledge that may be available. No statistical information from the training traces is captured in OG.
- The SG (Statistical object Graph); the object graph annotated with edge and node weights. The node weight (edge weight) is equal to the frequency the object (edge) appears in the training trace.
- The SMC (Simple Markov Chain) is the directed graph form of a first order Markov process that could have produced the object trace. For each accessible object in the system the graph contains a node. Any positive probability that one object can be accessed after some other object, is represented as a directed edge. The node (edge) weights are the the estimated stationary (transition) probabilities of the chain from the sample trace.

The plain object graph does not convey much access information. The last two graphs express the behavior of the client as it is manifested in the training trace, using a much more compact representation than the trace itself. The SG limits its information in the usage of objects and references, failing to capture access dependencies other than those that exist in the original object graph (for example, SG will not record a return from a node to its grandparent during a DFS traversal). SMC does not have this type of restriction since it records arbitrary transitions. However, it does not record access dependencies involving more than 2 objects because of its memoryless characteristics. Multi–dimensional access models (i.e. hyper-graphs) could be used to obtain more accurate access pattern description at the expense of size. Unfortunately processing those models would be computationally infeasible, since the high–dimensional hyper-graphs needed may have a large number of hyper–edges.

The majority of the algorithms assign objects to clusters sequentially, by performing a form of graph traversal on CG and assigning objects to clusters as they go. On each step the object is put in to the current cluster and if it fills up, a new empty cluster is created. Table 1 summarizes the algorithms used. The notation in that table is to generate the name of an algorithm from the type of clustering graph it uses (OG, SG, or SMC) and the style of the graph traversal or operation they perform.

The SMC.PRP and SMC.KL algorithms were proposed in [TN91]. The PRP (initially proposed for record clustering in [YW73]) method just uses the node weights of the SMC graph (i.e. the absolute probabilities), by sorting objects with respect to their probability and then assigning them to pages in that order. This scheme is also known as Probability Ranking Partitioning. The SMC.PRP algorithm has $O(n \log n)$ cost. The SMC.KL algorithm uses the standard Kernighan–Lin [KL70] graph partitioning algorithm to find a near to optimal clustering of the SMC graph. SMC.KL is the algorithm we have referred to in the introduction as "Stochastic Clustering." SMC.KL partitions the object graph so that the expected working set for window size 2 is minimized. SMC.KL is a heuristic partitioning algorithm that achieves only pairwise optimality, i.e., there will be no two nodes belonging to two different partitions that can be exchanged and result in a lower total cost partitioning. SMC.KL does not cluster sequentially, since it applies repartitioning until no cost improvement is possible. The complexity of SMC.KL is dominated by the complexity of graph partitioning and it is on the average $O(n^{2.4})$[PS82].

SMC.WISC, a new algorithm we propose, appears in many cases to approximate well SMC.KL in terms of performance. WISC is a traversal based (greedy) low cost graph partitioning that does non–backtracking clustering. Objects are visited with the order of their absolute probabilities, hotter objects first. Each un-clustered visited object is selected to start a partition. While there is room in the current partition, all objects *accessible* in terms of the SMC graph from the current contents of the partition are considered. The object that maximizes the overall probability of using that partition, is selected and the process is repeated until the partition fills up. At this point, the next un-clustered object from the sorted list is considered, and the whole process is repeated, until all objects have been clustered. SMC.WISC has cost $O(n \log(n))$.

The OG.DFS algorithm traverses the object graph in a DFS manner. It minimizes the number of different pages touched during a pure DFS traversal that uses all possible object graph edges. OG.BFS traverses the graph in a BFT manner, grouping siblings together as much as possible. Both algorithms have linear cost $O(n)$.

SG.WDFS is much like the OG.DFS except that during the DFS traversal, siblings are selected depending on how hot their edge is. This algorithm attempts to minimize the number of clusters "probable" DFS traversals will touch. Edge probability information can be supplied by the compiler based on static usage information (like in Semantic Clustering [SS90]), or as user hints (like those originally planned for E [RC89]). In our case, SG.WDFS gets its hints from the sample traces. SG.WDFS has linear cost $O(n)$. OG.BFS, OG.DFS and SG.WDFS have been proposed first in [Sta84] for clustering Smalltalk objects.

The SG.CACTIS is based upon the clustering algorithm proposed in [HK89], a heuristic algorithm that performs PRP scan of the objects, clustering the closure of each group as it is formed. Quoting from [DK90],

> Clustering starts by placing the most frequently referenced object in the database in an empty block. The system then considers all relationships that go from an object inside the block to an object outside of the block. The object at the end of the most frequently traversed relationship is placed in the block.

To apply this method we interpret "relationship" as "edge in the object graph."

Finally, we have implemented OG.PT, an algorithm based on placement trees. A placement tree is a pattern that matches a subset of objects reachable from a given node, the root of the placement tree. The matched objects are always connected with object pointers and form a tree. OG.PT traverses the object graph in some manner (e.g. BFS or DFS), and matches a given placement tree against the object graph starting at the visited object. All matched un-clustered objects are inserted to a logical cluster, which is subsequently assigned to physical pages. Although OG.PT is very intuitive, it is not an automatic method. On O_2 the data base administrator selects the placement trees [Deu91] based on his system experience. In our implementation, OG.PT refers to the performance of the best placement tree we were able to find for the workload in question. As we will see, there are cases where OG.PT achieves very good performance, mainly when the object graph is regular and is used in a uniform way.

