# Principles for Static Clustering for Object Oriented Databases

Emmanuel-Manolis Michael Tsangaris

Technical Report #1104

August 1992

# PRINCIPLES OF STATIC CLUSTERING FOR OBJECT ORIENTED DATABASES

by

Emmanuel–Manolis Michael Tsangaris

mt@cs.wisc.edu

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

Technical Report #1104, August 1992

at the
University of Wisconsin-Madison
1992

## Abstract

This thesis addresses problems of clustering in the context of object oriented databases, from a theoretical and a practical point of view. We identified and formulated the clustering problems of Object Oriented Systems, derived optimal solutions to those problems, proposed practical clustering algorithms, and completed a thorough performance evaluation of those algorithms as well as many others. Our major results are the following:

1. With respect to the clustering problem, object access patterns can be reasonably modeled using stochastic access models.

2. The fundamental Optimal Clustering problem is NP–Complete, being an instance of hyper-graph partitioning. Asymptotically, however, optimal clustering reduces to probability ranking partitioning and has low complexity ($N \log N$).

3. We proposed two new heuristic clustering algorithms that outperform most of the existing clustering techniques if access patterns information is available. However, good static clustering mappings are expensive to obtain and can be very sensitive to the access patterns.

Εις τον πατεραν μου οφειλω το ζειν
και εις τον δασκαλον μου το ευ ζειν.

Αριστοτελης

I owe my life to my father
and my good life to my teacher.

Aristotle

This thesis is dedicated to my parents and to all my teachers.

# Acknowledgements

Being a graduate student at UW-Madison/Computer Science Department was a stimulating experience. Its challenging environment taught me how to develop, present and qualify new research ideas.

It is with my deepest appreciation that I acknowledge my advisor Jeffrey F. Naughton for his constant encouragement and continuous support. In addition, his insightful comments and suggestions made my academic research fascinating.

I would also like to thank Prof. David DeWitt and Prof. Miron Livny and Prof. Carey for patiently reading my thesis and for their helpful suggestions during my Ph.D. tenure. Prof. Sangtae Kim for participating in my thesis committee. Prof. Condon, Prof. Tiwari and Prof. Chover, were a source of invaluable information, and always happy to help me with nasty theory-related problems. I learned so much about research from Prof. Ioannidis, who has been influential since the beginning of my career.

The company of my office-mates, house-mates and colleagues is also deeply appreciated. My heartfelt thanks to Ramalingam for reading though of my thesis, and actually understanding Chapter 3! To my colleagues Seshadri and Srini, who have been present so many times in my practice talks, enough to understand the material better than myself! Seshadri has been my best office mate and there was no single day without stimulating discussions. I would like to thank my first roommates, Patric Calmels and Jaideep Sen, for making the first tough days in Madison much happier. George Kalfas for his positive thinking and George Yaluris for his humor and his strong opinions. Dimitris Fotakis for his true friendship, his interesting ideas on life, and his great home-made bread. Finaly, Prof. Nikos Galatsanos, for "initiating me" to the UW-Madison life.

Living in Madison and spending countless hours in the UW-campus has always been an enjoyable experience. The first drafts of this thesis were written sitting on the Union Terrace sunflower–chairs overlooking the scenic Lake Mendota. Some of the problems discussed in the thesis were attacked during long sessions with Spyros Kontogiorgis at Cafe Espresso Royale in State Street.

The graphs and figures presented in this thesis were created using several Unix$^{TM}$ tools: `xgraph`, `gnuplot` and `idraw`. I would like to thank their authors. Special thanks to Prof.

Donald Knuth and Dr. Leslie Lamport for their great TEX/LATEX package used to typeset this thesis.

Special appreciation to my brother Dimitris, for his constant support that helped me in going through difficult times. Finally, I wish to express the deepest gratitude to my parents Michael and Vassilia and the rest of my family, for the love and affection that they have bestowed on me. This thesis is dedicated to them.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Clustering in OODBMS

## 1.1 Introduction

It is widely acknowledged that good object clustering is critical to the performance of object-oriented database systems. However, in most work to date on object clustering, it is not clear exactly what is being optimized, or how optimal the proposed solutions are. In this thesis, we give a rigorous treatment of what we believe is the fundamental problem in clustering: given an object base and a probabilistic description of the expected access patterns over the object base, what constitutes an optimal object clustering, and how can this optimal clustering strategy be found or approximated?

By focusing on this underlying problem we are initially omitting many commonly considered questions that arise when implementing a clustering strategy in a running system. For example, we do not consider how the description of the expected access patterns for the database is obtained. Many techniques for this have been proposed in the literature, including maintaining usage statistics, using programmer hints, and performing data-flow analysis of the methods and type hierarchy of the system. Similarly, we do not explicitly consider the "granularity" at which the clustering is performed. The clustering could be per type, or per instance, or per method (where the objects touched by a method are treated as a single, composite object) or even some combination of these.

By omitting these questions from consideration we do not wish to imply that they are unimportant. To the contrary, we consider good answers to these questions (and others) to be an essential component of the clustering strategy of an OODBMS. However, the fundamental question of what a "good clustering strategy" means underlies all these other questions, hence is worthy of study in its own right.

Previous work on object clustering in the context of object oriented databases can be divided into several categories. Methods that use programmer hints (E and EXODUS [RC88a], Semantic Clustering [SS90]) rely on the skill of the programmer and the programmer's under-

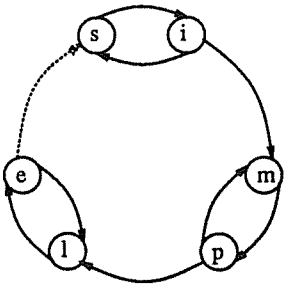



Figure 1.1: The "simple" example

standing of the problem. Syntactic methods (PS-Algol [CAC+84], LOOM-Smalltalk [Sta84]) determine a clustering strategy based solely upon the static structure of the object base. Dynamic methods (Cactis [HK89], ObServer [HZ87]) gather statistics about reference patterns, and try to find a "good" clustering based upon these statistics, while still other techniques ($O_2$ [BD90], WDFS in LOOM [Sta84]) can best be thought of as a hybrid of the syntactic and dynamic methods. The clustering criteria used in each of these types of methods are heuristics whose effect on the performance of the system is not known with any certainty. For this reason, evaluation of clustering strategies has relied upon either intuitive arguments or simulations, that mix the effects of the access stream, object base, memory size, and caching policy with the effect of the clustering strategy.

Our methodology is complementary to previous approaches; first we define a formal framework for studying object clustering in OODBMS, and next we search for provably optimal solutions within this framework. That way not only the fundamental problems of clustering will be understood, but "techniques" for solving the real problems will be derived. As it will become evident in Chapter 3, the practical problem of object clustering is very hard and many times ill-defined, so it is illuminating to have "optimal solution" even for special cases.

The following example illustrates why the clustering problem requires careful study. Suppose we have an object base that consists of six objects, which we will designate $s$, $i$, $m$, $p$, $l$, and $e$. Suppose also that there are pointers from $s$ to $i$, from $i$ to $s$ and $m$, from $m$ to $p$, from $p$ to $m$ and $l$, and finally from $l$ to $e$ and from $e$ to $l$ (this structure is illustrated by the solid arrows in Figure 1.1). Furthermore, assume that we can store any three of these objects in one disk page, and that our system can cache one disk page in main memory at any given time.

Next, suppose that when object $s$ is referenced, there is an overwhelming tendency to reference object $i$ next (say, 99 times out of 100). Also, suppose that if $i$ is referenced, there is an overwhelming chance of referencing $s$ next (again, say 99 times out of 100). Similarly,

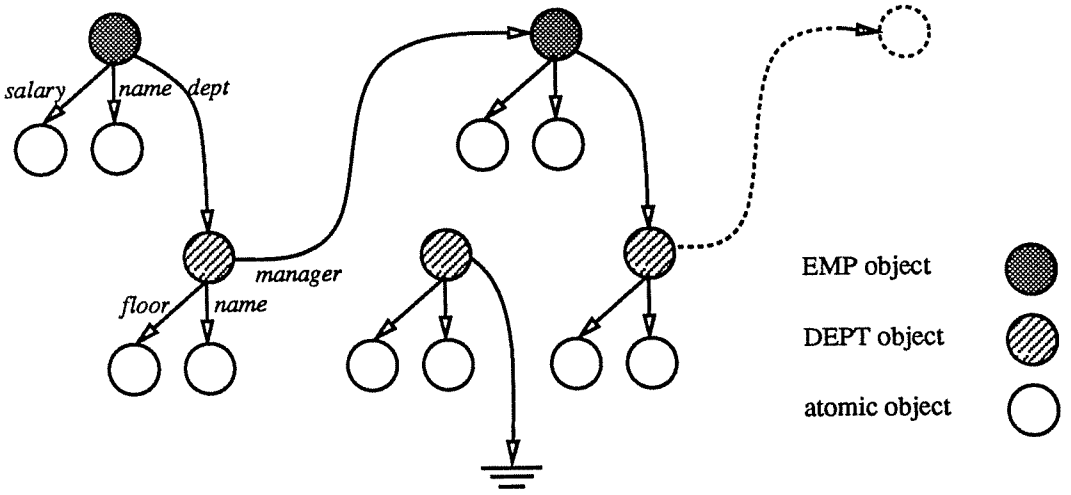



Figure 1.2: Example of an Object Graph
*The object oriented version of the traditional employee-department database. White circles represent atomic objects and the ground signs indicate null object pointers.*

assume that from $m$ we reference $p$ 99 times out of 100, and from $p$ we reference $m$ 99 times out of 100. Finally, assume that $l$ and $e$ are related in the same way (99 times out of 100, when referencing one of the two, the other is referenced next). Given this information, a probable reference string might be $w = (si)^{99}(mp)^{99}(le)^{99}$.

Now consider two clustering strategies: $C_1 = \{[sim], [ple]\}, C_2 = \{[si], [mp], [le]\}$. Under most of the clustering criteria we have seen proposed to date, $C_1$ is "optimal." In particular, $C_1$ minimizes the space occupied by the database; it minimizes the expected loading time for the database; it minimizes the number of pointers that cross page boundaries; for the reference string $w$ given above, no object is ever brought into memory but not used at some point in the future. However, $w$ will produce $1+2*98+1 = 198$ page faults under clustering strategy $C_1$, while it will produce only 3 page faults under clustering strategy $C_2$.

## 1.2 OODBMS Architectures

The notion of object abstraction was first introduced by object oriented programming languages (OOPL) like Smalltalk [Gol81]. Recently, Object Oriented Databases[1] have added database functionality to this abstraction as an attempt to increase the modeling power and the applicability of databases. The OOPL object abstraction is an extension of the data structure concept with the following basic characteristics:

[1]For a good description of current commercial and research object databases the reader is referred to [Cat91].

1. *structure*: consisting of components that can be atomic (i.e. flat attributes like integers, reals, or strings), objects (i.e. other objects), or object identifiers (i.e. "pointers" to other objects). Unlike data structures, the object state (values of the structure components) is neither directly changeable nor visible to the user.

2. *behavior*: determined by methods, predefined fragments of code that manipulate and export the object state (unlike conventional languages that allow arbitrary code to manipulate data structures).

3. *type*: prescribing the "structure" and the "behavior" of an object through the specification of its components and its methods.

4. *identity*: naming and locating an object in a manner independent of its state. Identity is typically supported identifying objects by a unique number, the object identifier (OID). OIDs are assigned by the system at object creation time, and cannot be reused, changed, or synthesized.

Conceptually, objects can be viewed as vertices of a directed and possibly cyclic graph, the *Object Graph*. The directed edges of the graph represent the object to object references and they are labelled by the names of their components (like in Figure 1.2). The graph representation of the object base is very useful in the subsequent discussion, especially in Chapter 2 where access patterns models are described.

The OODBMS supports additional database functionality:

1. *Persistence*: i.e., the ability of objects to maintain their state even after the termination of the program that has created them. Unlike OOPLs that only support as many objects as can fit into the main memory, OODBMS's provide access to large collections of objects stored in "stable" secondary storage.

2. *Storage Management*: efficient ways to represent objects in main memory, store them to disk, and distribute them to servers. That way the OODBMS relieves the clients from the burden of storing and retrieving objects from secondary storage, managing main memory, as well as the secondary memory.

3. *Concurrency Control & Recovery*: to allow concurrent accesses to the objects, and still ensure their integrity. The state of the object base is guaranteed to change only in a consistent manner, and is immune to client failures.

4. *Ad-hoc query facilities*: declarative languages to efficiently apply operations on large sets of objects.

Traditionally, Relational Database Management Systems (RDBMS) are implemented as monolithic systems communicating with the user through a query language interface. In contrast client-server architectures are commonly used in today's OODBMS's [DFMV90]. Clients run the user applications on private workstations and servers supporting the database functionality run on specialized servers with disks. Clients consist of the language run-time system, and the OODBMS run-time system necessary to communicate with the server. The server implements efficient stable storage for objects using the secondary storage, employing recovery, concurrency control, and other database protocols[2] (like versioning, indexing etc).

Except when the ad-hoc query facilities are used, the client programs access objects *sequentially* during their lifetime. One object at the time is "visited" by dereferencing object pointers and thus "walking on the object graph". When an access to some object component is needed, a memory representation of the object must be made available to the client. The client computation is suspended while the OODBMS run-time provides this representation by issuing a request to the server, and verifies that the attempted operation on the object is allowed. Depending on the concurrency control protocols used, a read or write lock may be needed to complete the operation. When the object[3] arrives from the server the suspended client computation resumes.

There are two variants of data communication protocols between client and server, depending on the unit of transfer during an object request: *Object communication* style transferring one object at the time, or *group communication* style transferring a set of objects at the time. *Paged OODBMS Architectures* use group communication and the server returns the group of objects mapped to the same page as the requested object. After the client receives the page it resumes execution, and all objects brought in with that page can be accessed with no further server intervention. Conversely, *non Paged OODBMS Architectures* use object communication and an object fault is answered with just the requested object.

Taking advantage of large memories, OODBMS's rely heavily on caches on both the clients and the servers. To decrease the overhead associated with servicing object requests, the client cache (allocated from the client process memory) reduces server requests, and the server cache (allocated at the server) reduces disk accesses. It is the responsibility of OODBMS to perform cache management and to guarantee cache consistency [CFL+91]. In addition, a page of the client cache contains the same objects as the corresponding page on the server. Redistributing objects to pages is avoided, since it can cause performance problems when writing back modified objects, and may force the client to acquire many fine

[2]It is not the scope of this thesis to discuss all aspects of client and servers, but instead, we will limit the presentation to architectural aspects related to clustering.

[3]From this point on we will use the term object not only for the the object entity but for the object representation as well.

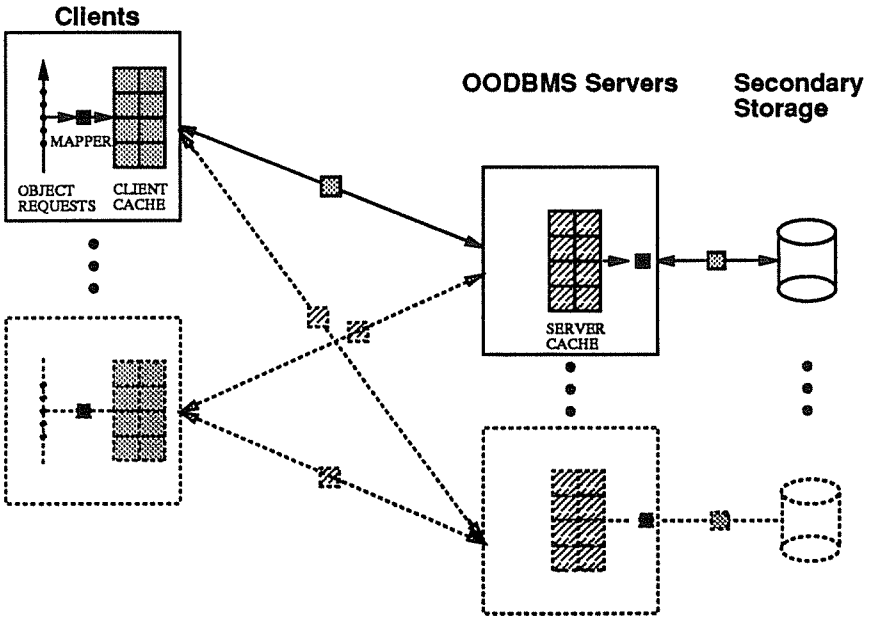



Figure 1.3: Client-Server OODBMS architecture
*The general architecture consists of a number of clients and a number of servers usually implemented by different processes interconnected by inter-process communication mechanisms.*

granularity object locks instead of fewer coarse page locks.

To summarize, a general Object Oriented Data Base Management System architecture (shown in Figure 1.3) consists of: a) data base servers serving one or more clients, b) the system's secondary memory only accessible to the servers, c) the clients, programs written in an object oriented programming language the OODBMS supports. Page caches and address translation units to map objects to pages and pages to disk blocks appear on both clients and servers.

In the rest of the thesis we will examine the storage management problems (and clustering more specifically) of the *OODBMS client-server architectures*, which use single page as the unit of communication, employ client and server caching, and do not redistribute objects between pages on the client. Our analysis is not directly applicable to less popular "object servers", which do not use page communication. At the server level, however, objects need to be retrieved from the secondary storage using pages, and our analysis applies there.

## 1.3 Storage Management Issues in OODBMS

As in relational systems, an OODBMS must deal with storage management issues [ADM+90]. Traditionally, storage management involves mechanisms and policies dealing with the defi-

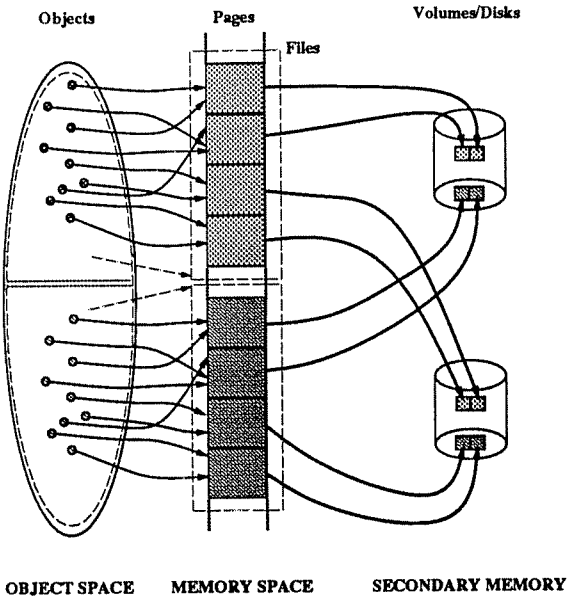



Figure 1.4: Clustering mappings
*File partitioning of the object space is shown with dotted lines & arrows, object placement with thin arrows, and page placement with thick arrows.*

nition, access, allocation, and management of object representations on main and secondary memory. The most important storage management issues are:

1. Clustering: mapping objects to main and secondary memory.
2. De-clustering: mapping pages to multiple disks/servers.
3. Memory Management: allocating and managing memory objects.
4. Index Management: associating attribute values to objects.

All the above problems are interesting from both theoretical and practical point of view; in this thesis we concentrate on the effect of clustering policies to the performance of the OODBMS, mainly with respect to their relationship with memory management.

### 1.3.1 Storage Mechanisms

In relational databases the basic unit of clustering is a file, and typically a whole relation is mapped into such a file. Such use of files is very convenient, since it simplifies addressing, locking, buffering and recovery of relations. In addition, a relation serves as a "natural partition": most of the times many members of a relation (tuples) are accessed together, either sequentially or in a group manner. The topic of physical relational database design

deals with problems such as which tuple ordering is appropriate and which attributes should be indexed (for example, Finkelstein, Schkolnick and Tiberio designed DBDSGN, a tool for physical Database Design described in [FST88]).

Influenced by RDBMS, many OODBMS or Object Stores provide files as basic coarse units of clustering. The file mechanism is a partition of the object space done on the basis of types, versions, protection, or user/programmer preferences. For example, E maps persistent collections to storage EXODUS Storage Manager files [RC88b], whereas O++ [AG91] provides a very similar abstraction named "clusters" mapped by default to Unix files. O2, CACTIS, and Observer provide segments defined as groups of pages. Finally both, persistent object stores and file-systems use files for same purpose. Typical examples are the Wisconsin Storage System (WiSS) [CDKK85] and the Exodus Storage Manager (ESM) [CDRS89] providing files as collection of objects, MNEME [MS88] supporting "segments", and finally the Unix Filesystem [MJLF84] supporting random access files.

Files alone may not be convenient objects for the system to manipulate, so they are further partitioned into pages. For example, if hardware memory protection is needed, it must be done in terms of the fixed size pages that the hardware provides rather than files. The memory management and allocation of fixed pages is much simpler than that of files, and network protocols manage packets better than streams of data. Finally, it is often possible that a given client transaction does not need to access all objects in a file, and providing only files will be waste of system resources. Files are also units of clustering for mapping pages to disk blocks; pages that belong to the same file are stored on consecutive disk blocks. Most systems allow a file to be stored in a single physical volume, although there have been some proposals to distribute pages to multiple disks or servers to exploit parallelism [BS89].

To summarize, today's OODBMS have the following basic storage management mechanisms:

1. *Files*: partitions of the object space. The partitioning process is called *file partitioning.*

2. *Pages*: partitions of objects belonging to the same file. Objects are stored in pages through *object placement* mapping.

3. *Volumes*: collections of blocks typically residing in the same physical disk. Pages are assigned to volumes by the *page placement* mapping.

4. *Caches*: collections of pages containing main memory representation of objects.

In addition to the above relatively simple clustering mechanisms, more complex schemes have been proposed in the literature. For example, multiple mappings of objects to pages

have been suggested in [HZ87], such that a different mapping can be chosen for each particular class of client applications. [Mos89] proposed partitioning the large object address space to several levels and using a hierarchical addressing scheme. In [PZ91] a non demand (prefetching) communication mechanism is suggested to request a set of objects possibly needed in the near future, instead of the single object that the client currently visits. Although all those mechanisms are certainly worth investigating, it is not certain how useful they are, especially if the limits of the simple mechanisms are not fully understood.[4] For those reasons, we decided study in depth policies related to simple clustering mechanisms. More advanced mechanisms and their corresponding policies are part of our future work.

### 1.3.2 Storage Policies

For all storage management mechanisms described in the previous section, there exist policies that designers have proposed. We will present some of them here and defer a more elaborate discussion until Section 3.5 and Section 4.3. In addition, we will be focussing on clustering policies since clustering is the main theme of this thesis.

The file partitioning is not usually up to the OODBMS. The user/programmer selects which file an object belongs to, at its creation time. Some systems allow objects to be moved between files, while other systems assign objects of the same type to the same file.

Object placement can be manually controlled by the programmer, using *clustering hints* like in the Exodus Storage Manager [RC88a] or semantic clustering [SS90]. Additionally it can be partially decided by the administrator, using more sophisticated hints (*placement trees*) as in $O_2$ [BD90]. Alternatively, it can be completely *automatic* by *graph partitioning* performed on the object graph (sometimes annotated with compiler information) like in LOOM [Sta84] or CACTIS [HK89], or on an access pattern graph like in Stochastic Clustering [TN91].

Page placement policies sometimes rely on object placement as well. Objects are ordered, placed on pages in that order, and pages are placed to disk with the same order as well. In Section 1.4 and Chapter 3 we will explain why if the disk is used by more than one client at the same time, the page placement is not as important as object placement for the overall performance of the system. In the context of operating systems and file-systems, the problem has been studied in depth for special cases of access patterns (see for example [Won80] for a good survey).

Effective de-clustering policies in the context of file-systems have been proposed in the past (for example in [SWZ92]). In the context of relational systems relation de-clustering

[4] For example, virtual memory research has shown that carefully designed demand paging usually outperforms prefetching.