3.3 Cache Simulation and Performance Metrics

Each clustering mapping was tested using a client simulator as in [TN91], and [HBD91]. The input to the simulator is the testing trace and a clustering mapping. Each object reference is converted to the corresponding page reference using the mapping. Then, that page is

Algorithm	Complexity	Proposed
OG.DFS	$O(n)$	[Sta84]
OG.BFS	$O(n)$	[Sta84]
OG.PT	$O(n)$	[BD90]
SG.WDFS	$O(n)$	[Sta84]
SG.CACTIS	$O(n \log n)$	[DK90]
SMC.PRP	$O(n \log n)$	[TN91]
SMC.WISC	$O(n \log n)$	here
SMC.KL	$O(n^{2.4})$	[TN91]

Table 1: Tested clustering algorithms

retrieved from a variable size LRU cache, and the number of page hits is updated for each cache size. Periodically or at the end of the simulation the average cache hit rate is reported.

The primary performance metrics used in this study, are the client cache hit ratio HR and the expansion factor EF (defined in Section 2). EF is more appropriate when cache sizes are large, where the particular arrangement of objects to pages does not matter. EF is less meaningful when client caches are small compared to the portion of the object base being accessed. HR illustrates better the performance of clustering mappings, when traversals are longer and fill up a small cache.

4 Results

In this section we present the results of our experiments with the performance of the clustering algorithms on a variety of workloads. In addition to exploring the performance of the algorithms on "pure" workloads, we also investigated the performance of the algorithms when "noise" references or references from workloads other than the training workload appeared in the testing trace. (Recall that saying that the training and testing traces are from the same workload does not imply that they are identical traces, since all of the workloads have a strong random component.) The experiments presented here are for a uniform object sizes of 200 bytes (typical, as reported in [Bai89]), and pages of 4k bytes each.

4.1 Performance of single traversals

This experiment tests the algorithms with respect only to their EF, i.e. their capability to group all objects requested during a single traversal as dense as possible. We measured

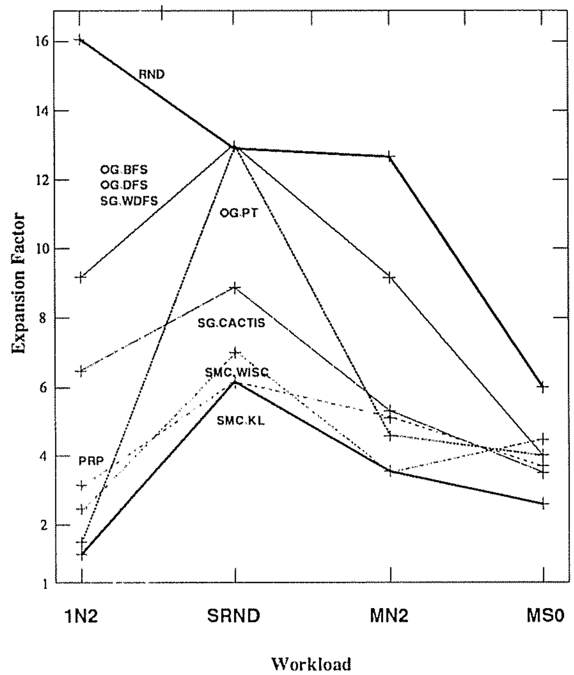

Figure 3: Performance on pure workloads

EF by running the client on a large cold cache, and counting the number of page faults at the end of the traversal. Since the client cache is initially empty this number gives exactly $N(Q)$, the number of pages the traversal is mapped to (also called "traversal pages"). EF is also an indication of the "loading time", or how quickly the required objects for a traversal are brought in. To estimate $\overline{EF}$, the experiment is repeated a number of times, each time selecting a different traversal.

Figure 3 shows the $\overline{EF}$ achieved by clustering algorithms on a variety of workloads. Note that one of the curves in the graph has three labels: OG.BFS, OG.DFS, and SG.WDFS. The meaning of this notation is that the curves for those three graphs were virtually indistinguishable within the margin of experimental error. We will follow this convention of condensing indistinguishable curves and their labels to a single curve with the union of the original labels in the graphs throughout this section. Workloads are ordered with respect to the amount of active objects, i.e. the percentage of the object graph they use. 1n2 accesses the most objects and mn2/ms0 the least.[4] The graph shows that very few algorithms perform always close to the minimum EF=1.The expansion factor of all algorithms fluctuates as the workload changes, and random clustering gets gets better as fewer the objects are accessed. Statistical

[4]We will postpone the explanation to Section 4.2

algorithms (SG.CACTIS, SG.WDFS, SMC.KL,SMC.WISC) are more adaptive, since they take advantage of the access pattern training. Algorithms based only on the object graph are less adaptive, since they do not train on access patterns. Most algorithms do not have a constant rank for all workloads, except of SMC.KL (best) and RND (worst). Note that the skewed random query (srnd) makes most algorithms as bad as random clustering, since srnd does not follow the object graph at all.

Notice that PRP performs well, although not optimally, on all of these workloads. The good performance of PRP on the object-graph traversal workloads (mn2, 1n2, and ms0) is surprising, since PRP never looks at the object graph and clusters based solely on the zero-order statistics from the training trace. The performance can be explained as follows. Since the root objects for the traversals in the training trace are randomly selected, due to randomness these objects will be selected with slightly differing frequency. Since every traversal out of the same root object is identical, every object (except for the leaves of the traversal) is selected with a frequency identical to that for the root of the traversal. Ignoring duplicates between traversals (which are relatively few in these workloads), this in turn means that the frequencies for objects belonging to the same traversal are identical and slightly different from the frequencies for objects belonging to different traversals. Finally, this means that since PRP groups objects according to decreasing frequency of access, it will tend to cluster an object with other objects that belong to the same traversal. In effect, PRP is using the different frequencies of objects belonging to different traversals to "learn" the structure of the object graph.

Placement trees are extremely good in the 1n2 queries, where traversals are disjoint and well known in advance. Each traversal accesses its own subtree rooted at level two, and a placement tree can easily pack objects exactly that way. As a result OG.PT gets an EF close to 1. In non-graph queries like srnd, placement trees are not applicable at all, since by definition placement trees attempt to group together only objects connected by edges of the object graph. Since the traversals of mn2 (ms0) are not disjoint (there can be multiple paths through part edges to the same node), we could not come up with placement trees that group each traversal as well as in the 1n2 case. Finally, the simple structural algorithms (OG.DFS,OG.BFS) cannot compete with the more sophisticated placement trees.

The stochastic clustering algorithm is definitely a winner in terms of EF. It performs as good as the manually chosen placement trees in its ideal case (1n2), it adapts to arbitrary random queries (srnd) as well as to the queries that traverse the object graph in a variety of ways.

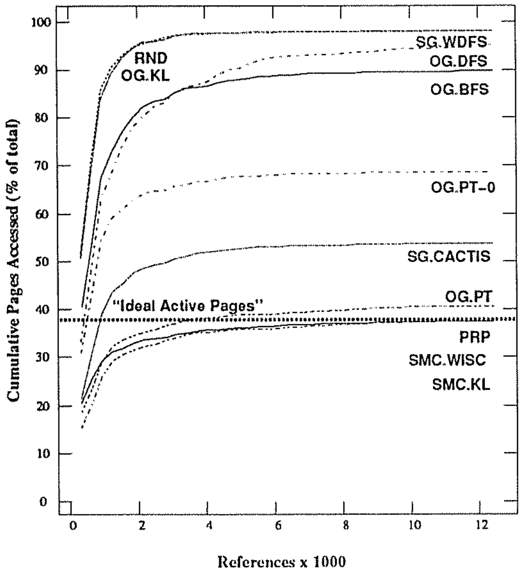

Figure 4: Cumulative page faults going to steady state for mn2

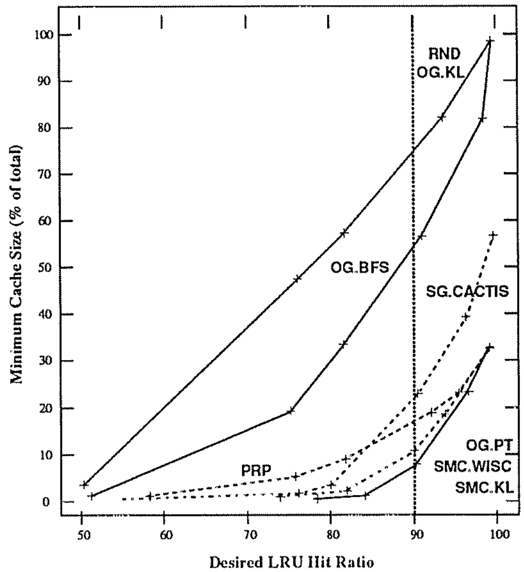

Figure 5: Minimum cache size to achieve a hit ratio for mn2

4.2 Steady State Performance

It is intuitively known that large caches are more forgiving to less efficient clustering mappings than are small caches, an effect we want to illustrate with this experiment. If the object graph is used repeatedly and the client cache is "large", then EF(or even $\overline{EF}$) is not the right metric of clustering performance. In the case of cold caches the number of page faults is proportional to EF. If the cache is not empty, then in addition to the intra-traversal locality, the hit ratio is influenced by the amount of pages shared between traversals (the inter-traversal locality).

The cache gets "hotter" as more and more traversals are executed on the client. The first traversals achieve a lower hit ratio, and the cache is in effect loaded with pages that were used by old traversals. Subsequent traversals find more and more of their needed pages in the cache, and their hit ratio improves. A simple analysis of random clustering suggests that very quickly, the whole object base will be touched, since there is a high probability that each object referenced by a traversal will be found on a different page. In the mn2 case each traversal makes 311 object requests and accesses up to 156 distinct objects. Under random clustering, about 1040 randomly selected objects are required to touch all 196 pages of the object graph. To touch 1040 objects, 9 different traversals are required approximately,[5]

[5]This number is only approximate because of the possible duplicates in the mn2 traversals and the randomness in selecting distinct traversals from all 25 possible ones.

thereby producing 2800 mn2 references. Indeed, as the results of Figure 4 show, random clustering arrives at the steady state in approximately 3000 references. Random clustering might seem an ideal way to increase the inter-traversal locality, and this is certainly true if the whole object graph is accessed with the same probability and ample cache memory is available.

However, an interesting property of the Tektronix Benchmark object graph restricts the number of objects accessed during mn2 traversals. The mn2 query uses exclusively the parts relationship, indicated by the dashed edges of Figure 2. In our case on each level h of the tree (level 0 being the root) there are $A(h) = 5^h$ nodes that point to $A(h+1) = 5^{h+1}$ children through the parts edges. Since each node has only 5 parts, the expected number of orphans (i.e. children that are not connected by the parts edges) at level h is:

$$A(h)\left(1 - \frac{1}{A(h)}\right)^{A(h)}$$

Those nodes cannot be reached by any node higher in the tree and are represented as round objects in Figure 2. If the previous formula is applied recursively from the starting level up to the leaves, it can be derived that the expected number of total nodes accessible for the mn2 query that starts from the second level of the 6-level tree, is just 33% of the total (for the derivation please refer to Appendix A.2).

The graph of Figure 4 shows the cumulative number of pages accessed during mn2 traversals as a function of the number of references, using a large initially empty cache, that can hold the whole object base. The graph begins after the first traversal has been performed, so the curves start from a position proportional to EF. Initially all clustering mappings rapidly bring in a number of pages, and after some point, they have a very slow page fault rate thereby reaching the steady state. The curve labelled as OG.KL corresponds to optimal clustering of the object graph using the Kernighan–Lin partitioning algorithm and giving all edges equal waits. In most of the cases, OG.KL performed no different than random clustering, and therefore straightforward partitioning even of a relatively sparse object graph may not be a good clustering heuristic. The OG.PT-0 curve corresponds to the second to the best placement tree (OG.PT) we found for this workload, and it performs much worse.