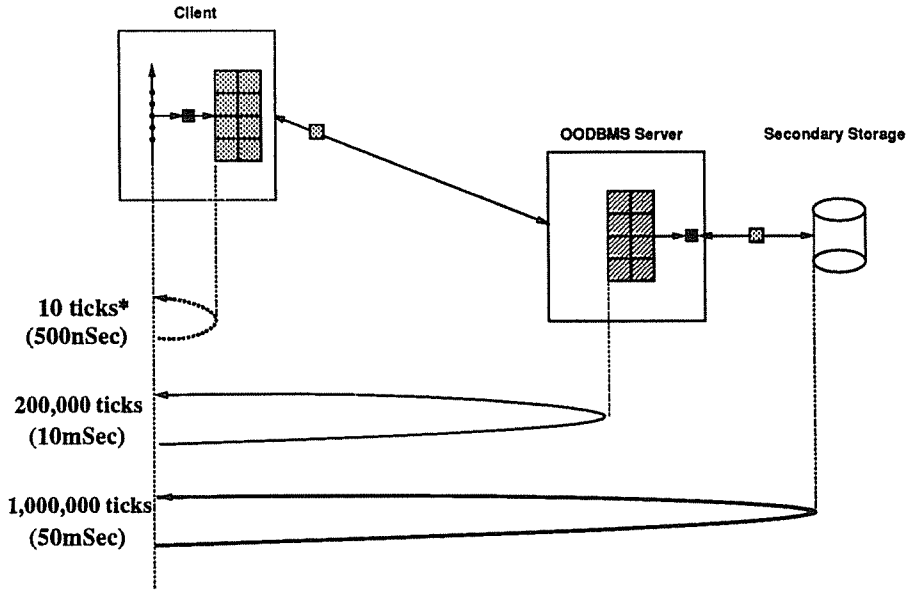



Figure 1.5: Critical paths in client-server OODBS
*The numbers shown represent lower bounds of data access costs only, assuming 20 Mips processors, a typical local area network between client/server machines, and an average seek time disk.*

is standard mechanism for parallel database architectures (as [DG92] reports). However, to the best of our knowledge, object de-clustering policies (i.e. page placement on different physical devices) in the context of OODBMS has not been studied yet, and this is subject of our future work.

## 1.4 Clustering and Performance

The choice of placing objects to pages affects the performance of the OODBMS in terms of server load, server overhead, concurrency control, and recovery. Client requests for objects are mapped to requests for pages, system resources are consumed at a rate proportional to that number, and thus, a large part of the performance of the OODBMS depends on the number of pages requested during a transaction. For example, 100 different objects accessed during some transaction 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. Our thesis concentrates on the effect of clustering on memory utilization and system load.

Paged client-server systems are subject to the same access cost characteristics as multi-

level memories, and in fact they experience even greater access differences between the different levels. Figure 1.5 illustrates the critical paths on such systems: If the page of the requested object is in the client cache, getting to an object involves performing OID to page translation (typically through a hash table), and costs approximately 10 instructions.[5] If the page is not in the local buffer (local miss), the page is requested from the server using a remote procedure call (RPC). Assuming the server runs on a separate machine and that the page is on the server cache, at least 10 ms are required to perform the RPC and send back the page. Therefore, a remote hit costs 200,000 instructions, 4 orders of magnitude more expensive than a local hit. If the page is not in the server cache and a read operation must be performed from the disk, an additional 40 msec are required to schedule and perform a random read. As a result, a remote miss costs 1,000,000 instructions and is 5 orders of magnitude more expensive than a local hit.

Object placement affects the number of pages a transaction will need during its lifetime. If the number of transaction pages is less than the size of the client cache, the computation will proceed with maximum speed. If this is not the case, thrashing may occur, requiring more page requests from the server than the total number of pages accessed by the transaction. Object placement has similar effect on the performance of the server cache, which can be viewed as an extension of the client caches. In general, depending on the access patterns, clustering policies that can fit the working set of pages in the client cache will perform well, requiring slightly more server requests than the total transaction pages. More formally, suppose that $\mu_c$, $\mu_s$ are the miss ratios of the client and server caches respectively. Let $t_{LH}$, $t_{RH}$, $t_{RM}$ the time for a local hit, remote hit, and remote miss respectively. In practice $t_{LH} \ll t_{RH} < t_{RM}$. The effective object access time $t_c$ on the client side is:

$$t_c = t_{LH}(1-\mu_c) + t_{RH}\mu_c(1-\mu_s) + t_{RM}\mu_c\mu_s \tag{1.1}$$

Substituting the parameters of Figure 1.5, we get:

$$\begin{aligned} t_c &= 10(1-\mu_c) + 10^4\mu_c((1-\mu_s) + 10\mu_s) \\ &= 10(1-\mu_c) + 10^4\mu_c(1+9\mu_s) \end{aligned}$$

A more elaborate analysis of clustering effect to the effective access time is in Chapter 3.

From the above equation it is evident that the effective object access time grows rapidly as the client cache miss ratio $\mu_c$ increases, and slower as the server cache miss ratio $\mu_s$ grows. A different interpretation of the same formula supports that in applications accessing large sets of objects, clustering might be more important than code optimizations; just avoiding one page request from the server offsets savings from tens of thousands instructions.

[5] If OID swizzling is performed, the cost can be made even lower [Mos90].

### 1.4.1 Overview of the Thesis

The first chapter presented the most common OODBMS system architectures and introduced the clustering problem. The second chapter deals with modelling issues of access patterns. We define three mathematical models suitable for access patterns that appear in OODBMS, and we give the intuition for why such models will be useful for clustering. Next, we examine closely the multi-client ODBMS system model, suggesting that clustering can be reduced to two independently solved sub-problems. Each sub-problem is an instance of the "Abstract Clustering", a formal clustering problem we formulate and solve in the rest of the chapter. Chapter Four presents our experimental testing methodology for performing systematic performance evaluation; we evaluate several clustering policies proposed in the literature, and the stochastic clustering algorithms derived from the results of Chapter Three. In the last chapter, we summarize the thesis results, suggest new clustering heuristics, and discuss remaining open problems.

# Chapter 2

# Access Models for Object Oriented Programs

In this chapter we examine the access generation process of object oriented applications, and investigate the impact of object oriented semantics on access patterns. We have found that accesses experience locality in both the time dimension (temporal locality) and the space dimension (structural locality). Based on those observations we propose several access models with varying degrees of complexity and accuracy. Those models are useful in defining and analyzing the clustering problem presented in Chapter 3.

## 2.1 Introduction

Attempting to study management issues of data storage facilities, be they file-systems, relational database engines, or object stores, one faces a difficult question: how do the clients of those systems use data, and what are the interesting usage patterns? Furthermore, to perform any analytical study formal models need to be developed.

The notion of access patterns (or more accurately access models) is not unique to object oriented systems. Models of some flavor have been proposed in the past to describe and characterize the behavior of clients in terms of code execution, data accesses, or I/O. Those models were valuable to study problems like caching [Aga88], paging [ADU71], register allocation [Sto87], or disk placement [Won83]. In the case of paging, for example, interesting properties of accesses like temporal/spatial locality were discovered, and used to develop efficient algorithms for managing virtual memories (like WSCLOCK [CH81]).

Scholnick in [Sch77] suggested some very simple access models for navigations of hierarchical structures, by classifying accesses depending on the relative ranks of nodes visited. Benzaken proposed a probabilistic model of method usage that led to an algorithm for producing placement trees for object clustering [BD90]. The concept of a formal object access

model, which will be used in Chapter 3 for clustering purposes, is informally defined here:

> *The Object Access Model is a a mathematically convenient way to describe accesses of object oriented programs.*

A good model has the following desired properties:

1. *Abstraction:* to express characteristics relevant to the studied problem (i.e. the storage allocation).
2. *Compactness:* to have a "small" number of parameters.
3. *Descriptive power:* to describe a *wide range of access behaviors* for specific values of its parameters.
4. *Insensitivity:* "small changes" of the modeled behavior should not result in big changes of the values of its parameters.

As it will become evident in Chapter 3, static clustering depends much on the stationary properties of object accesses. More specifically, on their *persistent correlations.* An access correlation between object $x$ and $y$ (denoted as $x \Rightarrow y$) exists when accessing $x$ at any point of time causes an access to $y$ after a short time with high probability. Having those requirements in mind, we start a search for access models by first examining the access generation process of object oriented programs. As we will see some simple statistical models are appropriate for modeling access patterns of object oriented systems, both at a detailed microscopic level, and at a more coarse macroscopic level.

## 2.2 The Access Generation Process

Depending on the time frame in which references are observed the generation process can be viewed at several levels. The *microscopic view* of the process, described next, reveals the reference by reference behavior, as opposed to the *macroscopic view*, developed later, which depicts longer term effects. A closer view of an object oriented application reveals a computation being performed by accessing a subset of the object space during a transaction. Object accesses are constrained to obey object oriented protocols like encapsulation, and must preserve object identity. As we will see next, those constraints cause some interesting properties in the access patterns.

The access generation process can be better described in terms of the "OODBMS state machine" shown in Figure 2.1, supporting the object oriented concepts and semantics [Bud90].

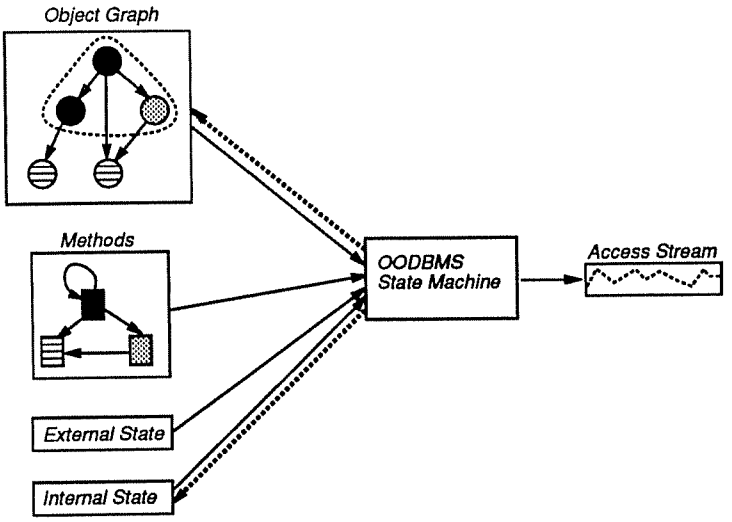



Figure 2.1: The Object Access Generation Process
*The OODBMS abstract machine performs a traversal in the object graph by issuing object requests dictated by the currently executing method. The behavior of the machine is modified by the internal or external state.*

For this discussion, the most relevant such concepts are the object graph (a result of typing), the methods (providing encapsulation and typing), and object identifiers (supporting identity). The machine has access to the *object graph* (defined in Chapter 1), to the *external state*, and to the *local state*. The "program" executing on the machine consists of a set of methods per object type (explained next). The user/system input are considered as external state, and program variables as internal state.

The state machine operates as follows:

> At any time there is the notion of the "current object" $x$ whose method $M_x$ is executing.[1] $M_x$ it is defined as a sequence of "instructions" which operate on the state of $x$, and may accept a set of parameters. The instructions include object component fetches, method calls, branches, and iterations.
>
> During its execution, $M_x$ may "call" some other method $M_y$ of an accessible object $y$ (the arrows between the method "boxes" of Figure 2.1 represent method calling sequences). The method invocation process is recursively applied until a method that does not call any other method is reached, thus performing an object state dependent traversal of the method call graph.

Strictly speaking, during the execution of $M_x$ the only object representation accessed is that of $x$. In practice $M_x$ may be allowed to access other objects, like the components of $x$

[1] In the beginning of computation, the object base is "opened" and its root becomes the current object.

or method parameters. Method parameters will vary at run time, and can be ignored in this analysis. Not every method accesses all object components, so let $G(M_x, x)$ denote the set of objects structurally related to $x$ that method $M_x$ may access with non zero probability (in Figure 2.1 those objects are enclosed by a dotted curve assuming that the current object is the root of the graph shown).

Typically, $G(M_x, x)$ is stored in method or program variables, at any time, and any member of it is a candidate for access depending on the program flow. For example, a variable *may not be accessed at all* if it is inside an "if statement", or it may be *repeatedly accessed* if it is inside a "loop construct". Therefore, execution of method $M_x$ applied to $x$ induces intra-method accesses on $x$ and a fixed subset of its components $G(M_x, x)$.

While executing $M_x$ another method $M_y$ can be called, resulting object accesses to $G(M_x, x)$ followed by accesses to $G(M_y, y)$. If $y$ is structurally related to $x$, $G(M_y, y)$ objects will be structurally related to $x$ too. If $y$ is not structurally related to $x$ it means $y$'s identity may very well vary at run time. Such cases will be ignored during this analysis, since if they vary they will not affect the stationary behavior, and if they do not vary, we have no way to know what values they will take. Therefore, a method $M_x$ calling $M_y$ induces inter-method accesses to structurally related objects $G(M_x, x) \cup G(M_y, y)$.

Real object oriented applications may have different access behavior from the one illustrated before. Internal or external state may modify the client behavior, whereas global variables or variables passed as arguments may generate additional accesses. We expect however, that all those factors will not qualitatively change the fundamental properties of object oriented accesses.

To summarize, we expect that object oriented accesses have the following characteristics:

- *restricted scope*: when running a method on some object a "small" set of objects can be possibly needed in the near future.
- *structural locality*: a structurally related object to the current one will be needed next with high probability, and accesses will appear as "walks" on the object graph.
- *temporal locality*: caused by a small group of objects repeatedly accessed, as it in the case with traditional program accesses.

Next we will see how those properties show up in the proposed models.

## 2.3 Access Models for Sequences

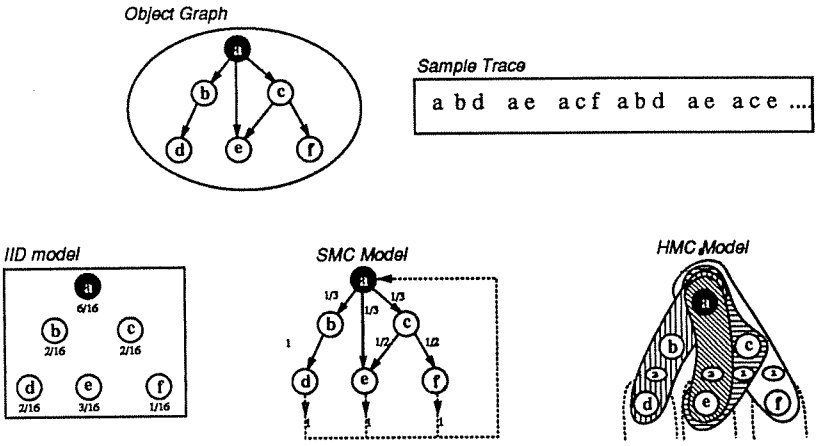



Figure 2.2: Object Access Models
*all models are described using a graph notion and they correspond to the object graph and the sample trace given here. The numbers denote probabilities or access counts (if greater than 1).*

As explained before, OODBMS clients that use a persistent programming language interface, *navigate* the object graph,[2] producing sequences of object requests. Next we will examine some standard statistical models based on stochastic processes.

Let the set $S$ represent a population of $N$ objects denoted by integers:

$$S = \{1, 2, 3, \ldots, N\}$$

The term "request stream" or "trace" refers to a sequence of $n$ object requests issued by some application:

$$(X_n) = x_1, x_2, \ldots, x_n$$

### 2.3.1 The Independent References Model

The independent references model, known formally as IID (sequence of Independent and Identically Distributed random variables), describes the request stream by a random process. Any time $t$ the probability an object $x$ appears in the trace is fixed and only depends on $x$:

$$\pi(x) \equiv Prob\{X_t = x\} \tag{2.1}$$

$$\text{where } \sum_{x=1}^{N} \pi(x) = 1$$

[2]The notion of the object graph was introduced in Chapter 1.

The IID model can be expressed in a convenient form as a $N$ row vector of probabilities:

$$\vec{\pi} \equiv [\pi(1)\pi(2)\ldots\pi(N)]^T \tag{2.2}$$

To estimate the IID model from a sample trace $(X_n)$ we use the following unbiased estimator [All78] for each vector element:

$$\hat{\pi}(x) = \frac{\sum_{t=1}^{n} \delta_{xX_t}}{n} \tag{2.3}$$

where

$$\delta_{xy} = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

Figure 2.2 shows an example of IID model derived from a client performing a random walk on a specified object graph.

### 2.3.2 Simple Markov Chain Model

The Simple Markov Chain model (SMC) describes the request stream as a Markovian Process. At any time $t$ the probability an object $y$ appears in the trace is fixed and only depends on $y$ and the previously requested object $x$:

$$P(x,y) \equiv Prob\{X_t = y | X_{t-1} = x\} \tag{2.4}$$

where

$$\sum_{y=1}^{N} P(x,y) = 1$$

The SMC model can be expressed in a convenient form as a $N$ by $N$ matrix of conditional probabilities:

$$P \equiv \begin{bmatrix} P(1,1) & P(2,1) & P(N,1) \\ P(1,2) & P(2,2) & P(N,2) \\ & \cdots & \\ P(1,N) & P(2,N) & P(N,N) \end{bmatrix} \tag{2.5}$$

If the chain is aperiodic and contains a single set of accessible states (all states are recurrent, i.e., mutually reachable), then there exists a unique stationary distribution for the probability $\pi(x)$ to find object $x$ in the sequence at some arbitrary time. As in the IID case,

this probability can also be expressed in vector form $\vec{\pi}$ and it satisfies the following matrix equation:

$$P\vec{\pi} = \vec{\pi} \tag{2.6}$$

To estimate the SMC model from a sample trace $(X_n)$ we can use the following unbiased estimator [All78] for each matrix element:

$$\hat{P}(x,y) = \frac{\sum_{t=2}^{n} \delta_{xX_{t-1}} \delta_{yX_t}}{\sum_{t=1}^{n-1} \delta_{xX_t}} \tag{2.7}$$

It is easy to show that the estimated IID probability vector is the same as the stationary probability of Equation 2.6 calculated from the estimated probability matrix of the SMC model.

The SMC matrix $P$ can also be represented as a labelled directed graph $G(P)$: The vertices of the graph are the objects of $S$. For each non zero entry $P(x,y)$ of the SMC matrix, there exists an edge

$$x \overset{P(x,y)}{\longrightarrow} y$$

i.e., an edge from node $x$ to node $y$ labelled by $P(x,y)$. In that case, the trace is considered as the result of a random walk on graph $G(P)$, and it expresses the "average" behavior of the application as seen in the sample trace. Figure 2.2 shows an example of SMC model derived from a client performing a random walk on a specified object graph.

### 2.3.3 Higher Order Markov Chains

A generalization of the SMC model is a k-th order stochastic process or $HMC_k$. At any time $t$ the probability an object $y$ appears in the trace is fixed and only depends on $y$ and the $k$ recently requested objects $x_1, \ldots, x_k$:

$$P(x_1, \ldots, x_k; y) \equiv Prob\{X_t = y | X_{t-k} = x_1, X_{t-k+1} = x_2, \ldots X_{t-1} = x_k\} \tag{2.8}$$

and

$$\sum_{y=1}^{N} P(x_1, \ldots, x_k; y) = 1, \ \forall x_i \in S$$

HMC models are qualitatively no different from SMC. The state space consists of the the Cartesian product $S^k$, and each state is now denoted by a vector of $k$ objects instead of just one, representing the last $k$ requested objects. Similar to SMC, a stationary probability for the new states is defined, and satisfies the same equation as 2.6.

To estimate the $HMC_k$ model from a sample trace $(X_n)$ similar statistical estimators can be used, but keeping track of a much larger set of states ($N^k$ to be exact) is necessary now:

$$\hat{P}(x_1, \ldots, x_k; y) = \frac{\sum_{t=2}^{n} \prod_{j=1}^{k} \delta_{x_j X_{t-j}} \delta_{y X_t}}{\sum_{t=1}^{n-1} \prod_{j=1}^{k} \delta_{x_j X_{t-j}}} \tag{2.9}$$

The HMC model can be represented as a labelled directed hypergraph $HG(P)$: The vertices of the graph are members of $S$. If we denote:

$$\vec{x} = x_1, x_2, \ldots, x_k$$

then for each non zero entry $P(\vec{x}; y)$ of the HMC there exists a directed hyper-edge:

$$(x_1, x_2, \ldots, x_k, y)$$

labelled by $P(\vec{x}; y)$. In the case of HMC, the trace is considered as the result of a random walk on $HG$, and the average behavior of the application is more accurately expressed by HMC than with $G(P)$. The hypergraph of Figure 2.2 shows part of an $HMC_3$ model derived from a client performing a random walk on a the object graph shown.

### 2.3.4 Probabilistic Set Model

The Probabilistic Set Model (PSM) is a compromise between IID and SMC/HMC models. PSM does not precisely specify the order of accesses as SMC/HMC do, but on the other hand it does not condense all information to simple probabilities like IID. Let us assume that the client can issue a set of $m$ queries $S_1, S_2, \ldots, S_m$ each one accessing a fixed subset of objects from $S$. We will assume that

$$S = \bigcup_{i=1}^{m} S_i$$

PSM models the request stream as a random sequence of queries, each query $S_i$ appearing with a fixed probability $\pi(S_i)$ and accessing all of its objects in an unspecified order:

$$(X_n) = S_{i_1}, S_{i_2}, \ldots S_{i_{k'}} = \{x_{i_1 1}, x_{i_1 2}, \ldots\} \ldots \{x_{i_2 1}, \ldots\}$$

In this model the order those objects appear in the trace is irrelevant. Once a query is selected all of its objects will be requested. As with HMC, the PSM model can be viewed as a hyper-graph, with nodes being objects from $S$ and hyper-edges designating the queries with weights equal to the query probabilities.

The (object) stationary probability of this model expressing the probability an object $x$ appears in the trace at some arbitrary time, can be defined by the following formula:

$$\pi(x) = \sum_{i=1}^{m} \frac{\pi(S_i)1(x \in S_i)}{||S_i||}$$

where the indicator function $1(x)$ is defined as:

$$1(x) = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

To estimate a PSM, the trace is first partitioned to locate the subsets $S_1, S_2, \ldots, S_m$. Then the trace can be viewed as a sequence of subset requests instead of object requests, and the probability of each subset can be estimated as in the IID case.

## 2.4 A Macroscopic View of Access Patterns

Having presented the microscopic view of the generation process, that focuses on the reference by reference properties of access patterns, it is time to look at the *macroscopic view.* We will examine the result of a series of references on a subset of the object space, i.e., references being made within a window of time and space. We will investigate three interesting cases: Induced access patterns on files containing interlinked objects, access patterns when hierarchical structures are traversed, and finally access patterns resulting during path expression evaluation. In all cases, the access models defined before are appropriate.

### 2.4.1 Inter-File Accesses

As we have seen in the first chapter, file clustering mechanisms partition the object space in some arbitrary manner. Pointers between objects from different files may exist and object accesses originating in one file will induce accesses on the other file. Since clustering objects between different files is clearly not possible, access models involving inter-file object references are not useful. Understanding access patterns induced on a group of objects by accesses originating on some other group of objects, is very appropriate in this case.

Suppose we have two sets of objects $S_1$ and $S_2$ that belong to two different files $F_1$ and $F_2$. Some objects of $S_1$ point to the objects of $S_2$, and this is the only away $S_2$ can be accessed. The induced access pattern on $S_2$ in general depends on the object graph structure of $S_1$. All accesses originate from $S_1$ access objects there and finally access some objects in $S_2$. Depending on how many objects are accessed in $S_2$ "per visit", we distinguish between two cases: Shallow accesses (few objects per visit) and deep accesses (many objects per visit).

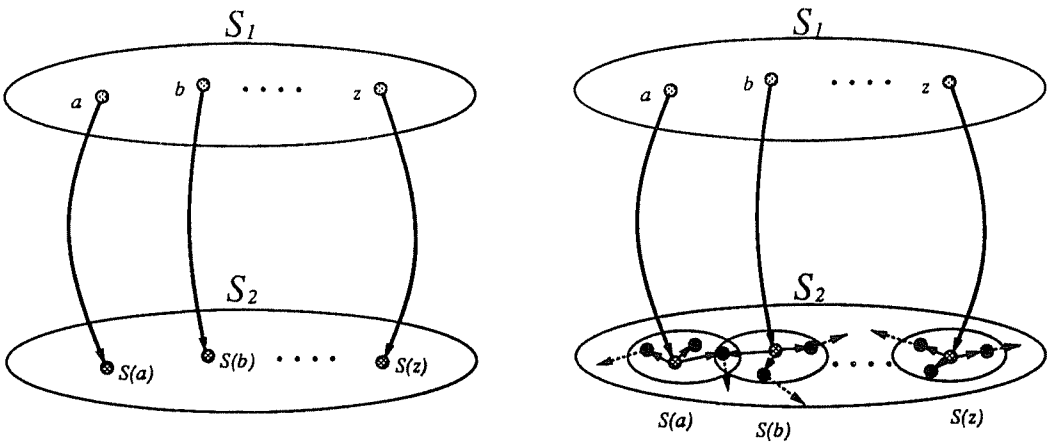



Figure 2.3: Inter-File Access Patterns
*Accesses on $S_1$ induce shallow (left) or deep (right) Access Patterns on $S_2$.*

If $S_1$ is a search structure indexing the members of $S_2$ then accesses will be shallow. Random access patterns can be expected for "point queries", and sequential access patterns for "range searches". The static behavior of the accesses can clearly be modelled by IID in case of point queries, and by SMC/HMC/PSM models for range queries. If accesses are deep, then $S_1$ affects the entry/root objects of $S_2$.

### 2.4.2 Navigation of Hierarchical Structures

Suppose that objects belong to a hierarchical structure, i.e. a graph that has objects arranged mostly in levels with respect to some root object. If that hierarchical structure is traversed repeatedly, what kind of accesses do we expect to see? Intuitively, the objects higher in the structure will be more frequently accessed than the ones at the lower levels of it. Assuming the observation window is more than the "depth" of the structure, the serial access dependencies will disappear. Objects will appear in that window with their stationary probabilities. In this case accesses can be satisfactory modeled using IID, with high probabilities assigned to nodes with high rank. Figure 2.4 illustrates the effect of IID access as a result of navigating an object graph used in our clustering experiments.

### 2.4.3 Path Expressions

In many cases object oriented applications evaluate "path expressions", having the form:

$$x.a_1.a_2.a_3....a_n \qquad \text{or}$$

$$x \rightarrow a_1 \rightarrow a_2 \rightarrow a_3....a_n$$

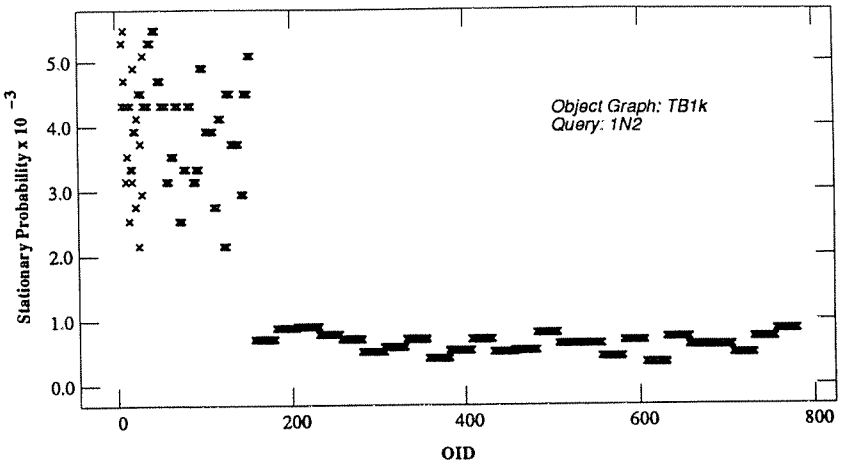



Figure 2.4: Stationary probabilities accessing a hierarchical structure
*the objects are arranged in BFS order resulting in clustering of probabilities with respect to the rank (i.e. range in BFS order).*

where $x$ is some object and $a_i$ is the name of a component, pointer, or method. Those expressions are usually evaluated in a left-to-right manner[3] i.e. starting from $x$ and going though the path from left to right. In a static environment, this style of expression evaluation corresponds to a navigation of the object graph starting from $x$ and following a fixed path ($a_i$'s are usually fixed).

Suppose we are given a set of path expressions:

$$Q = \{p | p = x.a_1.a_2.a_3....a_{n_x} : x \in S\}$$

together with their probabilities $P(p)$ for all $p$. PSM is a very appropriate access model to capture the fact that fixed groups of "connected" objects are collectively accessed. Each query $S_i$ of PSM contains all objects encountered in the path expression $p$ starting from each possible object $x$, and has probability $P(p)$. As opposed to an SMC/HMC model that can keep track of access order, the PSM will fail to capture the exact object sequences. This is not necessarily important to model because in a large window of accesses the order of path components hardly matters.

## 2.5 Practical Considerations

When it comes to using access models in real systems two practical questions arise: How should those models be stored to minimize the storage overhead, and how would those models be obtained? Next, we will discuss some ideas for storing and estimating access models. A

[3]assuming that no "index" has been used to bypass this evaluation.

thorough study of such problems is necessary for applying many clustering ideas to real systems, but are unfortunately beyond the scope of this thesis.

### 2.5.1 Storing Access Models

The IID model is not that hard to store since it only requires a single variable per object ($\pi(x)$). This can be easily implemented using an extra "hidden field" in the object representation, exclusively used and maintained by the system. Alternatively, $\pi(x)$ can be stored in the OID to physical address translation data structure if logical OID scheme is used. Each method has its advantages and disadvantages. The hidden field method causes object updates that need to be logged by the system and cause extra overhead to the system, but it offers a better integration. The translation table eliminates the update problem, but it requires changes to the translation mechanisms.

When it comes to SMC models, storing tables of $N^2$ elements seems prohibitive. In practice however some characteristics of the access patterns can be exploited; the restricted scope of accesses make the SMC matrix very sparse, and structural locality causes most of the objects accessed after some other object to be structurally related. Those observations give rise to access models represented by edges and nodes of the object graph: transition probabilities can be stored inside the object representation, one for every "object pointer", and stationary probabilities can be stored with the object as with the IID case. For example, if object $x$ has a pointer $x.a$ to object $y$, then an additional "hidden field" $x.a_p$ stores the transition probability $P(x, y)$. The above analysis explains the relative success of object graph based statistics kept by the CACTIS system [DK90].

### 2.5.2 Estimating Access Models

As with any modeling problem, great attention must be given to the quality of its estimated parameters. If sample traces are to be used for *statistical inference* of access models, they must be sufficiently long to ensure that the estimated models are statistically valid. A simple way to achieve this is by using traces collected over a representative period of the client behavior, especially when it is known that clients exhibit periodic behavior.

*Pearson's chi-square test for goodness* is a well known statistical test used in model estimations (also used in [Hol90] for parallel program access models). Suppose an IID model ($\vec{\pi}^0$) is derived from a sample trace, and we want to know if the testing trace ($Y_n$) could have been produced by the same model. Let $n$ be the size of ($Y_n$), and $\hat{\vec{\pi}}$ the access model estimated from ($Y_n$). The null hypothesis $H_0 : \vec{\pi} = \vec{\pi}^0$, i.e. that all objects appearing in the

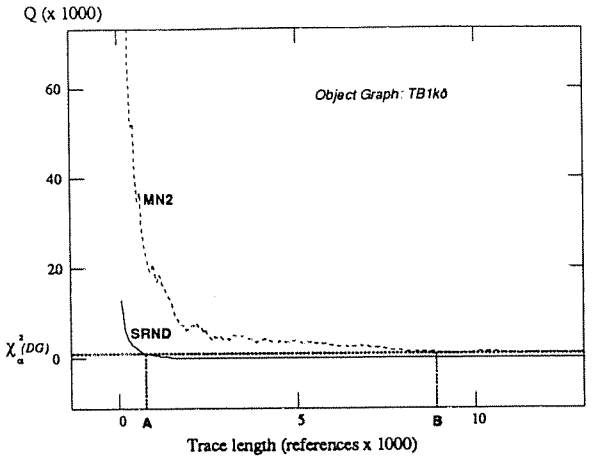



Figure 2.5: Statistical Confidence during Model Estimation
*The SRND is a real IID access pattern as opposed to MN2, which is a stochastic access pattern. For the same confidence value $\alpha$, the estimation of the SRND model can be achieved much faster than MN2 (point A as opposed to point B in the graph).*

trace ($Y_n$) have the same probability as in the tested model $\vec{\pi}^0$. The test statistic:

$$Q = \sum_{i=1}^{N} n \frac{(\hat{\vec{\pi}}_i - \vec{\pi}_i^0)^2}{\vec{\pi}_i^0} \tag{2.10}$$

has asymptotically a $\chi^2$ distribution with $DG = N-1-d$ degrees of freedom (assuming there are $d$ zero entries in $P^0$) $H_0$ is rejected if $Q > \chi^2_\alpha(DG)$, where $\alpha$ is the desired significance level of the test.

In the case of SMC, the problem of deciding if a sample trace (with an estimated transition probability matrix $\hat{P}$) could have been produced by a first order Markov Chain with transition probability $P^0$, is equivalent to testing the null hypothesis $H_0 : P = P^0$. The test statistic:

$$Q = \sum_{i=1}^{N} \sum_{j=1}^{N} n_i \frac{(\hat{P}_{ij} - P_{ij}^0)^2}{P_{ij}^0} \tag{2.11}$$

has asymptotically a $\chi^2$ distribution with $N(N-1) - d$ degrees of freedom (assuming there are $d$ zero entries in $P^0$), and $n_i$ is the number of times object $i$ appears in the sample trace. As before, we shall reject $H_0$ if the value of the test statistic is larger than $\chi^2_\alpha(DG)$.

The graph of Figure 2.5 illustrates those tests, as applied in our experiments for two different workloads (and hence two different access patterns) from the Tektronix Hypermodel Benchmark we used in our simulations.[4] The test statistic decreases as the number

[4]For the full description of the benchmark please refer to Chapter 4.

of references used to estimate the model increases. To gather access pattern of a given confidence value ($\alpha$), different trace lengths are required depending on the workload. Random workloads like SRND can be estimated fast, as opposed to stochastic workloads (like MN2) that converge slower.

One way to use those methods during the trace acquisition, is to periodically check if the estimated model so far could have produced a (random) subpart of the sample trace. If this is true, the statistics gathering can stop, otherwise the process continues by incorporating statistics from the recent part of the sample trace.

# Chapter 3

# The Abstract Clustering Problem

Object clustering has long been recognized as being important to the performance of object bases, but in most work to date, it is not clear exactly what is being optimized or how optimal the solutions obtained are. In this chapter, we give a rigorous treatment of a fundamental problem in clustering: given an object base and a probabilistic description of the expected access patterns, what is an optimal object clustering, and how can this optimal clustering be found or approximated? We present a system model for the clustering problem and discuss two models for access patterns in the system. For the first, exact optimal clustering strategies can be found; for the second, we show that the problem is NP-complete, but that it is an instance of a well-studied graph partitioning problem. We propose a new clustering algorithm based upon Kernighan's heuristic graph partitioning algorithm, and present an experimental comparison of this new clustering algorithm with several previously proposed clustering algorithms.

In this chapter we first present a system model for OODBMS, and from that we derive two reasonable clustering problems. We show that both problems are instances of the same abstract clustering problem partially studied in the past. Next, we discuss two interesting special cases of abstract clustering using two rigorous models for access patterns in the system. For the first, the *Independent Identical Distribution* (IID) model, exact optimal clustering strategies can be found. While the IID model is too simple to model detailed client access patterns, as we shall demonstrate, it can be a useful tool in designing clustering strategies for client-server environments. For the second, more powerful access model, the *Simple Markov Chain Model*, we show that the clustering problem is NP-complete; however, in this case, we show that the clustering problem is an instance of a well-studied graph partitioning problem for which good heuristic solutions are known. Finally, we propose two new clustering algorithms based upon the clustering theory developed.

## 3.1 OODBMS System Model

The basic system architecture of a *paged client-server OODBMS* was discussed in Chapter 1. In this section we will derive a system model from that architecture for the purposes of clustering. The model consists of:

- A data base server serving one or more clients.
- The system's secondary memory only accessible to the server.
- A number of clients running the applications requesting objects from the server.

In addition, clients and the server employ page caches and address translation, maps fully explained below. Figure 3.1 shows our model, which is heavily based on the OODBMS architecture shown in Figure 1.3.

Each client consists of the *client application*, a single-threaded program written in an object oriented programming language the OODBMS supports. The program accesses objects "sequentially"; only one object representation at the time is necessary for the client computation to proceed. The sequence $X_k(t) : t = 1, 2, \ldots$ denotes the *request stream* of object accesses being made by client $k$. Each object is *mapped* to a page through the mapping function $D_\pi$, thus generating a page request stream denoted by $Y_k(t) = D_\pi(X_k(t))$. A *demand page cache* contains a set of pages recently needed and it is managed by some page replacement policy. The page request stream is "filtered" through that cache, thus generating a *page fault request stream* denoted by $Y'_k(t)$. Every time $t$ the current page $Y_k(t)$ is not present in the cache, a page request $Y'_k(t) = Y_k(t)$ is made to the server. The client cache models systems that access objects from the buffer pool (through an OID interpreter), and systems that store objects in virtual memory using pointer swizzling.

The server accepts the *global request stream* (denoted by $W(t)$), a sequence of requests coming from $k$ different clients. We will assume that any given time only one client can send a request to the server. If this is not the case, the tie can be broken using some arbitrary ordering, like the ordering of messages. The server employs a *page cache* that contains a set of recently requested pages, and attempts to answer the client requests out of that cache. The global request stream is filtered through the server cache, generating a new stream of page faults $W'(t)$ that must be satisfied from the disk. To do so, the location of the page on disk must be found using a *page to block mapping* ($D_\sigma$), thus generating the new stream $V(t) = D_\sigma(W'(t))$. The server employs a *queue* that rearranges disk page requests, in a way that depends on the disk and the client scheduling policy. Finally, the rearranged requests coming out of the queue (denoted by $V'(t)$) are send to the *disk*. Here we will not consider

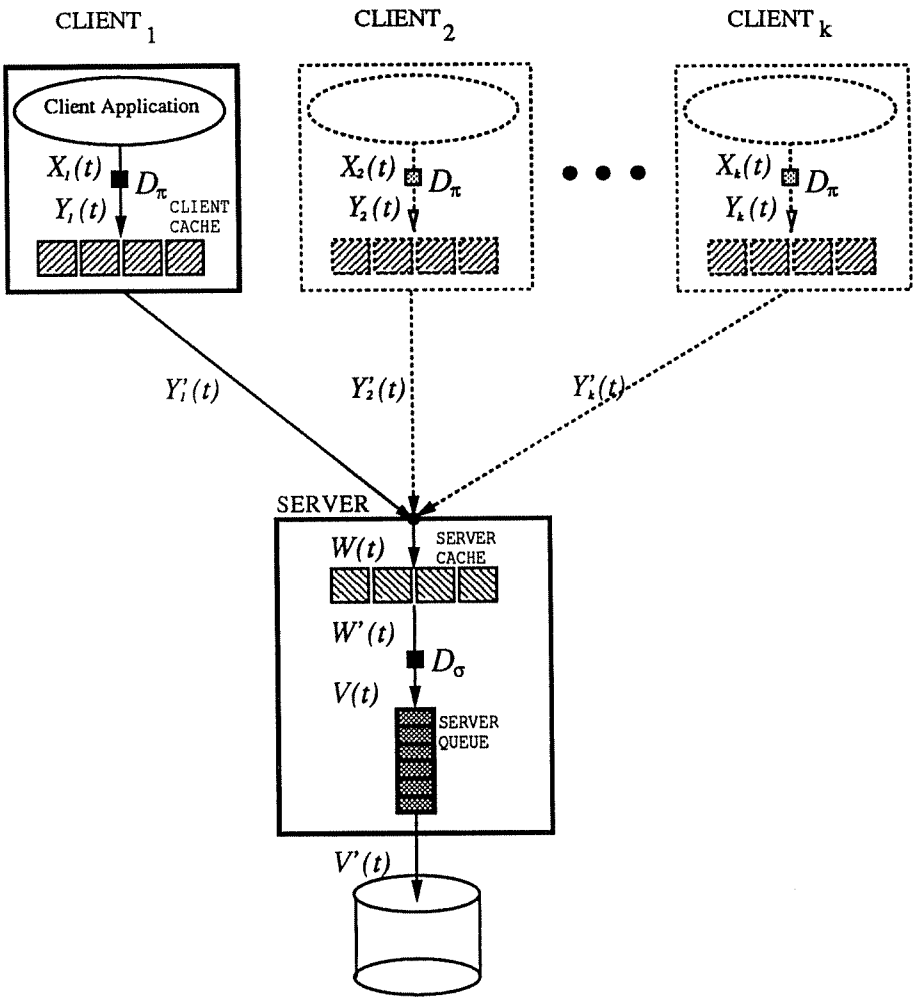



Figure 3.1: The OODBMS System Model

the case of multiple disk devices. There are some interesting problems related to distribution of objects on multiple disks (de-clustering), but they are out of the scope of this thesis.

The scope of the above model was intentionally limited to the data access aspects of OODBMS. Database protocols like concurrency control and recovery clearly affect the behavior of the system, but are omitted here so we can concentrate on the effect of clustering on the performance of the OODBMS. In addition, there are a few good reasons for ignoring them. First, logging typically uses a separate physical volume and in this case, it will not interfere with normal disk activity. Second, when logging is performed at the end of the client transaction, logging activities may not interfere with the read phases of the transaction. The read phase typically performs single page *demand reads*, which can be affected by clustering. In contrast, during the write phase *bulk writes* can be performed to send all dirty pages to the server and the disk. Bulk object writes make clustering less important, because they do not have to be performed in any specific order, and they can be efficiently done using scatter write techniques. Finally, data contention and locking may create unpredictable delays in answering page requests, for example when a requested page is currently locked by some other client. Those delays cannot be reasonably modeled to be accounted for when the static clustering mapping is computed.

From this description it becomes clear that the *page* is the unit of interaction between most of the system components (caches, queues, and disks), whose behavior depends on the properties of the access patterns. In addition, the system designer has two ways to affect the performance through clustering: the object to page mapping (object placement) and the page to disk block mapping (page placement). Next, we will elaborate on the effect of client caching and multiple clients to the access patterns observed in various parts of the system. Our objective is to derive reasonable clustering objectives for a multi-client OODBMS.

### 3.1.1 Model Properties

If we examine closely the above model, several observations can be made regarding the properties of the access patterns in several places of the system model (the reader is referred to the picture of Figure 3.1 for the subsequent discussion).

First, *object mapping and client caching does not change the client access patterns qualitatively.* That is, accesses observed after the cache (or the mapping), will appear to follow a similar model as the original accesses.

The client cache "sees" a different access stream ($Y_j(t)$) than the one the client originally produces ($X_j(t)$), due to the object placement mapping $D_\pi$. The statistical characteristics of the transformed access pattern do not qualitative change however: as we show in Appendix A.1 an IID (SMC) access pattern on objects remains IID (SMC) access pattern on pages. In

addition, it has been shown in [CD73] that page caches using traditional replacement policies like LRU, MRU, or FIFO, do not qualitatively change the access patterns.[1] For example, an LRU cache (typically used by OODBMS systems and virtual memories) will tend to keep "hot" pages, and thus filter out them from $Y_j'(t)$. In the steady state, an LRU cache of size $L$ will contain the $L$ hottest pages of $Y_j(t)$, and as a result, requests for the rest $M-L$ pages will pass through the server cache producing an IID page fault stream [CD73] ($M$ denotes the total number of pages).

Second, *multiple clients and server queueing generate uncorrelated access patterns on the server.* Suppose the clients run potentially different applications and there is no synchronization between them. $W(t)$, the global request stream at the server, will be a randomly merged sequence of requests made by the clients. If we examine a small (compared with the number of clients) window of consecutive pages from $W(t)$, we cannot find two requests from the same client (since clients make demand reads and are single threaded). As a result, there can be no dependency between pages of that window, and $W(t)$ will behave as an *uncorrelated (random) process.* Should the number of clients be small, or the traffic they generate be small,$W(t)$ will be no longer random process.

The server cache and the server queue will further "randomize" the pages observed in the access stream directed to the disk ($V(t)$). As discussed in [All78] the stream will contain page requests with order depending on the effectiveness of the server cache, and on the queueing policy used (FIFO, Priority based, etc.). Consequently, the disk stream $V(t)$ will appear to behave like *an uncorrelated process* as well.

Third, *Server Caching does not change the picture.* The server cache filters the global request stream $W(t)$, and generates a server cache miss stream $W'(t)$. Depending on the way this cache is designed and managed, the access patterns characteristics of $W'(t)$ may be different. The most common methods are global management and local management.

*Global Management* manages the cache by treating it as a global buffer pool, i.e. it makes no use of the origin of the client requests. This policy corresponds to global memory management commonly used by operating systems. Since the global request stream will be IID, $W'(t)$ will be the result of an IID stream filtered through the server cache. As with the case of client cache, $W'(t)$ will also be an IID stream with a different probability distribution.

Alternatively, the cache may be partitioned statically or dynamically to different clients, and each partition can be managed independently. Ignoring how the size of its partition is decided (for example as proposed in [SWT89]) the net effect of local policies is that the server cache partition allocated to requests from a client acts as an *extension of the client*

[1] More generally, if the original access stream is stationary, and the memory management policy makes decisions solely based on a finite portion of the stream, the cache fault stream will have a stationary behavior too.

*local page cache.* In that respect, the partitioned server cache does not add any interesting properties to the model.

### 3.1.2 Clustering Objectives

As shown in the previous analysis, there are two ways to affect the performance of the multi-client OODBMS of Figure 3.1. The choice of *Object Placement* $D_\pi$ affects the *"locality"* of each client page access stream, and thus the effectiveness of the client cache and the server cache. The choice of *Page Placement* $D_\sigma$ affects the *"arm movement"* of the disk on the server, which determines the average page access cost on the server.

Globally optimal static clustering decisions in a multi-client environment are unrealistic. In such environment object access costs vary greatly, depending not only on the controlled static parameters (like the placement of objects and pages), but also on uncontrolled dynamic parameters (like the number of clients and their request rates). For example, the cost when accessing a page from the server, depends on the server cache hit ratio, the load of the server and the disk seek time. Both are dynamically changing parameters and static clustering decisions should not depend on it.

The standard way to attack such problems is to optimize the performance of each system component separately, thus excluding dynamic parameters from the optimizations. Object and Page placement can be done *independently*, using appropriate static information for each one. As a result, optimal clustering for the model presented should be defined as two separate sub-problems:

1. *Choose an* ***object placement mapping*** $D_\pi$ *that maximizes the performance of the cache at each client.*

2. *Choose a* ***page placement mapping*** $D_\sigma$ *that minimizes the disk seek time at the server.*

The second problem requires a statistical description of the global stream $V'(t)$. $V'(t)$ depends on the client and server cache and the queueing at the server, and it is impossible to calculate them statically. This problem is a good candidate for dynamic or adaptive solution, where the global stream is inspected for some period, and the probabilities are estimated from the stream itself. In contrast, the first problem can be solved by just using static access models for the clients.

In this thesis, we will assume that no object is larger than a page. Each large object requires a number of pages and object placement is no issue. Page placement however, may become important in the case all pages of a large object are required when that object is

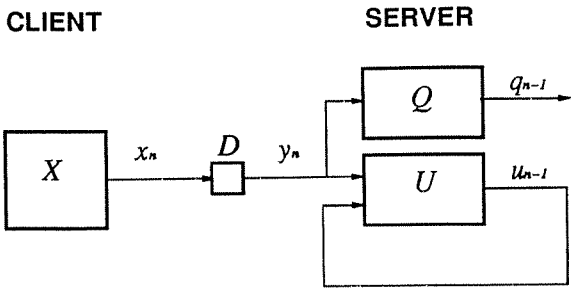



Figure 3.2: The Client-Server cost

requested by a client. As a result, no special attention is needed for large objects in terms of object placement, but page placement must treat them as a group.

Both placement problems, involve the minimization of the average access cost through a static set-to-point ($N$ to 1) mapping, given a description of the access patterns, and motivate the *"abstract clustering problem"*, which is formulated and solved next.

## 3.2 Problem Formulation

The two placement problems encountered in Section 3.1.2 and other frequently arising performance problems in software and hardware systems, can be phrased as follows:

> Two entities communicate by exchanging synchronous messages; the first (client) requests "objects" one at the time, and the second (server) provides those objects. Objects are accessed from pages containing groups of objects, and there is some cost associated with fetching pages. How should objects be assigned to pages, so that the communication is efficient?

The client can be a program whose data structures (objects) are assigned to disk pages accessed by the file-system server. In the case of a file-system, file pages are mapped to disk blocks provided by the disk (server). Program instructions are placed in pages by the loader, in a way that minimizes page faults a processor will generate while executing the program.

The above problems motivated the *Abstract Clustering* problem we will study in this chapter. The problem involves:

- The *object space* $S$ containing a set of $N$ objects, denoted by the positive integers: $S = \{1, 2, 3, \ldots, N\}$. Each object $x$ has a positive size given by the function $size(x)$.
- The *storage space* $F$ containing a set of $M$ "pages", denoted by the positive integers: $F = \{1, 2, 3, \ldots, M\}$. Each page has a fixed capacity $L$.

- The *client* entity generating a sequence of object requests called the "request stream" or "trace", denoted by $(X_n) = x_1, x_2, \ldots, x_n$.
- The *clustering mapping*, a set-to-point $(N \rightarrow 1)$ mapping $S \xrightarrow{D} F$ defining the assignment of objects to pages.
- The *server* entity that provides access to objects a client needs, accessing a page $y_n = D(x_n)$ at time $n$ with cost $q_n$.