The number of pages touched at the steady state shows an important property of clustering algorithms; their capability to map the active portion of the database to the minimum possible number of pages. The *long term expansion factor* or EF_∞ is an indicator of the steady state performance. EF_∞ is the ratio of pages accessed in the steady state (N_∞) to the number of pages that would be required ideally to pack all active objects (n_∞):

$$EF_\infty = \frac{N_\infty}{\left\lceil \frac{\|n_\infty\|}{L} \right\rceil}$$

In the object graph we tested, there were actually 1498 accessible objects through the parts edges, corresponding to 38% of the object graph[6] represented by the dashed line in the graph of Figure 4. EF_∞ is definitely related to EF; EF measures the average packing capability for each one of the possible traversals that influence EF_∞. In our case, since there are 25 possible equiprobable traversals, EF_∞ cannot be worse than $25EF$. However, large EF does not necessarily mean large EF_∞. Imagine a (possibly unrealistic) clustering mapping, that arbitrarily packs all accessible objects to a small number of pages. This mapping may have a bad EF since active objects accessed at different traversals will be mixed, but its EF_∞ will be the optimal since no inaccessible objects will be clustered with accessible objects.

If only EF_∞ is our concern (i.e. when the system operates near the steady state), then SMC.PRP suffices. PRP uses the absolute access probabilities of objects, and packs objects according to it. As a result PRP achieves the minimum EF_∞. Note that this is true even though PRP makes no use whatsoever of the object graph. The performance of placement trees in steady state can now be explained, since they manually assign all objects that can possibly be accessed during a traversal to a single cluster, assuming that in the worst case all reachable objects are used. Although their EF was higher than SMC.KL (see Figure 3), their EF_∞ was not much worse than optimal, due to sharing of pages between traversals.

The graph of Figure 5 is another interpretation of the steady state performance, indicating the minimum cache size needed to guarantee a given hit rate at the steady state (note that the graph ends at 99% and not 100%). The graph can be used for selecting a clustering algorithm for the mn2 workload, given an amount of available memory and a desired hit ratio. If the cache size can be close to the EF_∞ of a clustering algorithm, then the hit ratio will be close to 100%, and the algorithm is acceptable. If memory is a constraint, then potentially more expensive algorithms with smaller EF_∞ should be used. For example at the steady state, random clustering we achieve 90% hit ratio by caching 80% of the object graph. The same hit ratio can be achieved with SMC.KL/WISC/OG.PT by caching less than 10% of the object base.

[6] It is 38% on this particular graph and not exactly 33% due to statistical variation.

Graph Name	Graph Levels	Total number of objects	S1 experiment (obj/traversal)	S2 experiment (obj/traversal)	S3 experiment (obj/traversal)
TB1k	5	656	mn2 (131)	mn2 (756)	
TB4k	6	3781	mn2 (756)	mn3 (756)	mn2 (756)
TB20k	7	19406	mn2 (3881)	mn4 (756)	
TB20kR	7	19406			mn3 (756)

Table 2: Scale-up queries

4.3 Increasing the problem size

So far, the evaluation of clustering algorithms has been conducted on object graphs of fixed size, exactly as specified in the CLUB-0 benchmark. Next we will present some scale-up results, involving object graphs of different sizes and structure. The expansion factor results are much harder to understand across different graph and query sizes, so the main performance metric used is the traversal pages (i.e., the number of pages needed to hold all objects accessed during a traversal). Interestingly, the relative order of algorithms is maintained as the traversal size increases with the object graph, but not in other cases. Results from the steady state performance did not provide any insight to the problems and they will be omitted here due to lack of space.

We selected three different object graphs, a small (5 levels) a medium (6 levels) and a large (7 levels). Each graph is about 5 times larger than the previous one, and is constructed according to specifications of CLUB-0 except for the size. The third experiment uses just two graphs and it will be explained later (Table 2 shows the exact parameters used). We chose to present the Depth First Search traversal on parts (mnx), since it produced the most interesting results. Figure 6 gives a graphical representation of queries and object graphs used.

Experiment S1 runs the same mn2 query on all graph sizes, and the traversal objects (the objects accessed per traversal) range from 131 (on TB1k) to 3881 (on TB20k) being proportional to the object graph size. As the graph of Figure 7 shows the traversal pages increase with the graph size smoothly, and most importantly the order of the algorithms remains the same. The loading factor (i.e. the ratio of traversal pages to traversal objects) remains approximately the same for all algorithms.

If scale-up involves queries that access the *same number of objects per traversal* regardless