The abstract clustering problem involves finding a clustering mapping $D$ that minimizes the average access cost of client requests. More precisely, it can be formulated as an optimization problem. *Given:*

- *a statistical description of the client access stream $X_n$, and*
- *a page access cost formula $Q$ for the client-server interactions,*

*find the mapping $D : S \rightarrow F$ such that:*

- *the average cost $T$ is minimized, while*
- *the size of all objects in a page* [2] *is at most $L$ and*
- *other additional constraints are satisfied.*

The additional constraints can be arbitrary restrictions on where and how objects can be assigned. One such constraint would be to restrict the placement of objects to pages by their types or their sizes. Clustering in the presence of additional constraints can be very hard. For example, the page-space minimization problem under variable size objects and no regards to the request stream is NP–complete (the knapsack problem). In this chapter we will ignore those additional constraints so we can concentrate on the rest of the problem.

We further assume the following access cost model. The cost of accessing a page $y_n$ depends on the current page and on the current state of the server $u_n$, and the state changes depending on $y_n$ and on the previous state. The next equations specify them more formally:

$$\begin{aligned} q_n &= Q(y_n, u_n) \\ u_{n+1} &= U(y_n, u_n) \end{aligned}$$

where $Q$ and $U$ are the cost and state update functions. This cost model (also shown in Figure 3.2) is simple and yet as we will see, it can capture a variety of data-server situations.

[2] If $L = \infty$ the problem has a trivial solution: assign all objects to one page.

In addition, the motivation to introduce the server state in the cost is the following: very often, the data retrieval costs are not fixed; they may depend on the current physical "state" of the storage device, or on the current contents of the server buffer pool. The model provides a simple way to decouple the effect of the request stream on the access cost and on the server state.

The total cost $T_n$ and the average cost $T$ are defined as:

$$T_n \equiv \begin{cases} T_{n-1} + q_n & n > 0, \\ 0 & n = 0 \end{cases}$$

$$T = \lim_{n \to \infty} \frac{\bar{T}_n}{n}$$

A standard way to model $y_n$ and $u_n$ is as random variables, and the sequences $(X_n)$, $Y_n = D(X_n)$, and $(U_n)$, as stochastic processes. In that framework the average page access cost $T$ *converges* to the expectation[3] of the random variable $q_n$:

$$T = \mathcal{E}(q_n) = \mathcal{E}(Q(y_n, u_n)) \tag{3.1}$$

Using average costs is a reasonable way to evaluate the effect of a clustering mapping to the performance of the client-server system. In most common scenarios, the average access cost directly influences the execution time of the clients and the server load. In some cases cost metrics like the maximum cost could be used instead, with the assumption that costs below a threshold can be tolerated.

For mathematical convenience (and to be consistent with the definitions of Section 3.1.2) we sometimes view $D$ as a two step mapping:

- A partition of $S$; $D_\pi : S \to F$ (object placement).
- A permutation of pages; $D_\sigma : F \to F$ (page placement).

Consequently every such mapping $D$ can be expressed as: $D(\cdot) = D_\sigma(D_\pi(\cdot))$.

## 3.3 Special Cases

To make any progress, we need to know some statistical properties of the client access request stream $X_n$ (Chapter 2 discusses more modelling issues). In this section we will assume that $X_n$ is a sequence of independent and identically distributed random variables (IID model).

[3] $T$ is the mean of a sum of infinite random variables, and by virtue of the law of large numbers and assuming ergodicity of the $y_n, u_n$, it converges to the expected value of the random variable $q_n$.

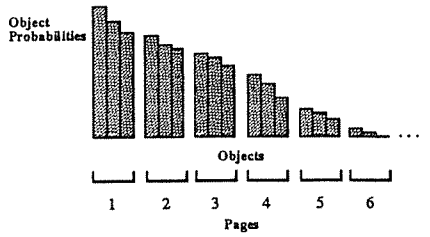



Figure 3.3: The Probability Ranking Partitioning to solve the delta-uniform cost problem

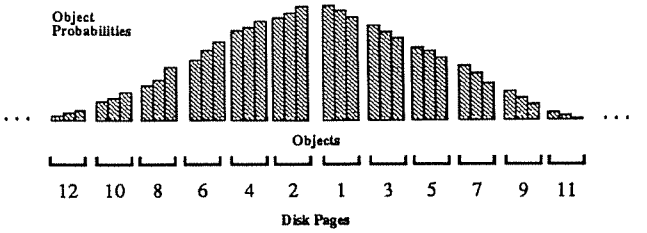



Figure 3.4: The Pipe Organ Assignment to solve the disk cost problem

Their common value domain is the set of objects ($S$) and their distribution is defined by a probability vector $\vec{\pi}$:

$$\pi(y) \equiv Pr(x_n = y),\ \forall y \in S,\ n = 0, 1, \ldots$$
$$\text{and}\quad \textstyle\sum_{y \in S} \pi(y) = 1$$

We will examine optimal clustering for the IID case for three different server models: the uniform, delta-uniform, and disk cost server.

If the access cost $Q$ is *uniform*, i.e., it does not depend on the server state or on the requested page ($Q(y_n, u_n) = 1$), then the average cost given in Equation 3.1 becomes a constant that does not depend on the clustering mapping $D(\cdot)$. In this case, the average cost of any page is always the same and optimal clustering is trivial to find.

The *delta-uniform cost* models an 1-page cache server. That is, the server keeps the last page requested ($u_n = y_n$) and there is no cost if the same page is requested again.

$$Q_{n+1} = \delta_{y_{n+1}, y_n} \tag{3.2}$$

where $\delta$ is Kronecker's function ($\delta_{x,y} = 1$ if $x = y$, 0 otherwise).

The *optimal* clustering scheme in the above case due to Yue and Wong [YW73], is known as the probability ranking scheme. Its partition component $D_\pi$ involves sorting the objects in descending order of their absolute probability $\vec{\pi}$ and successively assigning them to pages in that order as in Figure 3.3a, where objects with similar temperature[4] are assigned to the same page as much as possible. The permutation component is the identity ($D_\sigma(p) = p$), since the cost formula does not depend on page numbers but only on page equality.

The *disk cost* models the case of a single-arm disk server. The cost for fetching a block from such a disk is dominated by the seek time, which depends on the current position of the disk head. The server state in this case is simply the last page requested ($u_n = y_{n-1}$), and

[4]The term *temperature* of an object is sometimes used instead of the term *absolute access probability* of an object for the IID case, or *stationary probability* of an object for the SMC case.

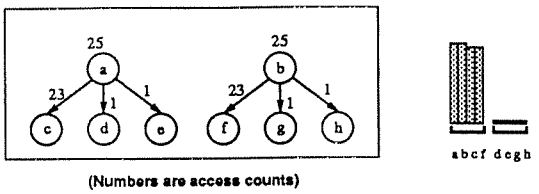



Figure 3.5: Clustering objects using probability ranking (page size=4)

the position of a page is a function of that page only $c(y_n)$. The new cost formula models the disk geometry characteristics closely, and more specifically the cost is some function of the absolute *distance* from the last request:

$$Q(y_n, u_n) = g(|y_n - y_{n-1}|) \tag{3.3}$$

Additionally, the function $g(a)$ is increasing with $a$, since the longer the head travels the greater the cost should be:

$$g(0) \leq g(a) \leq g(b), \ \forall\ 0 \leq a \leq b$$

The solution of the disk-cost problem was found by Wong and Yu [YW73], Grossman and Silverman [GS73]. Wong's book [Won83] generalizes those algorithms for the case of different disk devices and request protocols.

The partition component of the optimal mapping remains the same as in probability ranking. The permutation is not the identity any more, but involves sorting the pages in descending order according to the sum of the probabilities of the objects they contain, and placing them on disk as in Figure 3.4:

$$2n \ ...6, 4, 2, 1, 3, 5... \ 2n + 1$$

This clustering scheme is known as the *organ pipe arrangement*. In this arrangement, the hottest pages are placed in the middle of the disk. The two next to the hottest page are placed adjacent to it and the process is repeated until all the pages have been assigned.

An adaptive system, proposed and implemented by Vongsathorn and Carson in [VC90], uses the above method to dynamically permute disk blocks so that the average observed access time is minimized. During an observation period, disk block accesses in a Unix filesystem are measured, and an IID model is estimated from counting the accesses. Finally, a new permutation is computed and implemented if necessary. Substantial performance improvements have been observed in that system, and the average seek time was reduced by 40 - 50%.

In the context of OODBMS, IID-probability ranking clustering can be better than purely syntactic clustering. IID clustering considers usage statistics, whereas syntactic methods

use structural data only. But this data alone cannot reveal how the object structure will be used. In the example presented in Figure 3.5, syntactic methods will fail to place the hottest objects $\{a, b, c, f\}$ together because of their structural independence. However, the probability ranking method will do so.

On the other hand, probability ranking will fail in the example of Figure 1.1. It has no way to distinguish between $\{s, i, m, p, l, e\}$, since, under the IID model, all objects have the same access probability. An SMC model can capture correlations like $s \leftrightarrow i$, and as we will see next, it will produce the optimal partition $C_2$. The IID model captures no correlation information, since an IID stream may contain any object reference $y$ at any time with a fixed probability $\pi(y)$ independent of previous requests.

If the client request stream is not IID, the above solutions are only approximate except for the uniform cost server. In the next section we will generalize those problems considering SMC and higher order access models and multi-page caches.

## 3.4 The Multi-Page Cache Server

The analysis presented in Section 3.1, suggests that the object placement must be designed to maximize the performance of the client cache. This corresponds to a special case of the abstract clustering problem, where the server state is the current contents of a cache of size $K$ $(z_n)$, and some statistics $(r_n)$. The server cost is:

$$\begin{aligned} q_n \equiv Q(y_n, u_n) &= 1(y_n \notin z_n) \\ u_n &= (z_n, r_n) \end{aligned}$$

where $1()$ is the indicator function: $1(e) = 1$ if $e$ is true, 0 otherwise. The average cost is simply:

$$T = Prob\{y_n \notin z_n\}$$

The server state changes according to the current request, and the *page replacement policy* employed by the cache.

The expected server cost is equal to the expected number of cache misses, and minimizing them depends on the cache management policy. The cache policy introduces another parameter to an already difficult problem. After all, there is no easy way to describe all possible cache policies or to decide in advance which one would be the best. For this reason, instead of trying to minimize misses for caches of a specific type, it seems more reasonable that to cluster in a way that improves the *locality* of the client request stream, a metric independent of the cache management policy. Then, most cache policies can benefit from the enhanced locality and the expected number of misses will decrease.

### 3.4.1 A Measure of Locality

A standard metric for locality is the working set size [Den68], the expected cardinality of the set $R_t^{(M)}$ of $M$ consecutive page requests starting at time $t$ (also used in [YW73]). That is: take these $w$ page requests, eliminate duplicates, and compute the cardinality of the resulting set. Note that the larger the cardinality, the fewer the duplicates, hence the lower the locality. Therefore, in our case the working set sized (denoted by $WSS(w)$ or for short $K_t^{(w)}$) is:

$$K_t^{(w)} = \mathcal{E}(||R_t^{(w)}||)$$

Obviously $1 \leq K_t^{(w)} \leq w$. If $w > M$ then the upper limit is $M$ ($M = ||D_\pi(S)|| =$ the number of pages the whole object space $S$ maps to). The window parameter $w$ allows to optimize for different cache sizes, because the smaller $K_t^{(w)}$ is, the more effective a cache of size $w$ is. If the cache size is $\geq M$, any replacement policy will work, since the whole object base fits into the cache.

For *ergodic sequences* the distribution of the $K^{(w)}$ is independent of time $t$ (like IID and time invariant Markovian models). From this point on we will drop the $t$ subscript when we refer to those models. Consequently, the $WSS$ formula is:

$$K^{(w)} = \mathcal{E}(||R^{(w)}||) \tag{3.4}$$

The rest of this chapter is devoted to this problem. As we will see, partitioning the object space to decrease $WSS$ is not an easy task.

### 3.4.2 Maximizing Locality under IID

It is not hard to come up with a closed form expression of $K^{(w)}$ for the IID case. Appendix A.2.1 has all the steps of the derivation procedure:

$$\begin{aligned} K^{(w)} &= w - \sum_{q=1}^{M}(1 - \pi_F(q))^w \\ &= w - \sum_{q=1}^{M}(1 - \sum_{\substack{y: \\ D_\pi(y)=q}} \pi(y))^w \end{aligned}$$

$M$ is the total number of pages, and $\pi_F(q)$ is the total access probability of page # $q$.

It can be shown that $K$ is minimized for every value of $w$ *when each page $f$ contains objects placed using the probability ranking* (see Appendix A.2.1). This agrees with results in [YW73], where a different methodology was used.

### 3.4.3 Maximizing Locality under SMC

The *SMC (Simple Markov Chain) model* is a more powerful model, to describe access correlations (see Chapter 2 for the full definitions). In Appendix A.1 we show that a partitioned Markov state space appears at the limit like a Markov process. As a result, accesses of the memory space $F$ due to object accesses in $S$ can also be modeled as SMC, with stationary probability vector denoted $\vec{\pi_F}$ and transition probability matrix denoted $P_F$.

The derivation of $K^{(w)}$ for the SMC model is given in Appendix A.2.1. Using an occurrence random variable $W(w,q)$ $(= 1,$ if $q \in \{x_1, ..x_w\}, 0$ otherwise), the expected $WSS$ is:

$$K^{(w)} = \sum_{q=1}^{M} W(w,q)$$

For $w = 1$, $K$ is always 1. For $w = 2$:

$$K^{(2)} = 2 - \sum_{q=1}^{M} \pi_F(q) P_F(q,q)$$

For $w \geq 3$ the formula becomes π ore complicated, involving higher powers of the $P_F$ matrix. Using the results of Appendix A.2.2 minimizing $K^{(2)}$ corresponds to maximizing $L_2$:

$$\begin{aligned} L_2 &\equiv \sum_{q=1}^{M} \pi_F(q) P_F(q,q) \\ &= \sum_{q=1}^{M} \sum_{\substack{x: \\ D(x)=q}} \sum_{\substack{y: \\ D(y)=q}} \pi(x) P(x,y) \end{aligned}$$

or minimizing $G_2$:

$$\begin{aligned} G_2 &\equiv \sum_{q=1}^{M} \pi_F(q)(1 - P_F(q,q)) \\ &= \sum_{q=1}^{M} \sum_{\substack{x: \\ D(x)=q}} \sum_{\substack{y: \\ D(y)\neq q}} \pi(x) P(x,y) \end{aligned}$$

Minimizing $WSS$ for $M = 2$ is a weighted graph partitioning problem where the weight $q$ of an edge $e = a \rightarrow b$ is:

$$q(e) = \pi(a)P(a,b) + \pi(b)P(b,a)$$

Each partition must have up to a maximum number of nodes and $L_2$ must be maximized. Alternatively, the problem is equivalent to minimizing $G_2$, the sum of weights of the all

edges crossing the partition boundaries. It is interesting to observe that clustering scheme $C_2 = \{[si], [mp], [le]\}$ minimizes the $WSS$ $K^{(2)}$ for the example of Figure 1.1.

The weighted graph partitioning problem has been studied previously. According to [GJ79], deciding if there is a partition with cost $\leq J$, is an NP-complete problem. The clustering decision corresponds to partitioning the object graph vertices to $V_1, V_2 ... V_m$, with:

- vertex weight $q(v)$ = size of the object $v$,
- edge weight $q(e) = q(a, b)$ of an edge $e : a \rightarrow b$,
- $\sum_{v \in V_i} q(v) \leq L$, and
- $\sum_{e \in E} q(v) \leq J$.

where $L$ is a limit in the capacity of pages, and $E$ is the set of all edges that cross the partition boundaries. $J$ is the upper limit for the partition $WSS$ sought.

To prove that optimal object clustering in the case of SMC access patterns is NP-complete, we reduce the problem of graph partitioning to optimal clustering. Considering the formulation of the two problems, it suffices to show that for any instance of GP there is a Markov Chain SMC, such that the optimal clustering OC(SMC) has the same solution as GP.

We represent the edge costs of an $N$ node GP by an $N$ by $N$ matrix $A$:

$$A = \begin{bmatrix} 0 & a_{12} & \dots & a_{1N} \\ a_{12} & 0 & \dots & a_{2N} \\ & & \dots & \\ a_{1N} & a_{2N} & \dots & 0 \end{bmatrix}$$

The main diagonal values are set to zero since they cannot be cut by any partition and thus, do not affect the partition cost. The matrix is symmetric since the cost of cutting an edge does not depend on the direction of the edge (if any).

From $A$ we construct a scaled matrix $B$:

$$\begin{aligned} B &= \lambda A \text{ where} \\ \lambda &= max \left\{ \sum_{k=1}^{N} [A]_{ki}, \sum_{k=1}^{N} [A]_{ik} \right\}, \; i = 1, 2, \dots, N \end{aligned}$$

From this construction, it is obvious that $B$ is also symmetric, has zero in its diagonal elements, each element is at most 1, and the sum of each row or each column elements is no bigger than 1. Furthermore, partitioning of the graph defined by $B$ is the same problem as the original GP, since the relative cost of partitions does not change with scaling.

From matrix $B = [b_{ij}]$ we construct another matrix $C = [c_{ij}]$ as follows:

$$c_{ij} = \begin{cases} b_{ij} & \text{if } i \neq j \\ \frac{1}{N} - \sum_{k=1,k\neq j}^{N} b_{kj} & \text{otherwise} \end{cases}$$

We claim that $C$ *is a cost matrix of an object clustering problem under an SMC access model with paramaters* $P$ *and* $\vec{\pi}$ given by the equations:

$$\begin{aligned} \pi(y) &\equiv \sum_{x=1}^{N} c_{xy}, \text{ and} \\ P(x,y) &\equiv \frac{c_{xy}}{\pi(x)} = \frac{c_{yx}}{\pi(y)} \end{aligned}$$

Using the definition of $C$, it is easy to verify that the constructed $P$ and $\vec{\pi}$ represent indeed a Markov Chain. For any $x, y = 1, 2, \ldots, N$:

- $\pi(x)$ is probability:

$$\begin{aligned} \pi(x) &\geq 0 \\ \sum_{x=1}^{N} \pi(x) &= \sum_{x=1}^{N}\sum_{y=1}^{N} c_{yx} = \sum_{y=1}^{N} \frac{1}{N} = 1 \end{aligned}$$

- $P(x,y)$ is transition probability:

$$\begin{aligned} P(x,y) &\geq 0 \\ P(x,y) &= \frac{c_{xy}}{\sum_{x=1}^{N} c_{xy}} \leq 1 \\ \sum_{y=1}^{N} P(x,y) &= \sum_{y=1}^{N} P(y,x) = \sum_{y=1}^{N} \frac{c_{yx}}{\pi(x)} = \frac{\sum_{y=1}^{N} c_{yx}}{\pi(x)} = 1 \end{aligned}$$

- $\pi(x)$ is stationary probability:

$$\sum_{x=1}^{N} \pi(x)P(x,y) = \sum_{x=1}^{N} \pi(x)\frac{c_{xy}}{\pi(x)} = \sum_{x=1}^{N} c_{xy} = \pi(y)$$

Although the problem is NP-complete, there are some good heuristic algorithms to find close-to-optimal solutions fast. Notably, Kernighan and Lin partitioning [KL70] is a good $O(n^{2.4})$ heuristic algorithm. [Bar84] and [BVJ84] propose asymptotically faster but more complicated algorithms.

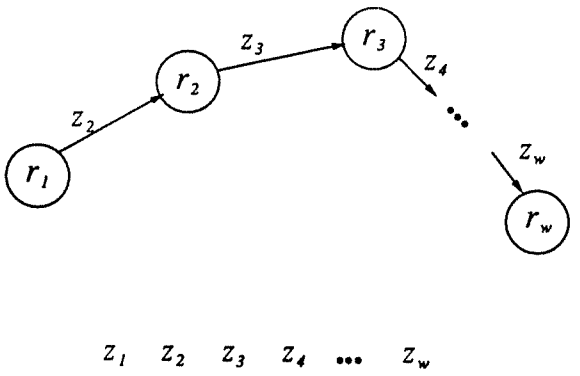



Figure 3.6: An HMC chain of states

### 3.4.4 Maximizing Locality under HMC

In this section, we will generalize the problem of optimal object placement, by considering higher order Markov chains as models of the object request streams. We assume that the object request stream is modeled by a $k$-th order Markov process (denoted by $HMC_k$), that has a stationary distribution, consists of a single set of recurring states, and furthermore is aperiodic and time invariant. HMC was fully defined in Chapter 2.

The state of the process at time $t$ is a $k$ element vector of the $k$ most recently accessed objects:

$$\vec{x}_t = [x_{t-k+1}, x_{t-k+2}, \ldots, x_t] \in \Omega$$

and the state space is $\Omega = S^k$. At any time $t$ the probability an object $y$ appears in the trace only depends on $y$ and on the $k$ most recently requested objects. Formally:

$$P(x_1, \ldots, x_k; y) \equiv Prob\{X_t = y | X_{t-k} = x_1, X_{t-k+1} = x_2, \ldots X_{t-1} = x_k\}$$

or using a vector notation:

$$P(\vec{x}, \vec{y}) \equiv Prob\{X_t = y | X_{t-k} = x_1, X_{t-k+1} = x_2, \ldots X_{t-1} = x_k\}$$

where $\vec{x}$ is the current state:

$$\vec{x} = [x_1, \ldots, x_{k-1}, x_k]^T \in \Omega,$$

and $\vec{y}$ is the next state:

$$\vec{y} = [x_2, \ldots, x_k, y]^T \in \Omega,$$

The stationary probability of the process being at state $\vec{u} \in \Omega$ is is given by a $k$-ary function, the *stationary distribution of the chain*:

$$\pi(\vec{u}) = \pi([\vec{u}]_1, [\vec{u}]_2, \ldots, [\vec{u}]_k)$$

Now let us consider a window of $w$ consecutive requests ($w > k$) starting at time $t$ and represented by a $w$ element vector $\vec{z}$:

$$\vec{z} = [z_1, z_2, \ldots, z_w]^T \in S^w$$

Because of the stationary properties of the chain, the distribution of states $z_1, \ldots z_w$ does not depend on the starting time $t$. In addition, the window represents a chain (see Figure 3.6) of the following states:

$$\vec{z} \rightarrow \vec{r}_1, \vec{r}_2, \ldots, \vec{r}_{w-k+1}$$

where

$$\begin{array}{lcccccccc}
\vec{r}_1 & = & [ & z_1 & z_2 & z_3 & \cdots & z_k & ]^T \\
\vec{r}_2 & = & [ & z_2 & z_3 & z_4 & \cdots & z_{k+1} & ]^T \\
\cdots & & & & & & \cdots & & \\
\vec{r}_{w-k+1} & = & [ & z_{w-k+1} & z_{w-k+2} & z_{w-k+3} & \cdots & z_w & ]^T
\end{array}$$

The probability to encounter chain $\vec{z}$ is given by the formula:

$$\pi(\vec{z}) \equiv Prob\{\vec{z}\} = \pi(\vec{r}_1) \prod_{i=1}^{w-k} P(\vec{r}_i, \vec{r}_{i+1}) \tag{3.5}$$

Now suppose there is an object to page mapping $D(\cdot)$, then each possible window of object requests $\vec{z}$ from the HMC process is mapped in to a set of pages with at most $w$ elements. Let $D(\vec{z})$ denote this set. Then the expected working set size for a window of $w$ requests is given by the formula:

$$\mathcal{E}(WSS_L(w)) = \sum_{\vec{z} \in S^w} \pi(\vec{z}) ||D(\vec{z})|| \tag{3.6}$$

Consequently, *if the object stream is a result of a k-th order Markov process, the expected working set size is given by a closed form expression that depends on the stationary and transition probabilities of the process* (Equation 3.6). The problem of finding the optimal object placement corresponds to choosing a partitioning function $D$ that minimizes the expected working set size $\mathcal{E}(WSS_D(w))$ given by the above cost equation.

To show the complexity of optimal object placement, we will use a hypergraph HG to represent the access pattern induced on a window of $w$ requests. Let $HG(V, E)$ be a weighted hypergraph with a set of vertices $V$ and a set of undirected hyper-edges $E$. Each vertex represents an object from $S$ and therefore $V = S$ and has a weight equal to the size of the object. Each hyper-edge links up to $w$ vertices representing all possible contents of a window of $w$ objects. It has weight equal to the probability a window has those contents.

More formally:

$$\begin{aligned} HG(V,E) &: \text{a weighted hypergraph} \\ V &= S \\ E &= \{e : e \subset S, ||e|| \leq w\} \\ q(v) &= size(v),\ \forall v \in V \\ q(e) &= \sum_{\vec{z}\ \in\ vectors(e)} \pi(\vec{z}) \\ vectors(e) &= \{\vec{u} : \vec{u} \in S^w \text{and } e = \cup_{i=1}^{w} \{[\vec{u}]_i\}\} \\ \pi(\vec{z}) &\quad \text{is given by Equation 3.5} \end{aligned}$$

An object placement $D$ corresponds to a partition of the HG vertices:

$$\begin{aligned} D &: V_1, V_2, \ldots, V_M,\ V_i \in V \\ V_i &= \{u : u \in V, D(u) = i\} \end{aligned}$$

such that:

$$\sum_{v \in V_i} q(v) \leq L$$

and

$$J = \sum_{e \in E_v} q(e)$$

where $L$ is a limit in the capacity of partitions, $E_v$ is the set of all edges that cross the partition boundaries, and $J$ is the partition cost, i.e. the average working set size of a window of size $w$.

From the above it is clear that the problem of optimal object placement, corresponds to finding a partition of HG that has the minimum cost. As reported in [GJ79], such graph and hypergraph partition problems are NP-complete. Even checking for the existence of a partition of a cost less than some constant is an NP-complete problem. Therefore, the problem of optimal object placement under the HMC model is NP-complete. The same problem under the PSM model is also NP-complete, since a PSM model can be trivially mapped into hypergraph partitioning.

## 3.5 Stochastic Clustering Algorithms

In this section we present two new algorithms for object placement based on the theory developed in Section 3.2. Both algorithms produce a placement of objects to pages that increases locality, using a graph representation of the stochastic access model as input. The

first algorithm (SMC.WISC) is greedy but has low cost, whereas the second (SMC.KL) is more accurate but more expensive.

The access patterns model employed is a first order Markov Chain (defined in Chapter 2) with stationary probability vector $\vec{\pi}$ and transition probability matrix $P$. More specifically, the input is an *undirected weighted graph* $SMCG(V, E)$ defined as follows:

- one vertex for each object accessed, $V = S$.
- one edge for each possible access correlation:

$$E = \{e = (v_1, v_2) : \forall e \in V^2, P(v_1, v_2) \neq 0\}$$

- vertex weight equal to the size of its object:

$$q(v) = size(v), \ \forall v \in V$$

- edge weight equal to the average probability of observing the corresponding sequence of objects:

$$q(e) = \pi(v_1)P(v_1, v_2) + \pi(v_2)P(v_2, v_1)$$

  where $e = (v_1, v_2)$.

In the subsequent presentation, $L$ denotes the capacity of a page.

### 3.5.1 A Greedy Algorithm: SMC.WISC

The algorithm assigns objects to pages so that the following conditions are met:

1. The total size of the objects assigned to any page is less than or equal to the page size.
2. Object placement is "locally optimal": as unclustered objects are being assigned to pages, they are selected so that the probability of re-using their page is maximized.

SMC.WISC assigns the objects sequentially and never reconsiders an object once it has been assigned to a page. For that reason the algorithm is called *greedy*. During the clustering process hot objects are considered first, by sorting them according to their stationary probability. SMC.WISC starts by placing the hottest object in the first page, and pages are filled incrementally as follows

Let $R_n$ represent the contents of a page on step $n$, and $C(R_n)$ be the 1-closure of the objects in $R_n$ in terms of the access graph $SMCG$:

$$C(R_n) = \{v : v \in V, \exists e \in E, x \in R_n : e = (v, x) \vee e = (x, v)\}$$

i.e., all objects reachable from or pointing to $R_n$. From all members of $C(R_n)$ the algorithm selects an object $y$ such that:

- $y$ has not been assigned to any group yet.
- $y$ fits in the current page, i.e.

$$size(R_n) + size(y) \leq L$$

- the probability to reuse the page after $y$ is added is maximized among all possible choices of $y$:

$$H(R'_n) \equiv Prob\{X_t \in R'_n | X_{t+1} \in R'_n\}$$

$R'_n$ denotes the contents of the page after $y$ has been added: $R'_n = R_n \cup \{y\}$.

Using the results of Appendix A.1 regarding the stationary behavior of partitioned state space (and more specifically Equation A.5) we can calculate $H(R'_n)$:

$$H(R'_n) = \sum_{z \in R'_n} \sum_{x \in R'_n} \frac{\pi(x)}{\sum_{u \in R'_n} \pi(u)} P(x,z) = \frac{\sum_{z \in R'_n} \sum_{x \in R'_n} \pi(x) P(x,z)}{\sum_{u \in R'_n} \pi(u)} \tag{3.7}$$

The numerator is simply the sum of edge weights of all edges present in the candidate group $R'_n$, and the denominator is the "probability mass" or stationary probability of $R'_n$.

When $n$ objects have been assigned to a page and there is still room, SMC.WISC calculates $H(R_n \cup \{y\})$ for each candidate $y$, and selects the $y$ that maximizes $H(R_n \cup \{y\})$. The calculation of $H(R'_n)$ is done incrementally, since adding $y$ to $R_n$ will add some edge weights to the numerator, and a probability mass to the denominator. A hash index on $SMCG$ quickly finds edges containing $y$, and thus, calculating $H(R'_n)$ requires at most checking $|d(y)|$ edges (where $d(y)$ is the degree of $y$). Therefore, the complexity of the SMC.WISC algorithm will be $O(N \log N + ||E||) = O(N(\log N + d))$, where $d$ is the average degree of the $SMCG$ nodes and $||E||$ is the number of $SMCG$ edges. For sparse graphs the average degree is usually less than $\log N$ and therefore the complexity of the SMC.WISC becomes $O(N \log N)$.

### 3.5.2 A More Accurate Algorithm: SMC.KL

SMC.KL is based on a well known graph partitioning algorithm (KL) initially proposed by Kernighan and Lin in [KL70]. SMC.KL employs KL to partition $SMCG$ to groups of $L/s$ objects each (where $s$ is the average object size). The KL algorithm partitions a graph into disjoint subsets, so that the sum of edge weights that cross the partition boundaries is minimized. The partition found is guaranteed to be *pairwise optimal*, that is, no better partition can be obtained by exchanging any two vertices from any two subsets. As we have

seen in Section 3.4.3, the minimum cost partition will minimize the *expected working set size* of a window of 2 requests.

KL operates by attempting to decrease the cost of the current partition until no further improvement is possible, and the quality of the partition found depends much on the *starting partition* [PS82]. Although a random initial partition can be used, we found that starting KL from the mapping of SMC.WISC is a very effective heuristic; KL produces a much better clustering than if the initial partition were random, and always improves the SMC.WISC clustering. The monotonic cost improvement property of KL makes it more accurate than SMC.WISC with respect to the optimal partition. This is why we use SMC.WISC as a front-end of SMC.KL, thus offering a variable cost variable quality clustering mapping.

Depending on the size of each object, a group generated by the KL algorithm may not necessarily fit in one page, or it may take less than one page.[5] To circumvent this problem, SMC.KL re-maps the groups generated by KL to fixed size pages as follows. First the stationary probabilities of each group is calculated (just the sum of the probabilities of all objects assigned to that group), and the groups are sorted with respect to that probability in decreasing order. Within each group, objects are sorted in decreasing order with respect to their size multiplied by their stationary probability. Then objects of each page are assigned to a page, and if the page overflows a new one is created. When all objects of group are assigned, the procedure continues with the next group, until all objects have been reassigned. Except the above changes, the original KL algorithm is employed intact.

## 3.6 Related Work

In this section we will discuss related work on problems related to that of object clustering: *Record Clustering, Graph clustering, and Program Restructuring.* Discussion of work directly related to clustering in object oriented databases is postponed until Chapter 4, where several such techniques are presented and evaluated.

In all cases the fundamental problem is very similar; minimizing the I/O cost of accessing a set of "objects" allocated on secondary storage. However the context, the terminology, the methodology, and the assumptions used are very different. The abstract clustering problem generalizes and unifies all these problems.

### 3.6.1 Record Clustering

Suppose we have a set of queries $Q = \{Q_1, Q_2, \ldots, Q_n\}$ each one issued with fixed probability $P(Q_i)$, each time accessing the same set of records. Suppose also that each record is assigned

[5]The total size of all groups will be exactly equal to the size of all objects.

to a block in secondary memory, and that $B(Q_i)$ denotes the number of blocks needed to retrieve $Q_i$. The problem of record clustering is deciding how to assign records to blocks so that the average number of blocks needed per query is minimized. More formally:

$$\sum_{Q_i \in Q} P(Q_i)B(Q_i)$$

should be minimized.

The problem has been shown to be NP-hard by Yu et al in [YLT83]. Hammer and Chan in [HN76], and Rivest in [Riv76], proposed two simple algorithms to solve the problem in case the query probabilities are known in advance. Yu, Sue, Suen and Siu in [YSLS85] proposed an algorithm for record clustering, which dynamically re-clusters records as queries are being submitted. This problem, although it appears similar to abstract clustering does not consider (neither benefit from) the effect of inter/intra-query caching. If queries are mutually exclusive the problem is trivial, but if there is any query sharing such caching can improve the performance and it should be taken into account.

### 3.6.2 Graph clustering

Suppose we have a directed graph $G(V, E)$. How should the vertices and edges of this graph be represented and stored on the disk, such that the access cost is minimized? This question is raised by many applications that use large graphs as the underlying data structure. For reasons of size and persistence the graph must be kept in the secondary memory, and "efficient" methods for storing and retrieving it, must be introduced.

The existing practice in dealing with graph clustering has been can be classified in two categories: a) to take advantage of specific ways graphs are traversed, and b) to make "worse case assumptions" about the accesses by giving equal importance to all edges of the graph. Clustering of CAD data structures belongs in the first category. CAD graphs are usually traversed in some "traditional" fashion, like Depth First Search (DFS) or Breadth First Search (BFS). In both cases, each graph node is only visited once and clustering is trivial; just assign objects to pages as they are encountered in the traversal. Typical such algorithms were proposed by Chang and Katz [CK89] and by Banerjee et al in [BKKG88].

An interesting graph clustering approach was proposed by Mendioroz in his Master's thesis [Men91], and belongs to the second category. The suggested method assigns graph nodes to disk pages dynamically as the graph is being "built" or "modified". To accommodate fast insertions and deletions without creating "sparse" pages, page occupancy is kept between an upper and a lower limit. During an insertion (deletion) page split (merge) may be initiated, should the page occupancies not fall within the limits. The principle criterion for node distribution (split) or node grouping (merge) is *local optimal partitioning*; among

the pages considered for splitting or merging, nodes are arranged in a away that is minimizes the number of edges crossing page boundaries. Network flow techniques are used to find the "optimal" distribution of nodes.

This method makes some worse case assumptions about the usage of edges; all edges are assumed to be equally important, an assumption that may not necessarily be true in practice. For example, if the graph is dense but not all edges are hot, considering only the hot ones can give better performance. As we will see in Chapter 4 a straightforward application of this technique to object clustering is not much better than random clustering.

### 3.6.3 Program Segmentation and Restructuring

Perhaps the most similar problem to abstract clustering is *program restructuring* and *program segmentation.* The objective here is to maximize the performance of a program running under swap based or virtual memory, by properly assigning "program blocks" to segments or pages. The client in this case is the processor running some program on behalf of the user, fetching code instructions from the main memory. The server is the memory abstraction provided by the operating system that uses the real memory. Program blocks must be assigned to segments or virtual memory pages, and the goodness of an assignment is measured typically by the number of segments/page faults a program will produce during its execution.

*Program Segmentation* techniques were very common on systems with no virtual memory capabilities to run programs that would otherwise not be able to fit in main memory. Segmentation in that context refers to partitioning the program code to portions that are mutually exclusive as much as possible (i.e. one does not call the other), and assigning them to different segments [Sto87]. During program execution, a number of segments may be present in main memory, and the operating system (perhaps under user control) swaps whole segments from main memory and brings in the requested segments. Similar techniques were used for swapping large data structures (e.g. FORTRAN arrays). Program (and data) segmentation techniques were usually manual, left up to the programmer to specify and apply them.

The concept of *virtual memory* freed the programmer from manually segmenting and swapping the program code, leaving the memory management responsibility to the operating system. Typically, LRU or Working Set based algorithms are used to manage the main memory pages [CH81], exploiting the *spatial and temporal locality* of program and data accesses. Under the UNIX operating system allocation of code and data is up to the applications. Typically the `loader` is responsible for assigning code pages to the virtual memory, and the application libraries for allocating memory for data (through the `malloc()` routine).

In the context of the virtual memory system managed by some policy, it makes sense

to allocate the program code on pages in such way that the access patterns induced on the virtual memory *matches* the ones the management policies expect. Hatfield and Gerald in [HG71], and Ferrari in [Fer74] and in [Fer75], suggested *restructuring* the program code to match the LRU or the Working set policy. A proposed algorithm called "critical LRU" (CLRU) performs program restructuring based on a reference string. CLRU does this restructuring by taking into account the underlining virtual memory management policy, in effect "tailoring" the program to fit the LRU policy. *Critical Working Set* (CWS), a similar algorithm assuming a working set policy is also proposed. The behavior of the program is assumed to be given in the form of a "reference string" a sequence of block requests. Both algorithms derive a "nearness" matrix from that string, which is used for a subsequent matrix clustering algorithm to assign blocks to pages.

### 3.6.4 Comments on Related Work

The *abstract clustering problem* we proposed is a unifying approach to clustering, exposing the fundamental problem behind all the above situations. The rigorous mathematical formulation introduced enabled us to identify and prove the problem complexity and to derive several algorithms for solving it or approximating it. We used general models to represent the access patterns and the costs involved, and stated clearly the performance objectives. By contrast, the above methods are heuristics heavily depended to specific access styles, "object graphs", and system properties.

With respect to the work on program restructuring, the formulation of abstract clustering problem is mathematically much more precise than the definition of the restructuring problem as presented in all three papers ([HG71], [Fer74] [Fer75]). When the locality of 1-page cache is considered, the nearness matrix produced by the CLRU or CWS methods will be exactly the same as the SMC graph. In the case of a multi-page cache however, both CWS and CLRU generated graphs overestimate the benefit of assigning two objects in the same page. In fact, this deficiency has been recognized by the author ([Fer74] page 615). As a result, the cost of any partition will not be correct. Finally, it was recognized that restructuring is related to graph partitioning, but neither a formal reduction, nor a multi-page cache generalization was ever presented.

## 3.7 Conclusions

In the previous sections the advantages of the stochastic approach in clustering have been demonstrated. A realistic system model has been introduced, and the clustering problem has been given a formulation precise enough so that we can rigorously prove optimal clustering for

object bases is an NP–complete problem. Furthermore, by reducing the problem to a well–studied weighted graph partitioning problem, clustering strategies can make use of existing heuristic solutions or variants thereof in order to improve the performance of OODBMS systems. Finally, our experimental results have verified that the stochastic approach was indeed correct in terms of improving the locality.

We have pointed out that object clustering has many similarities with data access problems that have been studied in the past. The key difference between those problems and clustering in OODBMS is in the amount of static, a priori knowledge available about the persistent objects, as well as in the amount of statistical information accumulated over their lifetime. This information, along with the semantics of object oriented systems that restrict the usage of object references to pre-compiled methods, make stochastic clustering techniques extremely attractive. OODBMS will be used to store and retrieve large amount of persistent data, possibly residing in distributed servers. Consequently good clustering techniques will be a necessity, and it merits a careful investigation of the underlying problems.

In this chapter, we have focussed on what we view as the fundamental underlying problem for clustering algorithms. Much important work remains. Some of the assumptions that we have made need to be further examined. For example, the modeling of client requests remains open, and alternative models (both stochastic and syntactic) should be investigated. Answers to questions such as: "how well does the SMC model approximate real-world OODBMS application reference patterns" must await experience with running systems; using the foundation laid by the work presented here, we will be able to make use of these answers as they become available.

Applying the above methodology to a "real system" is not an easy task. Many important questions such as, what to consider as usage data, how often re-clustering should be done, how to deal with changes of the object store, and how to cluster in the absence of usage data, need answers. But using the presented framework, one can formally define for the first time, what it is that needs to be optimized, how to optimize it, and how to measure the optimality of solutions.

# Chapter 4

# Peformance Evaluation of Clustering Techinques

In this chapter 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.

## 4.1 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 chapter 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 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 chapter is structured as follows: Next we give some background information on the problem of clustering. Section 4.2 describes our simulation environment used to run the experiments, and Section 4.3 describes and explains the algorithms tested. In Section 4.4 we discuss the results of the experiments, and in Section 4.5 we summarize our experience and draw conclusions for the performance of clustering algorithms.

### 4.1.1 Clustering Quality Metrics

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 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 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:

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

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 $N - C$ 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

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

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 $(X_n) = x_1, x_2, ...x_n$, the working set at time $t$ is defined as the cardinality of the set of the last $w$ requests:

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

By representing access patterns as stochastic processes, clustering can be formulated as an optimization problem [TN91], and optimal clustering corresponds to the clustering mapping that minimizes the expected working set size. The complexity of "optimal clustering" in general (which we have shown to be NP–complete in Chapter 3), makes it very hard to find the optimal clustering mapping. However, this formulation gives a new view to the problem, helps to come up with some more reasonable (close to "optimal") clustering algorithms, and also gives a practical limit to how much clustering can help the 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.

## 4.2 Methodology

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.

### 4.2.1 Cache Simulation and Performance Metrics

Our experiments we performed in CLAB an experimental clustering laboratory we designed, and is fully presented in Appendix B.1. Each clustering mapping was tested using a client

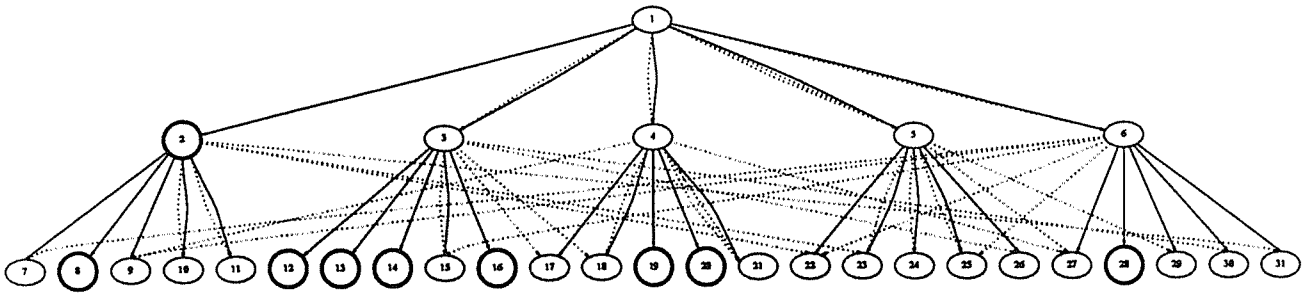


Figure 4.1: The Tektronix Benchmark Object Graph
*The graph shown is only a part of the actual object graph used in some of our experiments. Dotted lines denote the parts edges, dashed lines the children edges*

simulator as in [TN92], 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 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 4.1.1). $EF$ 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.

### 4.2.2 Workload description

Our study was influenced by CLUB-0, an earlier performance study performed for the $O_2$ system [HBD91]. The synthetic worklod used is based on the Tektronix Hypermodel Benchmark [And90]. The object graph in CLUB-0 is a DAG, derived from a balanced 5-way tree with some additional edges.[2] Figure 4.1 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 $N = 3906$ 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

[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.

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 by mixing, concatenating, or interleaving query traces.

## 4.3 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 CG's are used:

[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.

- 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. For a formal definition of the object graph please refer to Section 1.2.

- 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 estimated stationary (transition) probabilities of the chain from the sample trace.

Both statistical models are formally defined in Chapter 2. Procedures for estimating statistical models from traces is given in Section 2.5.2.

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). 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 4.1 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. $N$ denotes the total number of objects to be clustered.

The SMC.PRP 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

| 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 | $O(N \log N)$ | [TN91] |
| SMC.KL | $O(N^{2.4})$ | [TN91] |

Table 4.1: Tested clustering algorithms

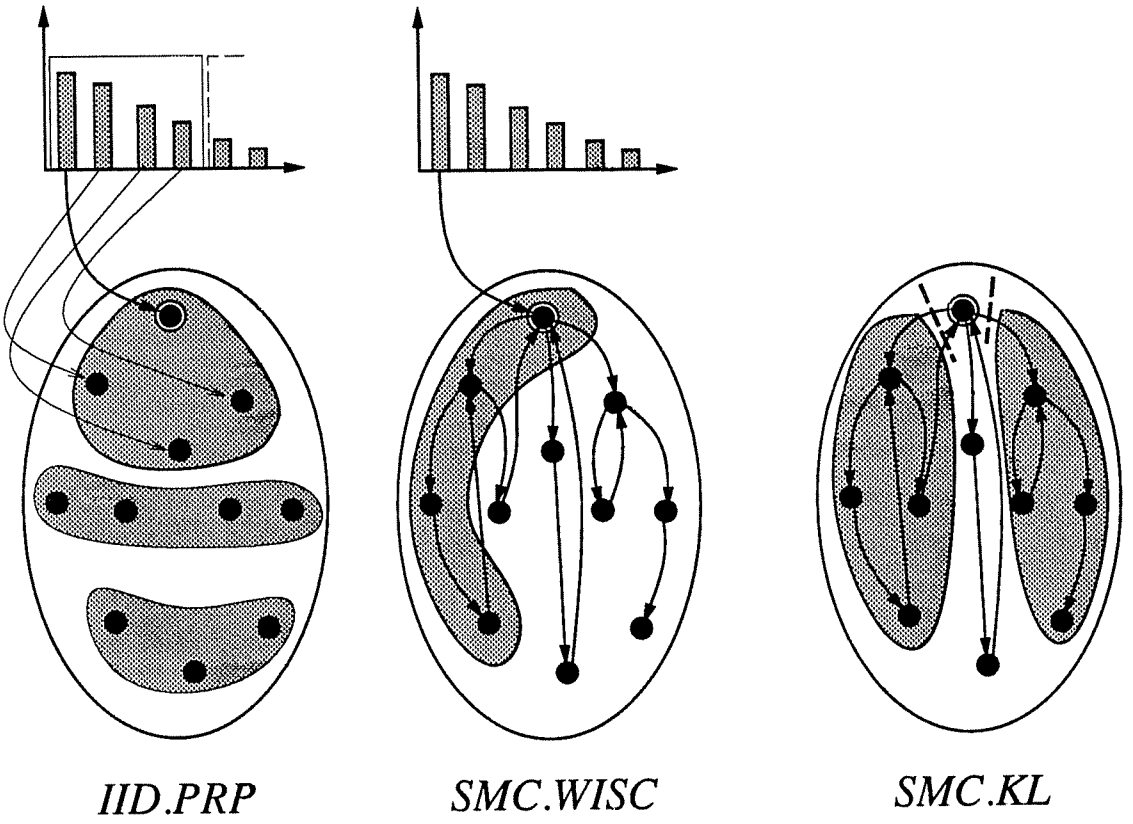



Figure 4.2: Example of applying IID.PRP/SMC.WISC/SMC.KL

(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 and was presented in Chapter 3. The SMC.PRP algorithm has $O(N \log N)$ cost.

The SMC.KL algorithm (fully described in Section 3.5.2), employs the classic Kernighan–Lin graph partitioning algorithm to find a near to optimal clustering of the SMC graph [KL70]. 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 (defined in Section 3.5.1), appears in many cases to approximate well SMC.KL in terms of performance. SMC.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 unclustered 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 unclustered 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 operation of SMC.WISC and SMC.KL is graphically illustrater in Figure 4.2).

The OG.DFS algorithm traverses the object graph in a DFS manner. It minimizes the number of different pages touched during a pure DFS that uses all possible object graph edges. OG.BFS traverses the graph in a BFS manner, grouping siblings together as much as possible. Both algorithms have linear cost $O(N)$ and their pseudocode is given in Figure 4.3 (DFS), and Figure 4.4 (BFS). Figure 4.5 shows their graphical representation, when applied to the object graph shown.