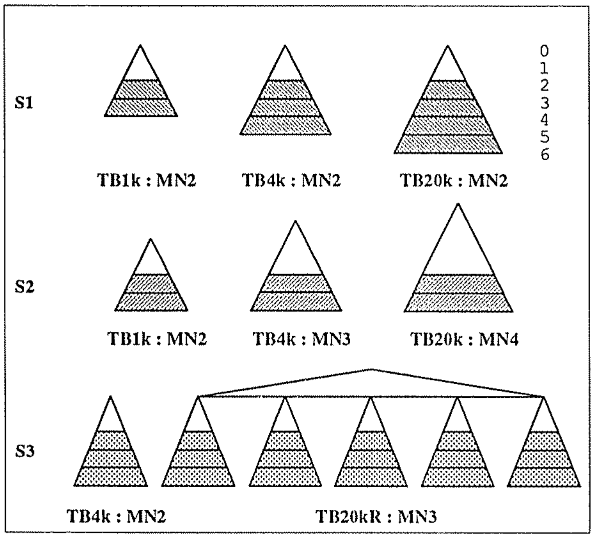

Figure 6: The scale-up experiments setup

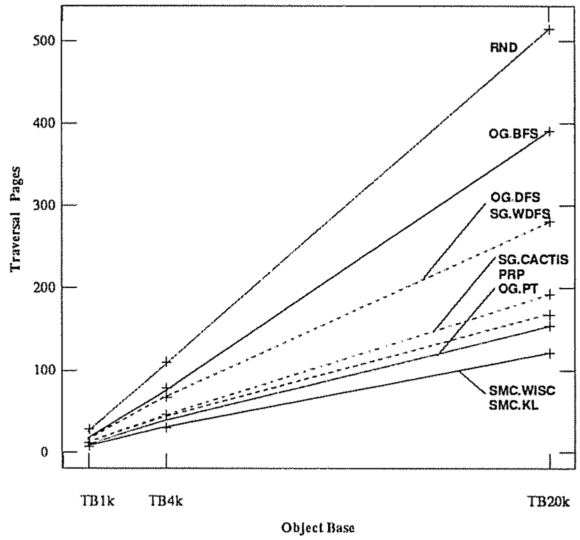

Figure 7: Scale-up Experiment #1

the object graph size, there are some surprises, because of the changing traversal localities. Experiment S2 always accesses 756 objects, by starting the traversals 1–level lower for every 1–level higher graph. Because queries start lower on bigger graphs, there are more possible traversals and less sharing (common sub-traversals). For example, in the 5–level case the mn2 query starts at the second level, and each part edge is selected among 125 possible nodes of the third level. In the 7–level case, there are 3125 possible traversals, and there are virtually no duplicate objects. As a result, such queries on larger graphs will have less duplicates per traversal (i.e. lower intra–traversal locality) and less sharing between different traversals (inter–traversal locality).

Figure 8 shows a sharp increase in the traversal pages, because the intra–traversal locality drops significantly going from the 5–level to the 6–level graph, and random clustering illustrates that effect. As we move to the 7–level tree, most of the algorithms access the same number of pages or slightly less, except for PRP. For an explanation of the PRP performance please refer to Section 4.1. Since SG.CACTIS is basically based on PRP, it performs slightly worse too. Finally, by tuning placement trees we were able to maintain their previous rank.

A real object base may contain sets of similarly structured complex objects, and the next experiment investigates that case. TB20kR uses a 7–level object graph constructed by 5 disjoint TB4k graphs as Figure 6 shows. The algorithms were tested with mn3 on TB20kR and mn2 on TB4k, to ensure that on both graphs 756 objects are accessed per traversal. Interestingly, the mn3 traversals have much less inter–traversal locality than the mn2 ones: A given traversal from TB4k can share (i.e. have common subtraversals) with any of the rest

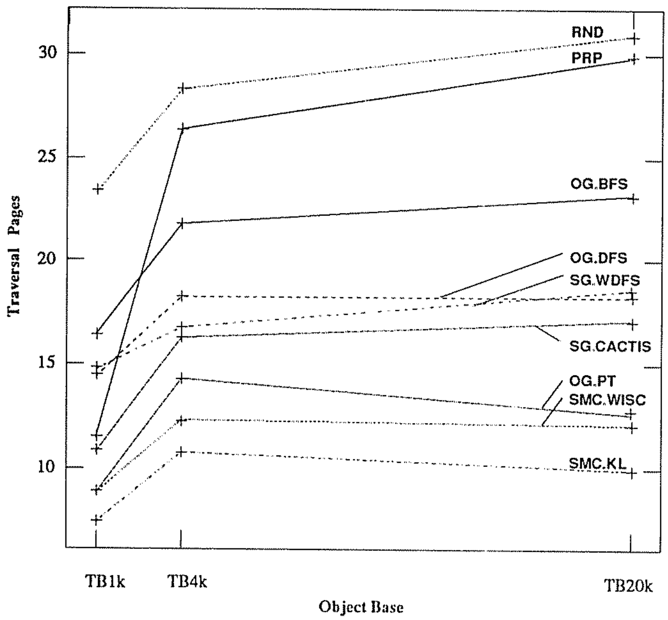

Figure 8: Scale-up Experiment #2

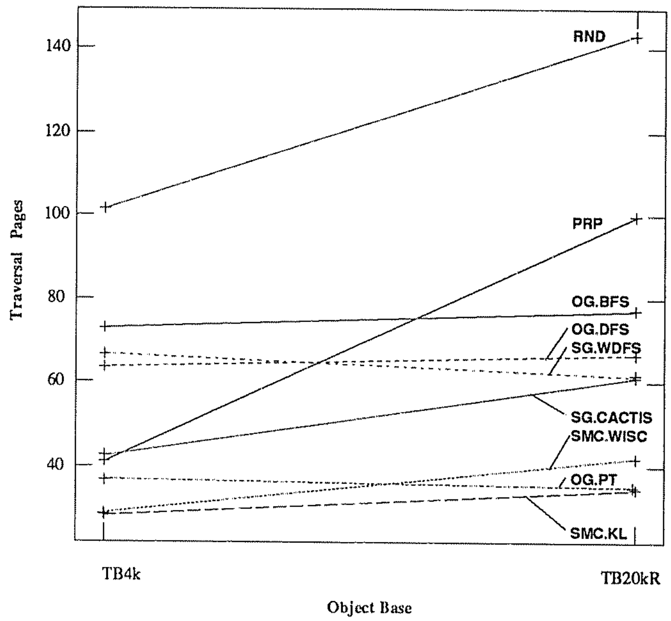

Figure 9: Scale-up Experiment #3

24 possible mn2 traversals. However, a given traversal from TB20kR can only share with with any of the other 4 traversals that belong to the same subtree; as opposed to TB4k, the structure of TB20kR does not allow sharing between all the $25 \times 5 = 225$ possible traversals of TB20kR.