SG.WDFS (shown in Figure 4.6) is much like the OG.DFS except that during the DFS, siblings are selected depending on how hot their edge is. This algorithm attempts to minimize the number of clusters "probable" DFS 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

**Input:** *Object Graph*
**Output:** *Clustering Mapping*

```
initqueue(Q,'LIFO')      // create a LIFO queue for DFS
enqueue(ROOT,Q)          // start from the root of the object graph
P=0                      // current page number

while Q is not empty {
      X = dequeue(Q)
      if (X is clustered)
            continue
      if (no room in page P)
            P++
      assign X to page P
      foreach (children Y of X)
            enqueue(Y,Q)
}
```

Figure 4.3: DFS Clustering Algoritm

**Input:** *Object Graph*
**Output:** *Clustering Mapping*

```
initqueue(Q,'FIFO')      // create a FIFO queue for BFS
enqueue(ROOT,Q)          // start from the root of the object graph
P=0

while Q is not empty {
      X = dequeue(Q)
      if ( X is clustered )
            continue
      if (no room in page P)
            P++
      assign X to page P
      foreach (children Y of X)
            enqueue(Y,Q)
}
```

Figure 4.4: BFS Clustering Algoritm

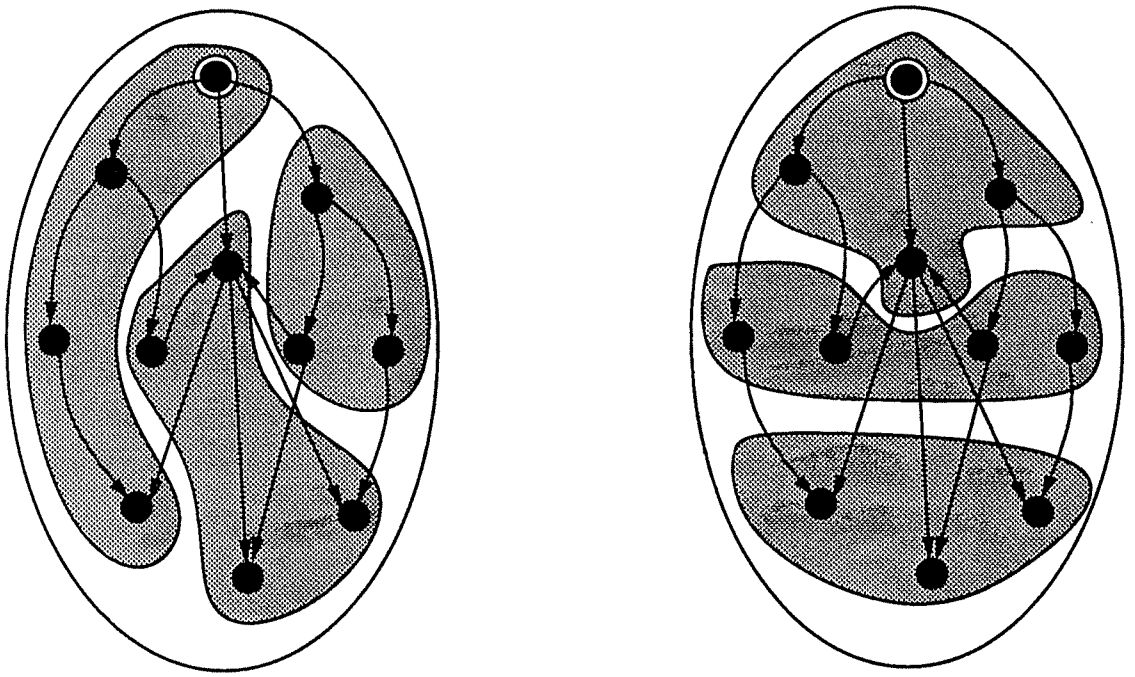



Figure 4.5: Example of applying OG.BFS/OG.DFS
*The picture shows the result of applying the two clustering algorithms on the graph shown. Observe that OG.BFS is optimal for a BFS traversal, whereas OG.DFS is optimal from a DFS traversal with respect to the number of pages crossed during the traversal.*

**Input:** *Object Graph with weighted edges* $q(x, y)$
**Output:** *Clustering Mapping*

```
initqueue(Q,'LIFO')      // create a LIFO queue for WDFS
enqueue(ROOT,Q)          // start from the root of the object graph
P=0

while Q is not empty {
      X = dequeue(Q)
      if (X is clustered)
            continue
      if (no room in page P)
            P++
      assign X to page P
      S = all children of X
      sort S with respect the children edge weights q(X,Y)
      foreach (children Y from S)
            enqueue(Y,Q,PRIORITY)

}
```

Figure 4.6: WDFS Clustering Algoritm

**Input:** *Object Graph with weighted edges $q(x, y)$, prob. vector $\vec{\pi}$*
**Output:** *Clustering Mapping*

```
P=0
sort(π⃗)      // sort objects with respect to their probability

while (there are unclustered objects){
      X = next hottest unclustered object from π⃗
      assign X to page P
      while (there is room in page P){
            S = objects reachable from any object X in P
            Z is the maximum weight edge q(X,Z) from S
            delete Z from S
            if (Z is clustered)
                  continue
            assign Z to page P
      }
      P++           // create a new page
}
```

Figure 4.7: CACTIS Clustering Algoritm

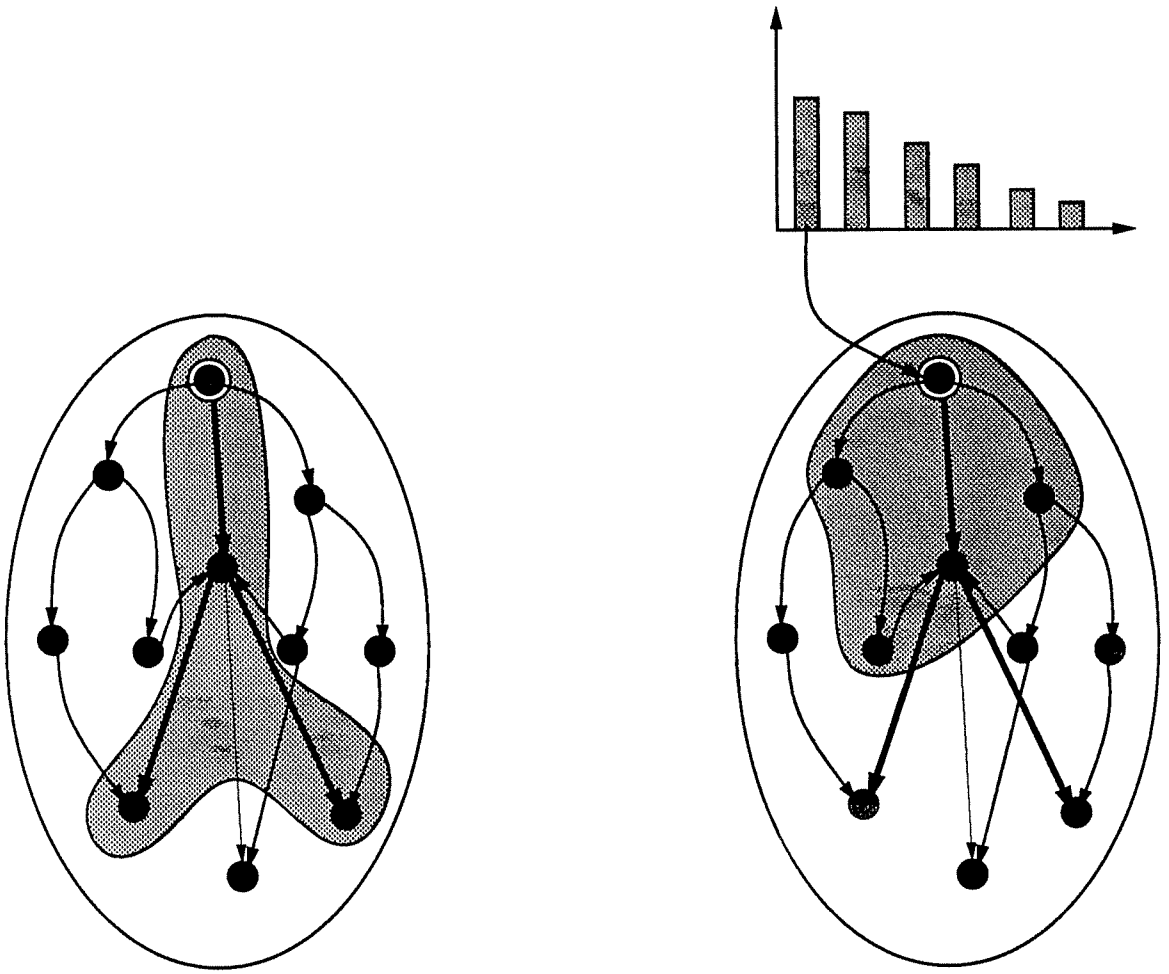



Figure 4.8: Example of applying SG.WDFS/SG.CACTIS
*Observe that, SG.WDFS is optimal for DFS traversals performed with priority specified by the "edge-heat". SG.CACTIS attempts to improve the quality of PRP clustering by clustering structurally related objects of similar temperature.*

```
Input:   Object Graph, clustering pattern V=[v1, v2, ...vn]
Output:  Clustering Mapping

P=0

for (every object X in the BFS order of the Object Graph){
      if (X is clustered)
            continue
      assign X to page P

      // group all objects that match the
      // placement tree V using X as a root
      for ( i=1 to n ){
            Y = match(X,vi)    // Y matches pattern vi starting at x
            if (no match)      // no match can be found for vi
                  continue
            if (Y is clustered)
                  continue
            if (there is no room in page P)
                  P++
            assign Y to page P
      }
}
```

Figure 4.9: Placement Trees Clustering Algoritm

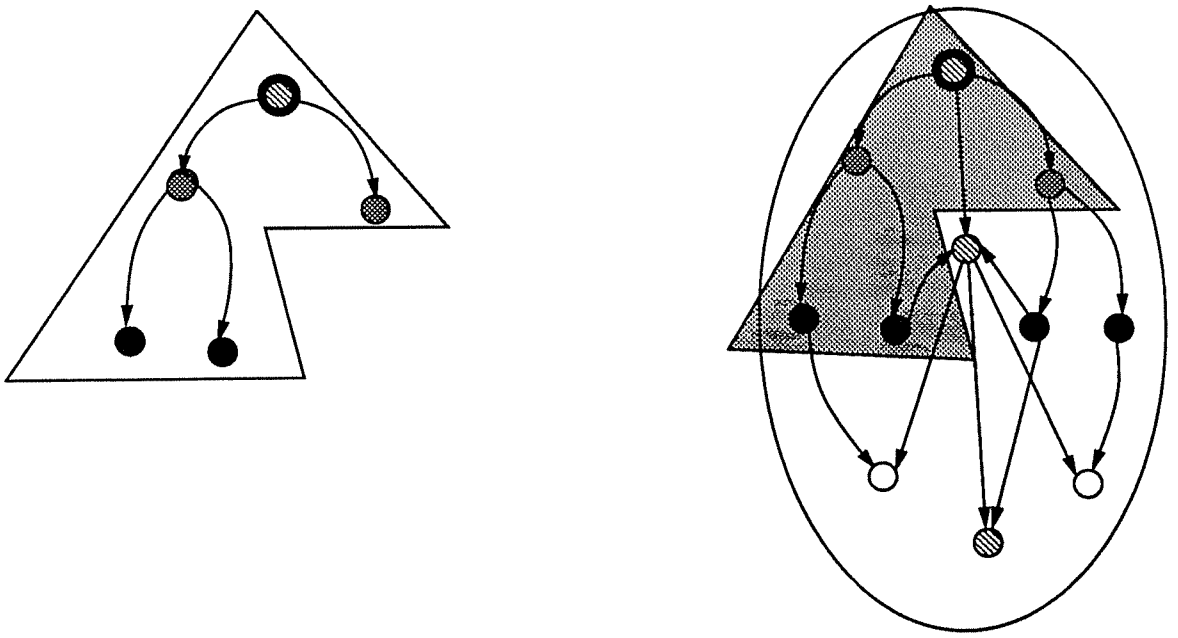
gets its hints from the sample traces. SG.WDFS has linear cost $O(n)$. OG.BFS, OG.DFS and SG.WDFS have been proposed first in [Sta84] for clustering Smalltalk objects.

The SG.CACTIS (shown in Figure 4.7) is based upon the clustering algorithm proposed in [HK89], and it is 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." (Refer to Figure 4.8 for a graphical description of SG.WDFS/SG.CACTIS algorithms).

Finally, we have implemented OG.PT (shown in Figure 4.9), an algorithm based on



*OG.PT*

Figure 4.10: Example of applying placement trees clustering
*The placement tree pattern shown on the left is applied to the object graph, and the resulting matched objects are assigned to the same page.*

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 unclustered 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.

## 4.4 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

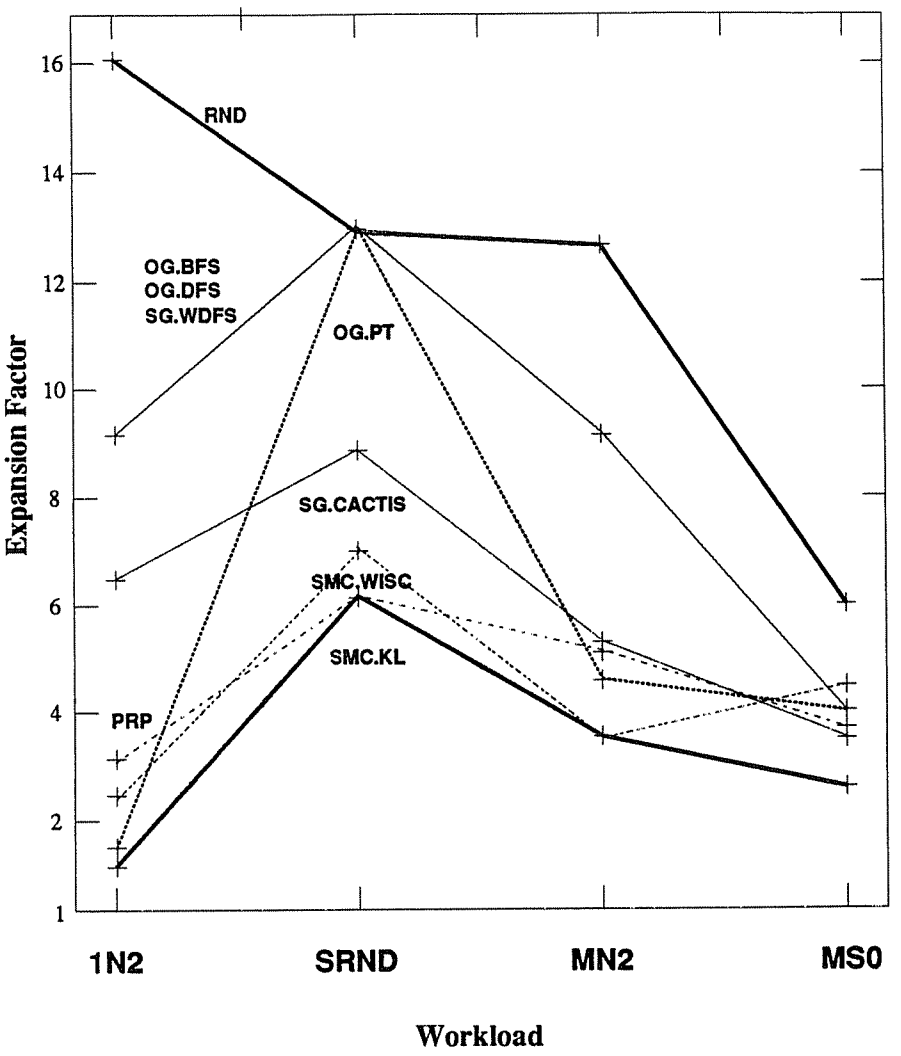



Figure 4.11: Performance on pure workloads – single traversals

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.

### 4.4.1 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.

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

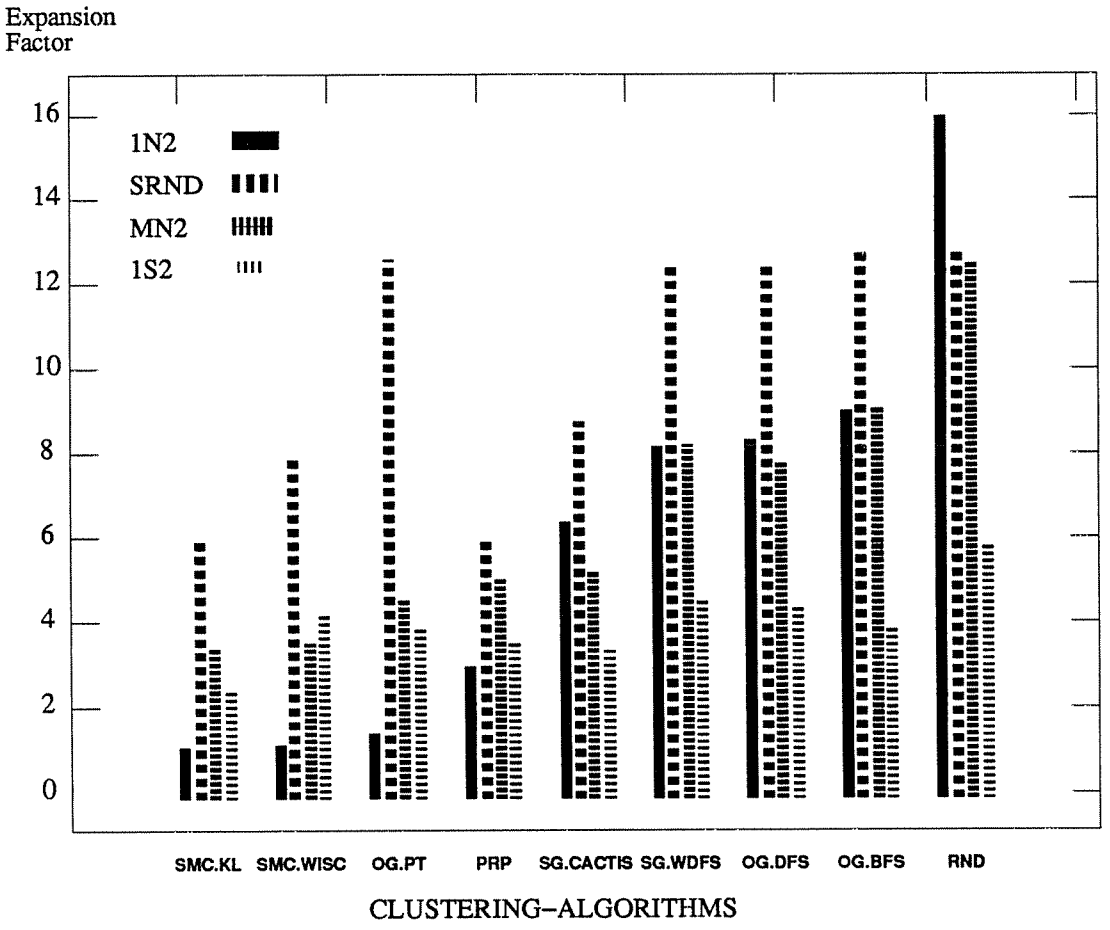



Figure 4.12: Performance on pure workloads – Bar Graph Version

$EF$ 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 4.11 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 better as fewer of the objects are accessed. Statistical 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,

[4]We will postpone the explanation to Section 4.4.2

| Algorithm | CPU Time) (sec) |
|---|---|
| OG.DFS | 2.9 |
| OG.BFS | 3.0 |
| OG.PT | 52.2 |
| SG.WDFS | 3.0 |
| SG.CACTIS | 3.1 |
| SMC.PRP | 1.9 |
| SMC.WISC | 3.0 |
| SMC.KL | 2042.4 |

Table 4.2: Run Time costs of clustering Algorithms
*The table shows the average cpu time of the clustering algorithms, applied on the MN2 workload with N=3906 objects and L=20 objects/page.*

PRP is using the different frequencies of objects belonging to different traversals to "learn" the structure of the object graph. A more elaborate discussion on this can be found in Appendix B.3.

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.

### 4.4.2 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

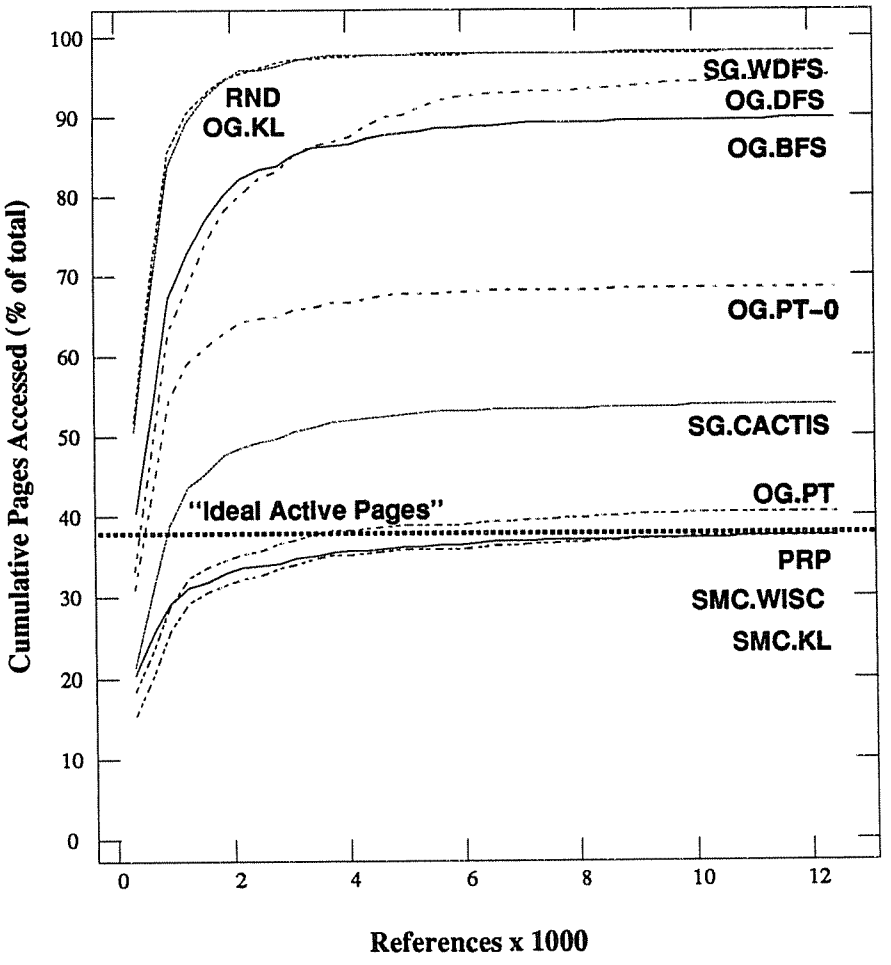



Figure 4.13: mn2 page faults as approaching the steady state

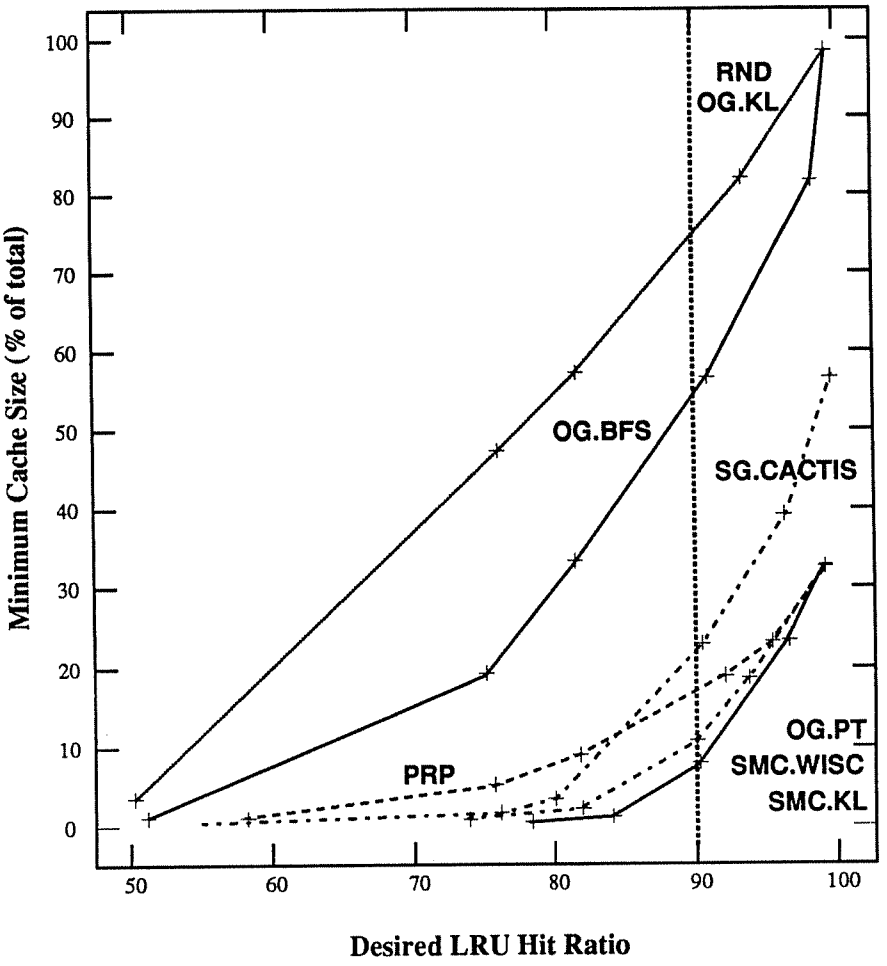



Figure 4.14: Min cache size to achieve a hit ratio in the steady state

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] thereby producing 2800 mn2 references.

To study those steady state effects we used the MN2 workload.[6] As the results of Figure 4.13 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 4.1. 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 4.1. 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 B.2).

The graph of Figure 4.13 shows the cumulative number of pages accessed during mn2 traversals as a function of the number of references, using a large initially empty cache,

[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.

[6]Table 4.2 depicts the average cpu times each clustering algorithm, when running on a Digital Equipment Corporation DecStation 5100. Times do not include any statistical processing necessary for estimating the access models from sample traces.

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 (this technique was suggested by Mendioroz in [Men91]). 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 $EF_\infty$ is an indicator of the steady state performance. $EF_\infty$ is the ratio of pages accessed in the steady state ($N_\infty$) to the number of pages that would be required ideally to pack all active objects ($n_\infty$):

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

In the object graph we tested, there were actually 1498 accessible objects through the parts edges, corresponding to 38% of the object graph[7] represented by the dashed line in the graph of Figure 4.13. $EF_\infty$ is definitely related to $EF$; $EF$ measures the average packing capability for each one of the possible traversals that influence $EF_\infty$. In our case, since there are 25 possible equiprobable traversals, $EF_\infty$ cannot be worse than $25EF$. However, large $EF$ does not necessarily mean large $EF_\infty$. 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 $EF_\infty$ will be the optimal since no inaccessible objects will be clustered with accessible objects.

If only $EF_\infty$ 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 $EF_\infty$. 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 4.11), their $EF_\infty$ was not much worse than optimal, due to sharing of pages between traversals.

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

The graph of Figure 4.14 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 $EF_\infty$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 $EF_\infty$ 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.

### 4.4.3 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 scaleup 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. 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 of course. The third experiment uses just two graphs and it will be explained later (Table 4.3 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 4.15 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 4.16 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 the object graph size, there are some surprises, because of the changing traversal localities.

| 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 4.3: Scale-up queries

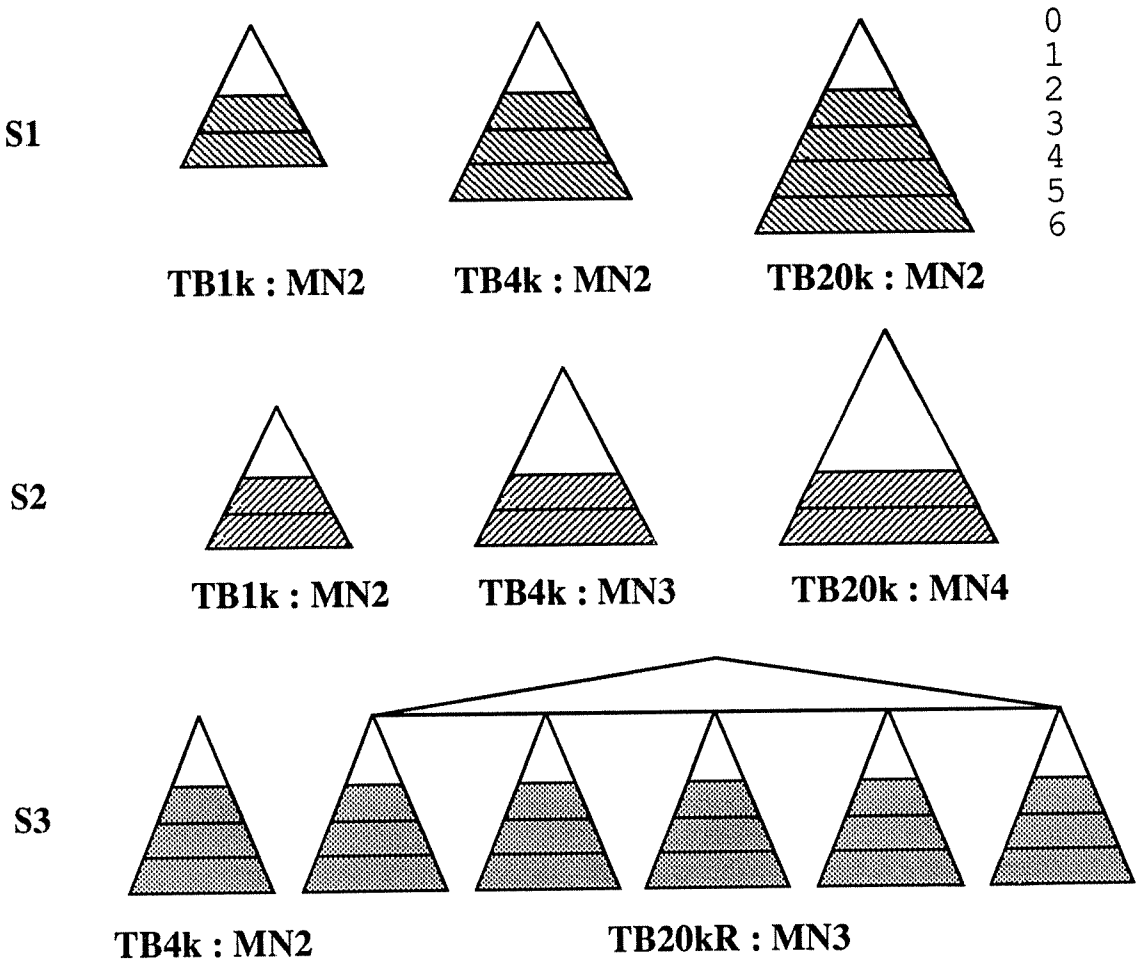



Figure 4.15: The scale-up experiments

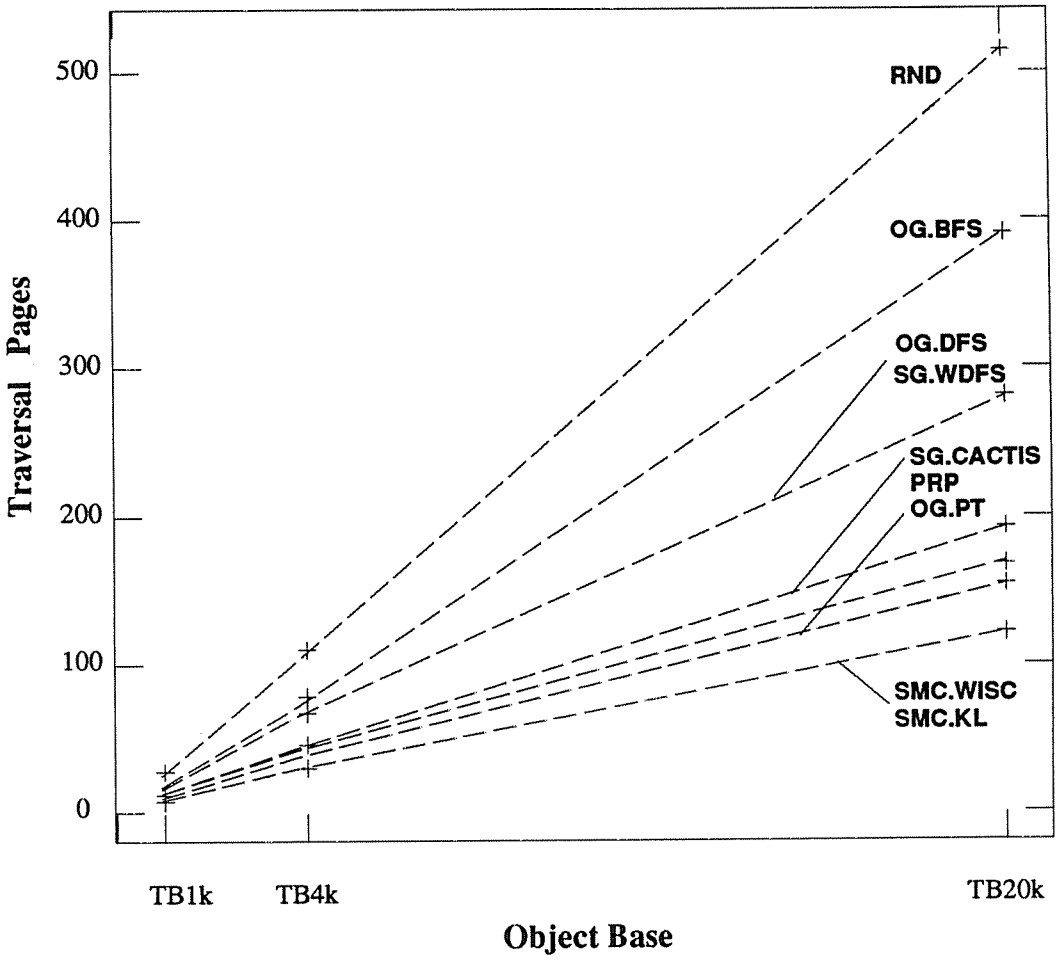



Figure 4.16: Scale-up Experiment #1

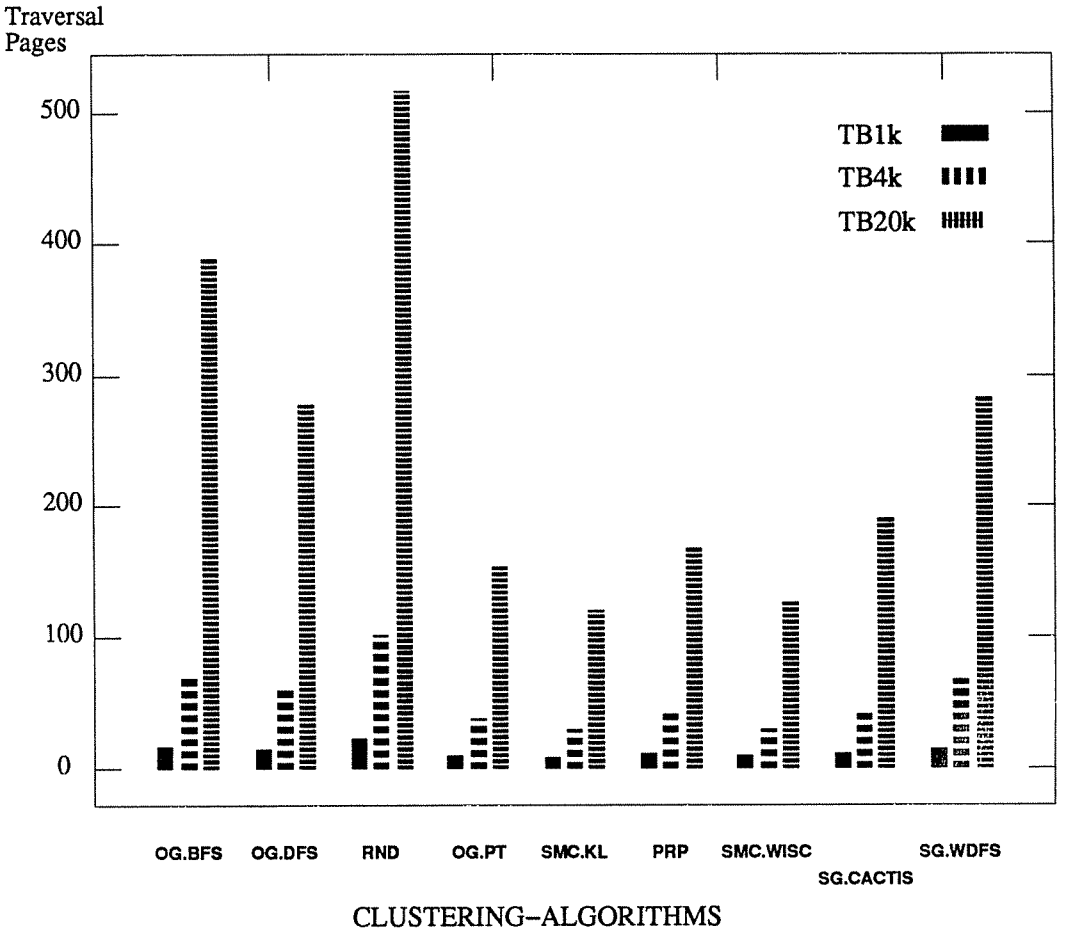



Figure 4.17: Scale-up Experiment #1 - Bar Graph version

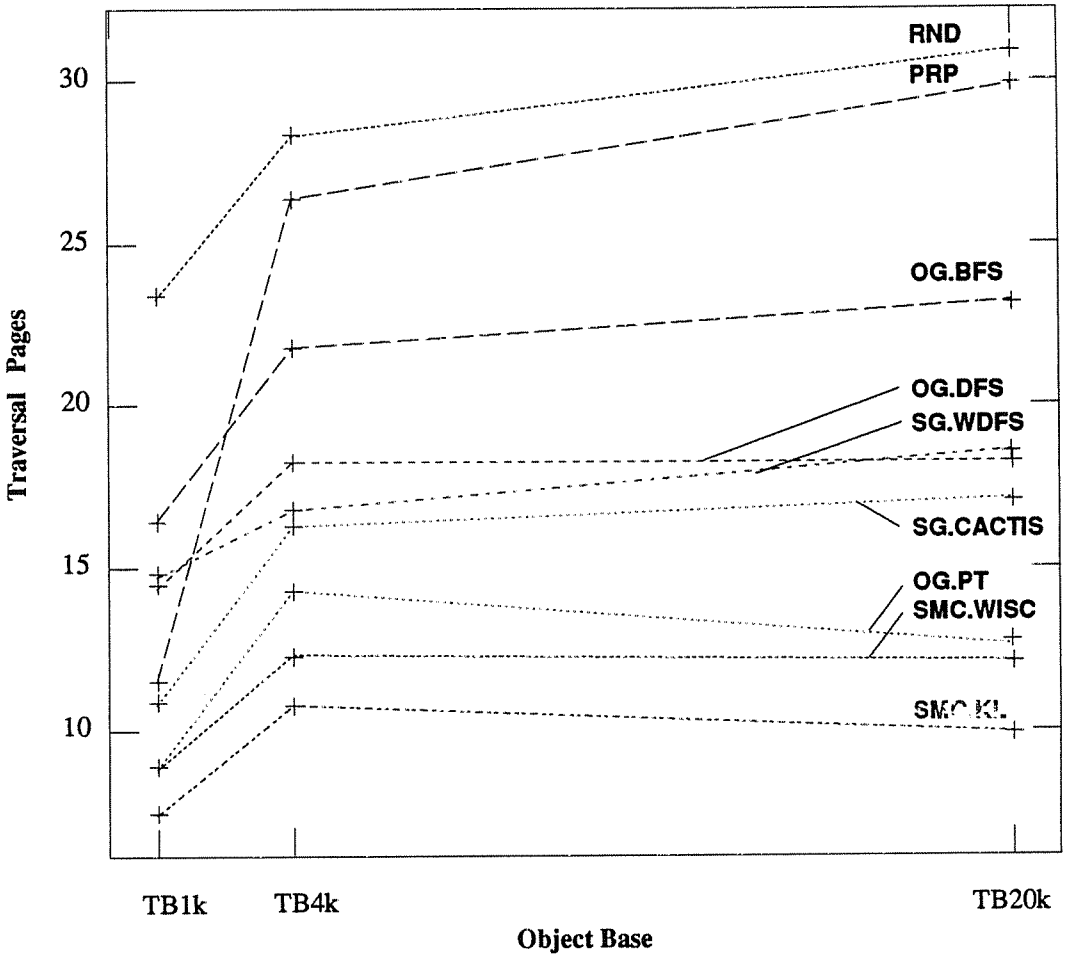



Figure 4.18: Scale-up Experiment #2

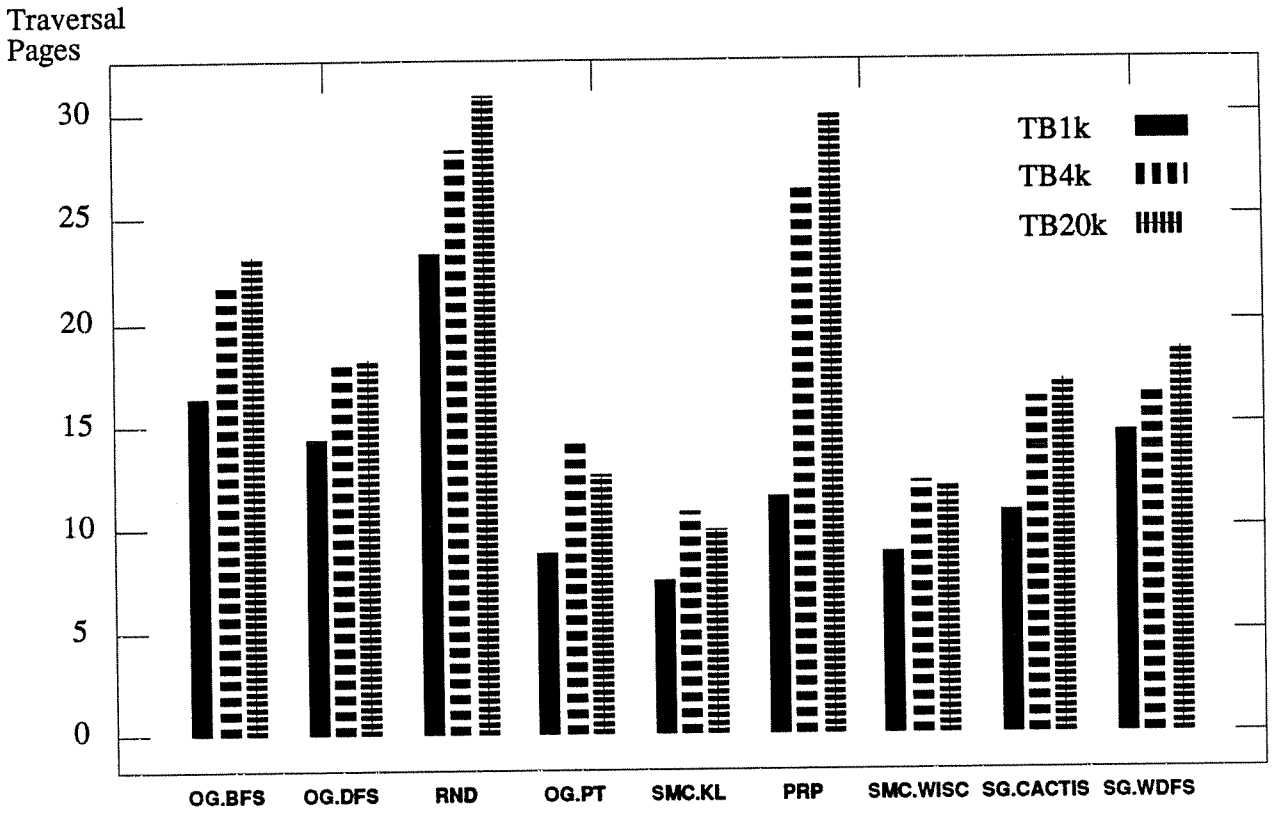



Figure 4.19: Scale-up Experiment #2 – Bar Graph Version

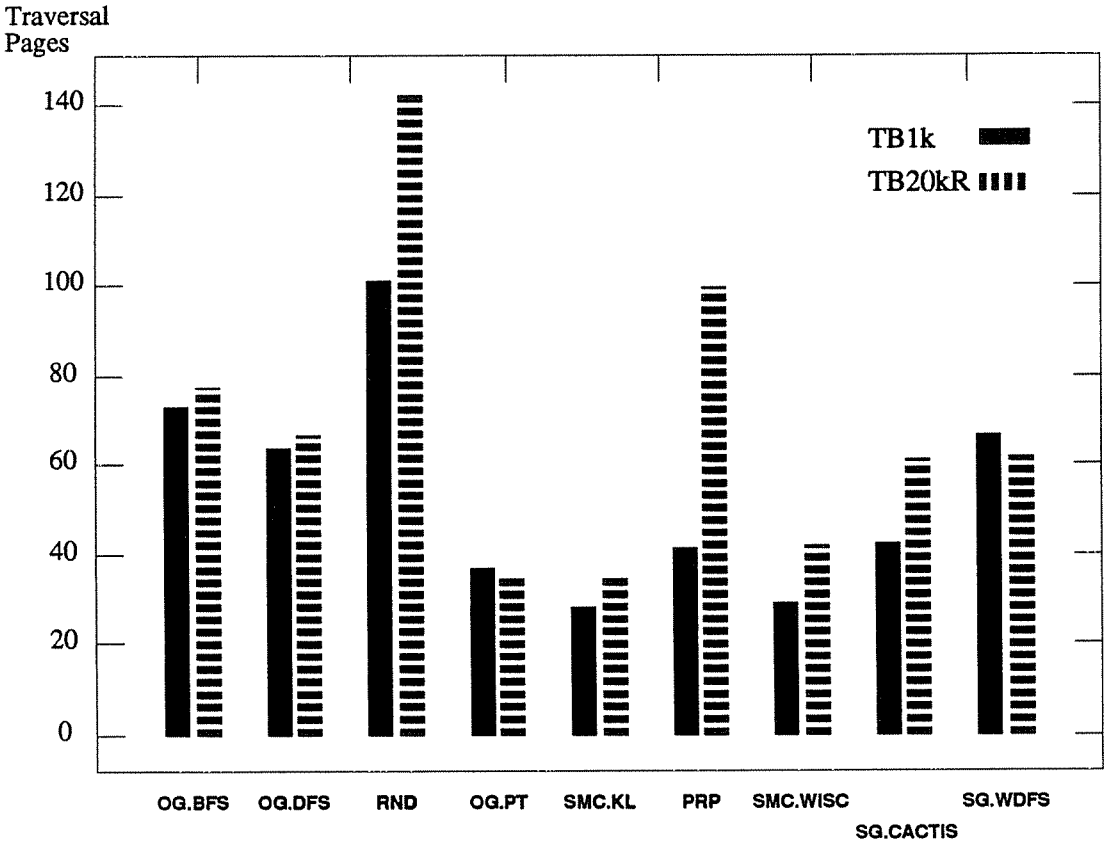



Figure 4.20: Scale-up Experiment #3

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 4.18 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 4.4.1. 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 is a 7–level object graph constructed using 5 disjoint TB4k graphs as Figure 4.15 clearly shows. The algorithms were tested using mn3 on TB20kR and mn2 on TB4k, to ensure that on both graphs 756 objects are accessed. The mn3 traversals have much less inter–traversal locality than the mn2 ones, since for each one there only 4 other traversals that can possibly have common objects. It is guaranteed that the rest of the 620 possible traversals will be disjoint, because they belong to different subtrees. Clustering algorithms should be able to take advantage of the separability of subtrees on TB20kR–mn3, so their traversal pages would should not be more than on TB1k–mn2.

It is interesting to observe that even the simplest structural algorithms like OG.BFS and OG.DFS are not affected much by the factor of 5 increase in size (see graph of Figure 4.20). PRP however, quickly deteriorates by the same factor as random clustering! 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. 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. On the other hand, because of the uniformity, objects on different subtrees may very well have similar probabilities, and as a result PRP will mix them.