It is interesting to observe that even the simplest structural algorithms like OG.BFS/DFS, are not affected much by the factor of 5 increase in size (graph of Figure 9). PT improves by a small factor, helped by the smaller inter–traversal locality. Inside each subtree a placement tree may mix traversals; different subtrees however will never be mixed though. PRP in general will mix subtraversals: Mixing two subtraversals in the TB4k case is not as bad as in TB20kR, since in the former case the probability that two clustered subtraversals belong to the same traversal is very high. As it can be seen from the graph, PRP quickly deteriorates almost by the same factor as random clustering, and as a result the PRP based SG.CACTIS and SMC.WISC are both affected, but to a lesser degree since they also use the structure of the clustering graph.

4.4 Noise effects

Most clustering algorithms base their performance on the knowledge of access patterns, as it is registered in their access models. Real access patterns however, may be different than the ones used for training. One way real access patterns may be different is that some unexpected

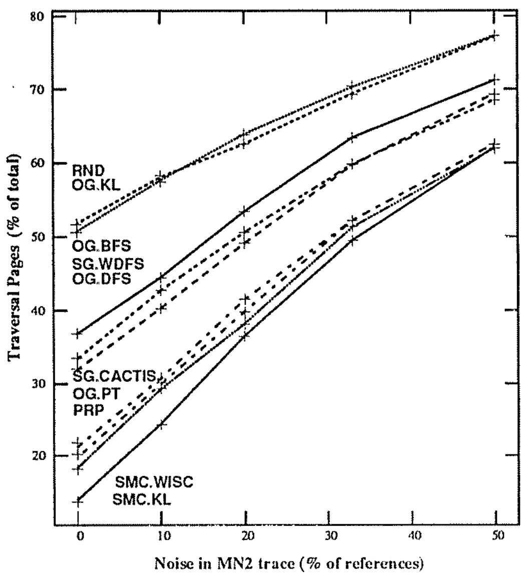

Figure 10: "Noisy" references in mn2 queries

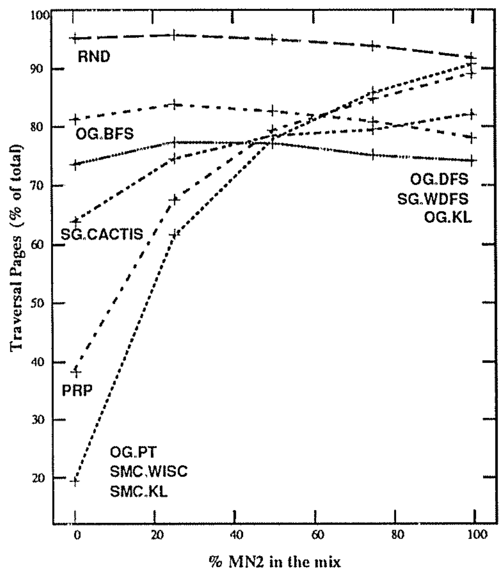

Figure 11: Changing access patterns

references may appear in the actual trace. To study this effect we added white noise to the testing object stream (as in [PZ91]). *White Noise* is a stream of random references chosen with uniform probability from the whole object population.

The graph of Figure 10 shows the average number of traversal pages of mn2 as a function of noise level. For small caches, practically every noisy reference is a miss, so we should expect an increase of the page faults with noise level. With 20% noise level there is one noisy reference every 5 mn2 references, resulting to 62 random references during an mn2 traversal. 62 random objects map to 53 pages (out of 200) on the average. Since SMC.KL maps an mn2 traversal to 28 pages, there will be 7 pages in common on the average and therefore, 74 pages accessed totally (or 38%). Similarly, random maps 156 objects to 101 pages and has 27 pages in common with the noise on the average. As a result, it should require 128 pages (65%) and indeed it does.

4.5 Changing Access Patterns

So far, clustering algorithms were tested on pure traces, that is, traces containing traversals from a single query. The mixed workloads contain traversals from more than one pure workload. Mixed workloads will be used to observe the behavior of the clustering algorithms when the training trace is not entirely indicative of the testing trace.

Since efficient statistical algorithms are heavily dependent on the expected access patterns, they should be affected the most by differences between testing and training. Less efficient algorithms will be less affected, and finally random clustering should not be affected at all. This phenomenon resembles the behavior of a filter designed to accept a certain frequency and reject others, so we will call it the *tuning effect.* The interesting question is the tuning sensitivity and the next experiment was done to investigate it.

In this experiment the clustering algorithms were trained with a pure 1n2 workload, but are tested with a varying mix of 1n2 and mn2 traversals. Although functionally both traversals perform the same DFS traversal starting from the same randomly selected nodes, they follow different edges in the object graph, and generate different access patterns. The graph of Figure 11 shows the (average) number of traversal pages as a function of the mix proportion. The random algorithm remains practically unaffected, and the less efficient algorithms did not loose much either since their performance does not depend on the access patterns. The most efficient algorithms remain good initially, but they lose performance quickly and for a mix of over 50% they are no longer the best.

From this experiment, one can conclude that the simple structural algorithms (which behave like coarse filters) are not bad for dealing with highly unpredictable access patterns. Good algorithms behave as more fine filters and great care should be taken when using them, since a significant change in the access patterns may affect them dramatically. We did additional experiments that trained on a mixed workload and tested on a varying mix; that mix did not contain "unknown access patterns" but simply different proportion of known access patterns, and the tuning effect was not as severe as here.

5 Conclusions

We have investigated the performance of a number of well-known clustering algorithms over a standard object-oriented benchmark, the Tektronix Hyper-model benchmark. The main points observed in our study are that

- Stochastic clustering, while expensive, performed the best in all the tests we ran. While other, less expensive algorithms on occasion performed similarly to stochastic clustering, no single algorithm was close for all workloads, so if stochastic clustering cannot be used, care should be taken to match an appropriate inexpensive clustering algorithm with a given application.

- The more precise the clustering algorithm, the more sensitive it is to mismatches between training and testing access patterns.

- In contrast to mismatched access patterns, sporadic unexpected references affect the clustering performance of algorithms by approximately the same degree, and the algorithm ranking is not affected at all. As a result, such references can be safely ignored during the statistics gathering process.

- For cold-cache traversals of small portions of the object graph, the expansion factor is the important performance metric; for steady-state large cache performance the number of pages to which the clustering algorithm maps the active portion of the database is the appropriate metric. A clustering algorithm that performs well on one metric may not perform as well the other.

- Structural clustering techniques and especially placement trees, are very effective when a subset of the object graph edges is almost exclusively used to reach objects. However, when accesses are not done through the object graph (like the SRND queries), or when they use most edges of a complex graph structure (like mixed mn2/1n2 workloads), additional statistical information is necessary to produce a good clustering mapping.

We are currently investigating further issues in object clustering, including the effect of non uniform object sizes, low-cost approximations to stochastic clustering, efficient algorithms for re-clustering sets of objects, and techniques for generating and propagating statistics throughout the database based upon type information and partial access statistics.

Acknowledgments

We are thankful to professors Miron Livny, Mike Carey, and Mary Vernon and to Dr. James Stamos, for their helpful advice and suggestions.

References

[And90] T. Lougenia Anderson et al. The Tektronix HyperModel Benchmark. *EDBT*, 1990.

[Bai89] P. Bailey. Performance evaluation in a persistent object system. In *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, September 1989.

[BD90] Veronique Benzaken and Claude Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. Technical Report 50-90, Altair, August 1990.

[Den68] P. J. Denning. The working set model of program behavior. *Comm. ACM*, 11(5):323–333, May 1968.

[Deu91] O. Deux et al. The O_2 system. *ACM Communications*, 34(10):34–49, 1991.

[DK90] Pamela Drew and Roger King. The performance and utility of the CACTIS implementation algorithms. In *Proceedings of the 16-th VLDB Conference*, pages 135–147, Brisbane, Australia, 1990.

[HBD91] Gilbert Harrus, Veronique Benzaken, and Claude Delobel. Measuring performance of clustering strategies: the CluB-0 benchmark. Technical Report 66-91, Altair, January 1991.

[HK89] Scott E. Hudson and Roger King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Data Base Systems*, 14(3):291–321, September 1989.

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

[KSC78] Valentin F. Kolchin, Boris A. Sevast'yanov, and Vladimir P. Chist'yakov. *Random Allocations*. Halsted Press, 1978.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice–Hall Inc, 1982.

[PZ91] Mark Palmer and Stanley B. Zdonik. FIDO: a cache that learns to fetch. In *Proceedings of the 17-th VLDB Conference*, Barcelona Spain, 1991.

[RC89] J. Richardson and M. Carey. Persistence in the E Language: Issues and Implementation. *Software Practice and Experience*, 19, 12 1989.

[SS90] Karen Shannon and Richard Snodgrass. Semantic clustering. In *Proceedings of the 4-th Int'l Workshop in Persistent Object Systems*, pages 361–374, Martha's Vineyard, MA, September 1990.

[Sta84] James W. 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] Manolis M. Tsangaris and Jeffrey F. Naughton. A stochastic approach for clustering in object stores. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 12–21, Denver, Colorado, May 1991.

[YSLS85] C. T. Yu, Cheing-Mei Suen, K. Lam, and M. K. Siu. Adaptive record clustering. *ACM Transactions on Data Base Systems*, 10(2):180–204, June 1985.

[YW73] P.C. Yue and C.K. Wong. On the optimality of the probability ranking scheme in storage applications. *JACM*, 20(4):624–633, October 1973.

A Appendix

A.1 The short-term performance of PRP

Although the performance of the PRP algorithm in the steady state is expected, it seems counterintuitive that PRP achieves good short term performance (EF) too. This has puzzled us, until we realized the following: The mn2 workload consists of a number of DFS traversals of the parts edges. Each traversal accesses 156 objects (*"the traversal objects"*), possibly with some of them appearing multiple times. The DFS code generates access patterns in such way that all objects in a traversal but the leaves have the same probability of access, say a, whereas all leaves have probability equal to $\frac{a}{2}$. Assume for the moment that traversals access distinct objects, and that there are no duplicates in each traversal.

The training workload for mn2 is a long random sequence of mn2 traversals. Because of randomness, it cannot be expected that all traversals will show up the same number of times; in fact, most of the traversals will appear slightly different number of times. As a result, the estimated probabilities of one traversal objects will be slightly different than the estimated probabilities of some other traversal objects. In other words, traversal leaf and non-leaf objects will be clustered with respect to their absolute probabilities. Since PRP uses those probabilities to cluster, it automatically groups together traversal objects.

Since the mn2 parts edges are selected randomly, a whole subtree may be accessible by more than one node, and that creates duplicate subtrees (if the subtree is accessible more than once from the same traversal), or shared subtrees (if a subtree is accessible by more than one different traversals). Such a subtree will always have higher probabilities than others, since it can be accessed by more than one places. Under moderate to light amount of sharing/duplicates, the previous analysis is approximately correct, since all the shared/duplicate subtrees are clustered first.

The graph of Figure 12 illustrates the effect, by marking the probabilities of objects that are accessed by a single mn2 traversal. As you can see many such objects have distinct probabilities and they are the only ones with that probability. If a very large number of traversals were contained in the training trace, then the graph would only have 2 probability levels; one for the leaves and one for the rest. If no sharing existed objects of a single traversal would be on just two levels.

In general, suppose the workload consists of a number of traversals each one visiting a fixed set of objects only once; also we have a training trace that consists of a long sequence of randomly selected traversals; finally, there is no object sharing between traversals. Because of randomness, we cannot expect that each traversal is going to show up the same number of

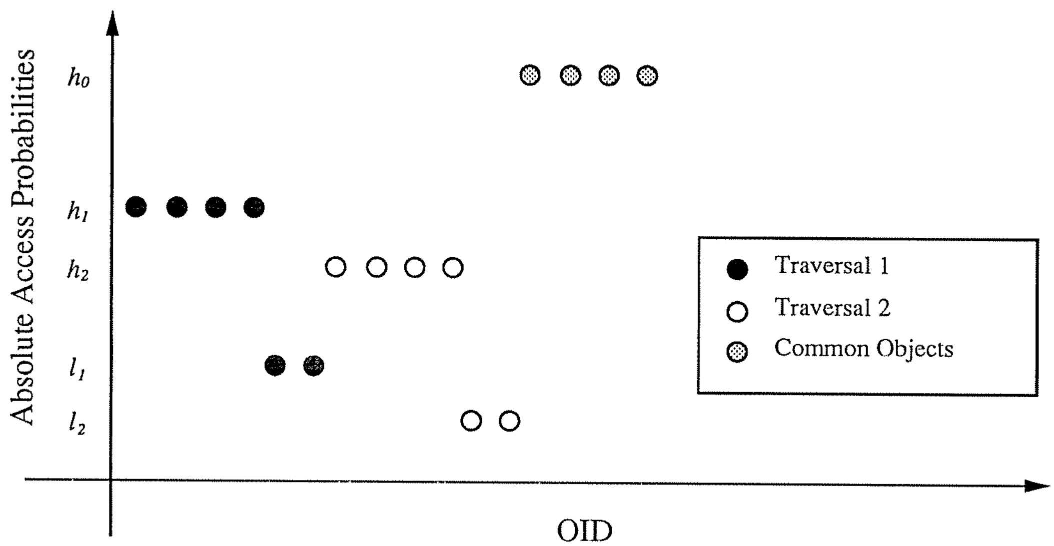

Figure 12: Object Probabilities on mn2

All unshared objects of traversal 1 (2) are clustered in two levels h_1 and l_1 (h_2 and l_2). Shared objects of traversal 1 and 2 have higher probability (h_0), but are still clustered together.

times in the trace. In fact, most traversals will appear different number of times, and some may not appear at all (especially if the trace is short). As a result, the estimated access probabilities for all objects that belong to the same traversal will be the same. In contrast, probabilities of objects in different traversal will be different. Thus, access probabilities automatically cluster objects of the same traversal, and PRP performs well since it does not mix objects from different traversals.

If there are objects commonly accessed by different traversals they will have a higher probability than the non-shared objects. This will distinguish them from the non-shared objects, and the PRP algorithm will cluster them first. Thus PRP may mix objects from different traversals, but under low sharing (like in the Tektronix Benchmark Object Graph with the given parameters), the amount of mixing will not be much. In any case, clustering such hot objects is not bad for the long term performance; shared objects are more likely to be needed in future traversals than unshared objects.

Level (t)	N_t	A_t	(%)	TN_t	TA_t	ρ(%)
2	25	15	(63.96)	31	15	(51.58)
3	125	59	(47.39)	156	75	(48.22)
4	625	236	(37.76)	781	311	(39.85)
5	3125	983	(31.46)	3906	1294	**(33.14)**
6	15625	4217	(26.99)	19531	5511	(28.22)
7	78125	18480	(23.65)	97656	23991	(24.57)
8	390625	82285	(21.06)	488281	106276	(21.77)
			....			

Table 3: Reachability Figures for TBOG

In our experiments we used a 6 level tree ($t = 5$) with 3906 total nodes. On the average, only 1294 nodes are accessible, i.e. 33.14%. The actual graph we used had slightly more accessible objects (approximately 38%) due to statistical variation.

A.2 Reachability of the Tektronix Benchmark Object Graph

We examine here the reachability properties of the Tektronix Benchmark Object Graph (TBOG). TBOG has nodes connected by the "children edges" forming a complete c-way tree hierarchy. In addition, another logical tree hierarchy is formed by connecting a node at level t with p nodes from the next level $t+1$ using the parts edges. The target nodes at level $t+1$ are selected randomly with uniform probability among the

$$N_{t+1} = cN_t$$

nodes of the $t+1$ level (N_t represents the number of nodes at level t and $N_0 = 1$).

Now suppose that a traversal starts at level h and only accesses TBOG nodes through the parts hierarchy. Let A_h be the number of nodes accessible at level h. At the next level up to pA_h nodes are accessible, with possible duplicates. The expected number A_{h+1} of accessible objects at the $h+1$ level corresponds to the expected number of occupied boxes when pA_h balls are thrown randomly with uniform probability to N_{h+1} boxes. Using the results from the problem of "balls and boxes" [KSC78] we get:

$$A_{h+1} = N_{h+1} - \left(1 - \frac{1}{pA_h}\right)^{pA_h} N_{h+1}$$

As A_h grows and N_{t+1} remains fixed, the expected number of inaccessible objects goes to 0. However, for small values of A_h there will be quite a few inaccessible objects.

A query that starts at level h_0 and goes up to the level $h = t$ will access on the average TA_t nodes of the total TN_t:

$$\begin{aligned} TN_t &= \sum_{h=h_0}^{t} N_h = \frac{p^t - 1}{p - 1} \\ TA_t &= \sum_{h=h_0}^{t} A_h \\ \rho &= \frac{TA_t}{TN_t} \end{aligned}$$

where ρ gives the ratio of accessible objects to the total objects.

Table 3 shows ρ as a function of the graph depth, for a TBOG with $c = 5$, $p = 5$ when the query starts at level $h_0 = 2$. From there you can see that the 6-level TBOG we used in our experiments allows access to only 33% of the object graph on the average.

Contents

List of Figures

List of Tables