### 4.4.4 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

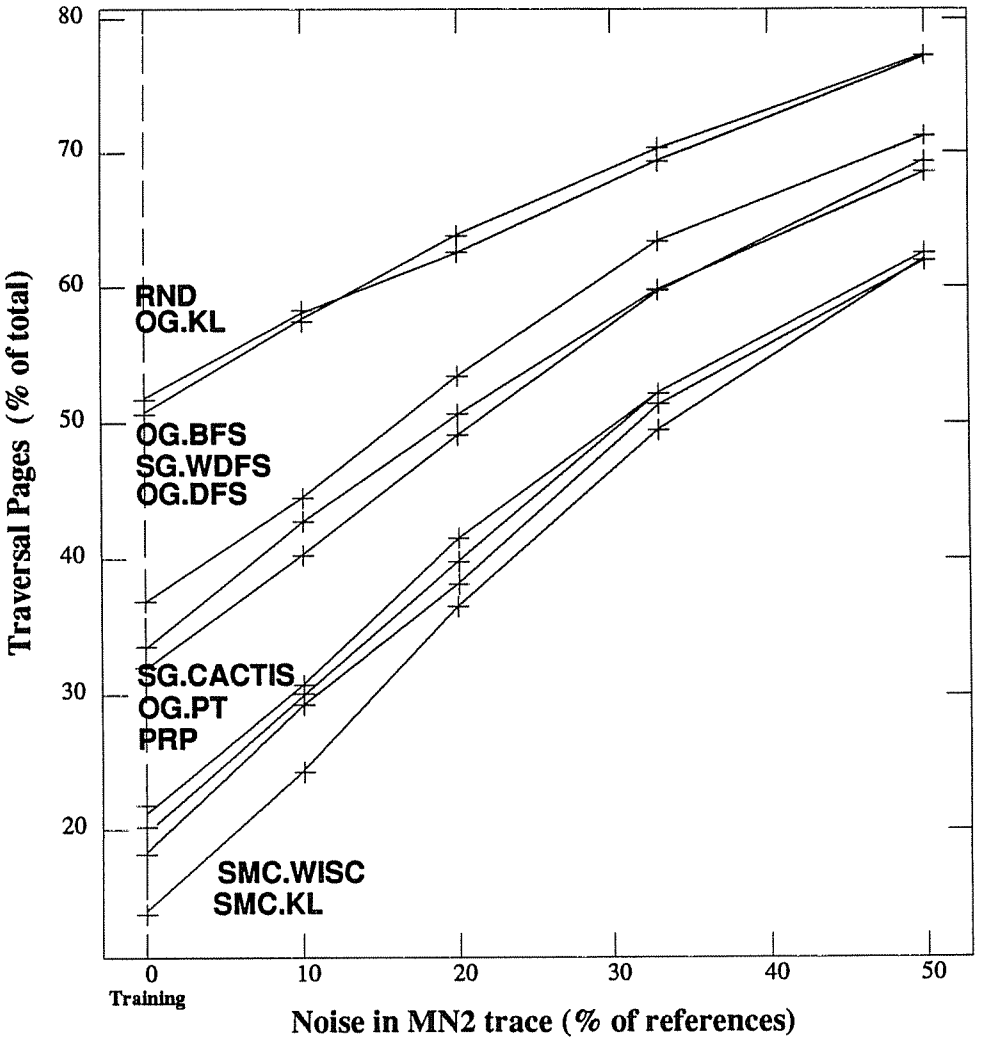



Figure 4.21: "Noisy" references in mn2 queries

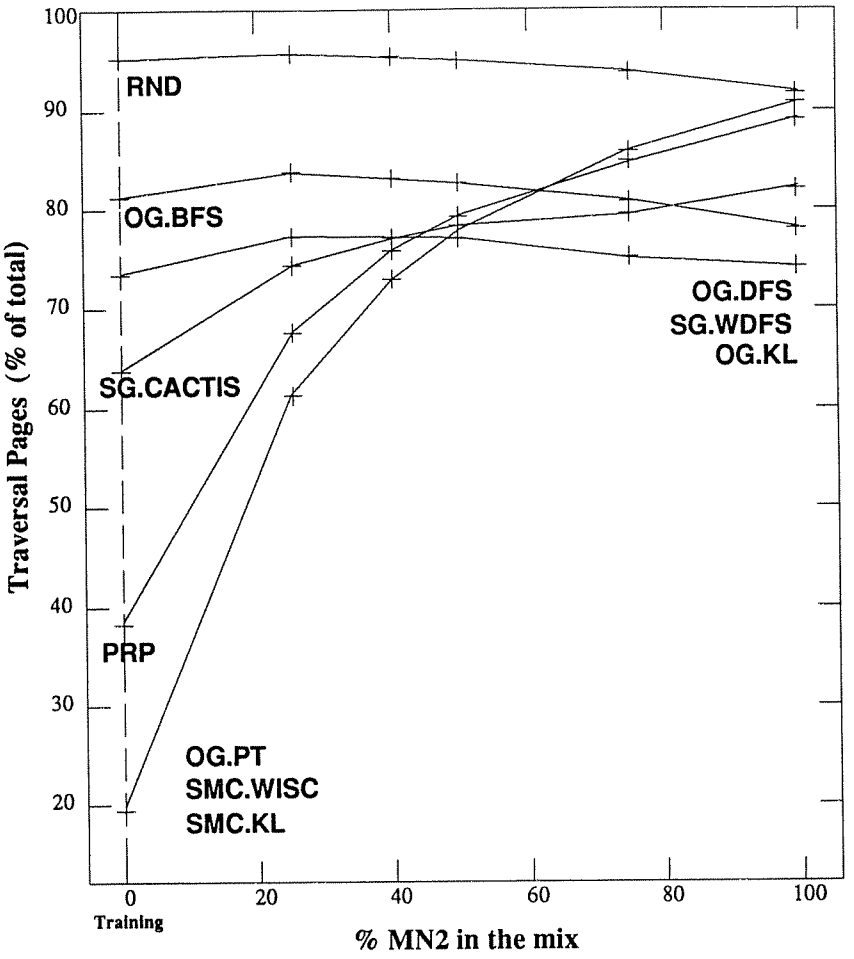



Figure 4.22: Changing access patterns for 0:100% training

the ones used for training. One way real access patterns may be different, is that some unexpected references may appear in the actual trace. To study this effect we added white noise to the testing object stream as suggested in [PZ91]. *White Noise* is a stream of random references chosen with uniform probability from the whole object population.

The graph of Figure 4.21 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.

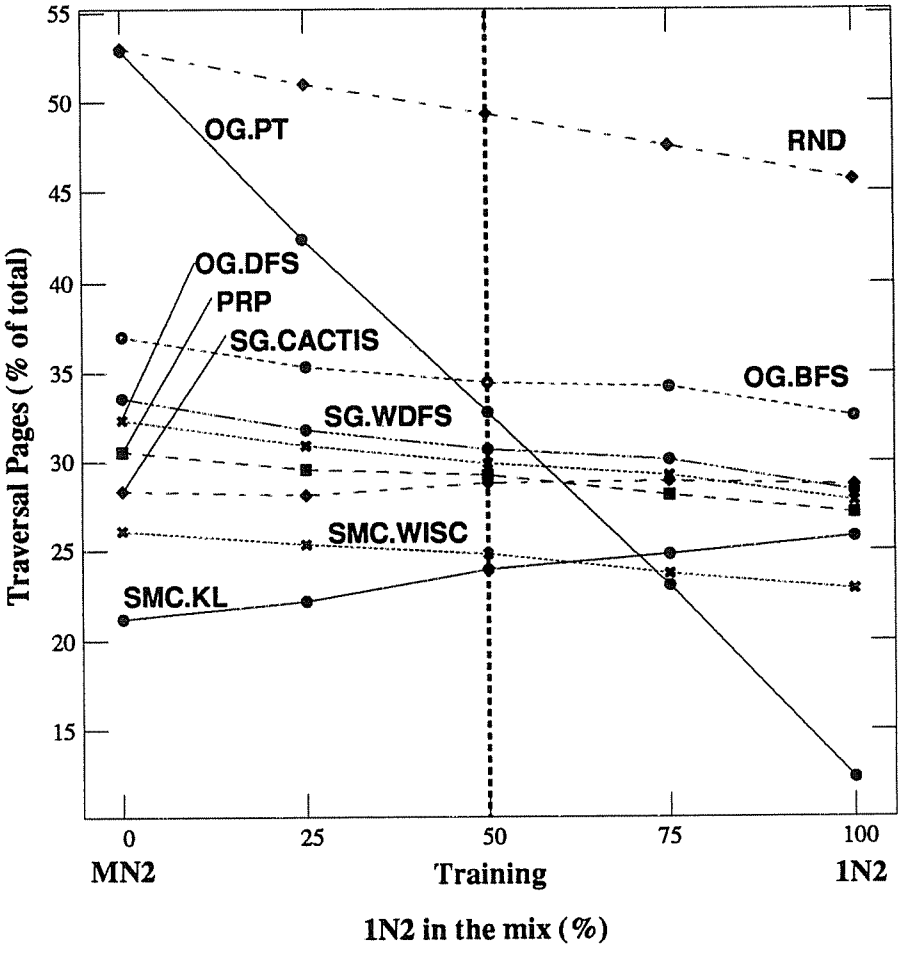



Figure 4.23: Changing access patterns for 50:50% training

### 4.4.5 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 two experiments were done to investigate it.

In the first 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 DFT starting from the same randomly selected nodes, they follow different edges in the object graph, and generate different access patterns. The graph of Figure 4.22 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.

In the second experiment the clustering algorithms were trained with a 50:50% workload mix from 1n2 and mn2 traversals, and were tested with a varying mix of 1n2 and mn2 traversals. With this experiment we wanted to investigate the sensitivity of clustering to variations of the testing workload components, with respect to the training workload. This setup was motivated by real life situations, where changes to usage of an application causes changes like those in the access patterns. The graph of Figure 4.23 shows the (average) number of traversal pages as a function of the mix proportion. The striking difference between this graph and the graph of Figure 4.22 is that there is no significant change of the performance for any algorithm but the placement trees. For the same reasons as before, non-statistical algorithms are practically unaffected. The only explanation we can offer for the stable performance of the statistical algorithms, is that they have "seen" both workloads and produced a compromised clustering mapping that handles both.

The most interesting effect is with the placement trees clustering (OG.PT curve). It appears that the placement tree chosen for the 50:50% case actually benefited the 1n2 much more than the mn2 workload. This placement tree had to compromize between two different conflicting clustering objectives: favoring mn2 traversals and favoring 1n2 traversals. Given

a node of the object graph, the mn2 query will access a different set of children than 1n2, and it is impossible to come up with a balanced assignment using just placement tree rules. Even if we do so for one node, due to structural randomness of the parts edges, it will not be as good for a different node. After a long experimentation, we found that the best placement tree we could come up with for the 50:50% case was the one that gave priority to 1n2 rather than the mn2 traversals.

From the first experiment, one can conclude that simple structural algorithms (like coarse filters) are not bad for dealing with highly unpredictable access patterns. Good algorithms (which behave as finer filters) are more sensitive, and great care should be taken when using them, since changes in the access patterns may affect them dramatically. Statistical algorithms will perform well, when all components of the testing workload were used in the training workload, perhaps with different weights, as it was the case with the second experiment. They are less robust when the testing workload "shifts" to a new component (like in the first experiment).

## 4.5 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. Apart from SMC.WISC a good alternative is SG.CACTIS and OG.PT although placement trees are not always easy to come up with.

- 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 the correct 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.

# Chapter 5

# Conclusions

## 5.1 Thesis Summary

This thesis represents a performance oriented approach to the problem of clustering in the context of Object Oriented Database Systems. To achieve this, we first examined the architecture of typical OODBMS's, presented a system model, and discussed the distinction between clustering mechanisms and the clustering policies they provide (Chapter 1). We argued that an OODBMS functions as a three–level memory system, with up to four orders of magnitude speed difference between the three levels (with current hardware and software technology). In such an environment, the effectiveness of all memory levels is greatly affected by access properties, mainly locality of references.

Recognizing that access patterns play a significant role in the performance of this three–level memory system, we analyzed carefully the access generation process and pointed out three interesting properties of accesses (Chapter 2): the restricted scope, the structural locality, and the temporal locality. We introduced and motivated three statistical models suitable for describing those access patterns.

In Chapter 3 we re-examined the OODBMS model, and taking into account the access properties described in the second chapter, we proposed a set of reasonable clustering objectives for multi-client OODBMS's. We discovered that all those objectives were instances of the same fundamental problem, the "abstract clustering" problem, whose solution depends on the access pattern model. If accesses are uncorrelated, optimal clustering can be found in $N \log N$ time (where $N$ is the number of objects to be clustered). If they are serially dependent (i.e., accessing an object now affects the object to be accessed next), the problem is very hard. In fact, we rigorously proved that abstract clustering under correlated access patterns is an instance of a graph theoretic problem "graph partitioning", and it is *NP–complete.* Exact knowledge of the access stream, large caches, or more complex access correlations do not change the complexity of the problem. Taking advantage of the graph reduction, we

proposed two new heuristic algorithms for object clustering, a greedy one (SMC.WISC) and a more accurate one (SMC.KL).

To test our approach as well as some heuristics suggested in the literature, we conducted a performance study of clustering techniques. Our main observation was that, although other techniques on occasion performed very close to stochastic clustering, in general they were not reliable. Their performance was affected by the access patterns, and no single one could be used as a replacement of stochastic clustering. We also observed that statistical clustering algorithms in general adapt well to the access patterns and they are relatively insensitive to workload changes. Efficient statistical algorithms, however, may perform very badly when a totally new access pattern appears. Finally, if the system has large caches and sub-optimal short-term performance can be tolerated, we found that the probability ranking partition algorithm (SMC.PRP) (which has a reasonable logarithmic cost) is a very good heuristic. PRP achieved a nearly optimal long term performance.

Our approach can be used as an upper bound of the obtainable performance through static clustering, assuming that clustering resources are unlimited. But how far are we from solving the practical problem? The next sections give some thoughts on this question, motivating and suggesting new clustering heuristics and discussing remaining problems.

## 5.2 New Clustering Heuristics

In this section we want to summarize our experience, by proposing heuristics for object clustering, derived from the theory we developed in Chapter 3, and the performance study presented in Chapter 4.

As it is intuitively expected, random clustering (or no clustering at all) should be avoided, since it has the worse possible performance. Simple structural clustering techniques (i.e., the ones based on the object graph), can be implemented easily and can perform much better than no clustering.

Assigning equal importance to all object graph edges, and attempting to optimally partition the object graph, does not appear to be a good heuristic. As we saw in the performance study, "optimal" (in the sense of the Kernighan-Lin algorithm OG.KL) object graph clustering performs almost as bad as random clustering. The explanation we can offer is that complex graphs like the ones in the Tektronix Benchmark, contain a large number of edges that confuse the partitioning algorithm. Even traversals that uniformly follow all edges may in reality use some edges more often because of sharing.

Simple structural techniques like OG.DFS and OG.BFS are usually effective when all edges are used during each traversal. In particular, OG.DFS reduces the inter-traversal sharing and decreases the per traversal expansion factor (short term performance). In contrast,

OG.BFS has a better long term performance at the expense of the short term performance, because it increases the inter-traversal sharing. However, DFS and BFS cannot cope with queries that use a subset of the edges and have different probabilities of being submitted. They do not have a way to distinguish between different edges.

If knowledge of object traversals is available, and object instances have the same access behavior, then placement trees are extremely attractive. As we saw in Chapter 4, OG.PT achieves a very close to the optimal performance in simple workloads that access uniformly the object graph. Placement trees do not seem applicable if accesses are induced to a set of objects from outside of the object graph (e.g., through an index structure). The major problems with placement trees, however, is how to come up with them, and how to resolve placement conflicts in case two different placement trees can be applied to cluster the same set of objects.

High fanout object graphs are not hard to cluster, assuming we know which edges are actually used most of the times by the traversals. Traversing such graphs generates short access correlations (e.g. from a parent to a child and from there to a grand-child), and PRP clustering optimizing long term performance is appropriate. Optimizing for short traversals in a high fanout graph will be very ineffective, since many different possible traversals will be performed in such graph.

*Separability* is an important property of object graphs. If for any reason we know in advance that non-intersecting subsets of the graph will be accessed independently, then clustering each component independently is the right thing to do. Taking advantage of separability not only simplifies the clustering problem of the object graph, but also allows the clustering algorithms to run faster than if they had to be applied on the whole object graph. Separability manifests itself when the object base contains "natural partitions" like files, or when large graph components are "almost disconnected", because the edges connecting them are rarely used by traversals.

If the database consists of collections of identically structured complex objects, then clustering can also be replicated. Assuming that each complex object has a size larger than one page, it would be sufficient to produce "optimal clustering" for one such object (using the stochastic clustering algorithms we proposed in Chapter 3), and then cluster all complex objects the same way.

Access statistics are taken from just one sample (i.e., a complex object of a given type), may be biased towards the behavior of that specific instance. Monitoring a set of complex objects with identical structure will certainly generate more reliable statistics. The access behavior of an "average instance" will be determined by properly "merging" statistics from all samples. Subsequently, an optimal clustering will be produced for one such complex

object, and it will be repeatedly applied to each instance.

A highly dynamic environment where new access patterns may show up, can be disastrous for statistical techniques, and simple structural techniques should be preferred then. If the training workloads are representative of typical access patterns, statistical algorithms appear to be stable enough in situations where the workload component proportions change.

## 5.3 Future Work

In this thesis we have examined the problem of object clustering alone, considering its interaction with some data access dependent functions of the OODBMS. In this section, we will step back for a while and analyze how clustering affects (and it is affected by) other system components. Understanding these interactions will enable us both to understand the system constraints, and to realize how much further we must go to solve the practical problem of clustering.

Clustering and *Query Optimization* interact in a very interesting way. In the context of OODBMS, query optimization aims at a) locating "queries" (expressions or code fragments that can be optimized), and b) generating alternative ways to execute those queries and still preserve the OODBMS semantics. The second step generates execution plans that the optimizer predicts will require the minimum amount of I/O traffic, by making cost estimates for each plan. The access cost depends on the physical storage organization of objects, a decision made by the clustering subsystem of the OODBMS, and no longer being easily available as in the relational case. On the other hand, the access patterns generated by different query plans may be different. In fact, they should be quite different, otherwise they may not have variations in their costs. Therefore, query plans may drastically affect access patterns, and thus, make current clustering decisions suboptimal.

Clearly, some reconciliation between the clustering and query optimization subsystems is necessary. For example, cost estimates for accessing members of collections should be provided to the optimizer. If object clustering is applied in an instance-by-instance basis, i.e., clustering different members of that collection in a different manner, it will be nearly impossible to come up with reasonable cost estimates. Perhaps, if clustering is applied in a per-type basis those cost estimates will be easier to calculate reliably. On the other hand, attempting to tune the clustering performance to the access patterns of each different execution plan, may cause a dangerous and resource-wasteful oscillation. A new plan selected by optimizer generates new access patterns, causing the clustering subsystem to reorganize the data, making the previous plan no longer optimal, in turn forcing the optimizer to generate a new plan. A quick (and dirty!) solution is to exclude access patterns generated by query plans from the clustering decisions, but there may be better alternatives.

Some OODBMS provide automatic *Garbage Collection*, and even if they do not, the user must provide his own. Garbage collection removes the "dead" objects and physically reorganizes the "live" objects, to increase the space utilization and the I/O efficiency. If live objects are arbitrarily placed, space may be well utilized but the I/O efficiency will not necessarily be better. If the two space management mechanisms (object clustering and garbage collection) are not effectively combined, it is conceivable that the system performance will be penalized at least by the additional I/O necessary to re-read in and re-cluster the live objects.

The performance of the *Concurrency Control* mechanisms may be affected by clustering decisions. In environments where page locking is used, placing objects from different transactions to the same page introduces "false sharing", which creates unnecessary data contention and reduces concurrency. Clustering should avoid increasing the inter-transaction sharing of conflicting transactions (read/write or write/write), but at the same time it should not penalize non-conflicting (read only) transactions. One strategy could be to use clustering mappings that minimize the inter-transaction sharing, by maximizing intra-transaction locality. In Section 4.4.2, where the performance of multiple traversals was examined, a class of clustering algorithms generating mappings appropriate if each transaction performs a randomly selected traversal. We believe that such *"concurrency aware clustering"* must take into account contention costs in addition to the access costs, by explicitly modeling the transactional read/write behavior in addition to the access behavior. Alternatively, solutions like finer granularity locking should not be overlooked to supplement or perhaps replace the concurrency aware clustering.

Modern OODBMS may be running on powerful workstations, with large memories and plenty of disks. In such environment, using demand paged requests and utilizing just one disk at the time, appears to be a very conservative approach. *De-clustering* mechanisms will spread requests to many disks, and *Prefetching* will utilize both the server memories and their disks by introducing low-priority non-demand traffic. We believe that clustering, prefetching, and de-clustering are strongly related problems, that should not be solved in isolation from each other. Prefetching is a dynamic instance of clustering, deciding which objects are possibly needed in the near future, given the current state of the client. De-clustering is a problem dual to clustering, aiming at using as many disks as possible, i.e. achieving minimum locality in terms of the different disks needed for servicing a series of requests. Furthermore, we should not overlook that both mechanisms operate on page granularity, and object to page placement is decided by clustering.

A final issue is dealing with changes, since the OODBMS behavior will almost never be static. In fact, object bases may evolve in their structure, their contents, or in the way they

are used. Since the goodness of storage management decisions depends on all three factors, ways for *detecting changes* and efficiently *adapting* to them, must be found. In our opinion, this is one of the most difficult problems, which can not be solved satisfactory before enough experience from real world OODBMS has been accumulated.

In an attempt to focus on the "principles of optimal clustering", this thesis has not addressed the above integration issues. We have demonstrated however, that our methodology contributed to the understanding of the clustering problem alone and with better ways to solve it. Using the foundations and ideas presented in this thesis, the above problems can be attacked in the future.

# Appendix A

# Stochastic Processes

## A.1 The Behavior of a Partitioned State Space

We will analyze the behavior of a partitioned state space whose states behave as a first order Markov Chain. Let $S$ be a Markovian state space with stationary probability $\vec{\pi}$ and transition probabilities $P(i,j)$. Let $\Delta$ be a a partition of $S$ consisting of non-overlapping nonempty subsets of $S$:

$$S = \bigcup_{C \in \Delta} C \tag{A.1}$$

The chain $S$ starts at time 0 with some probability vector $\vec{\pi}_0$. Suppose at time $n$ we are given the information that the system is in some state $x$ in partition $C_i$. The probability for this event is given by the formula:

$$\pi_n^{(C_i)}(x) \equiv Pr\{x_n \in C_i\} = \frac{\pi_n(x)}{\sum_{w \in C_i} \pi_n(w)}, \quad x \in C_i \tag{A.2}$$

where $\vec{\pi}_n$ is derived from the initial probability vector $\vec{\pi}_0$ and the transition probability matrix $P$:

$$\vec{\pi}_n = P^n \vec{\pi}_0 \tag{A.3}$$

When $n$ is large (i.e., long after the start of the chain), $S$ reaches its (unique) stationary distribution[1] and:

$$\vec{\pi}_n \rightarrow \vec{\pi}, \text{where } \vec{\pi} = P\vec{\pi}$$

then, the probability vector converges:

$$\pi_n^{(C_i)}(x) \rightarrow \pi^{(C_i)}(x) = \frac{\pi(x)}{\sum_{z \in C_i} \pi(z)} \tag{A.4}$$

[1]Since $S$ is an irreducible and recurrent Markov process with no traps or closed subsets.

The ratio is defined at the limit because we assumed that $C_i$ is non-empty.

The probability that at time $n+1$ the system will be in $C_j$ given that at time $n$ it was in $C_i$ is given by the formula:

$$\begin{aligned} p_n(C_i, C_j) &\equiv Pr\{X_{n+1} \in C_j | X_n \in C_i\} \\ &= \frac{Pr\{X_{n+1} \in C_j \cap X_n \in C_i\}}{Pr\{X_n \in C_i\}} \\ &= \sum_{x \in C_i} \frac{Pr\{X_{n+1} \in C_j \cap X_n = x\}}{Pr\{X_n \in C_i\}} \\ &= \sum_{x \in C_i} \frac{Pr\{X_{n+1} \in C_j | X_n = x\} Pr\{X_n = x\}}{Pr\{X_n \in C_i\}} \\ &= \sum_{x \in C_i} \sum_{y \in C_j} \frac{Pr\{X_{n+1} = y | X_n = x\} Pr\{X_n = x\}}{Pr\{X_n \in C_i\}} \end{aligned}$$

using (A.2):

$$p_n(C_i, C_j) = \sum_{y \in C_j} \sum_{x \in C_i} \pi_n^{(C_i)}(x) P(x, y)$$

Using (A.4) $p_n(C_i, C_j)$ converges to $P_\Delta(C_i, C_j)$:

$$P_\Delta(C_i, C_j) = \sum_{y \in C_j} \sum_{x \in C_i} \pi^{(C_i)}(x) P(x, y) \tag{A.5}$$

*So at the limit, $\Delta$ becomes an induced Markov process with transition probabilities $P_\Delta(C_i, C_j)$[2] given by the above formula. $\Delta$ is an irreducible and recurrent Markov process with no traps or closed subsets.*[3] The stationary probability of that chain is given by the formula:

$$\pi_\Delta(C_i) = \sum_{x \in C_j} \pi(x) \tag{A.6}$$

(A.7)

It can be easily verified that $\vec{\pi}_\Delta$ satisfies the matrix equation: $\vec{\pi}_\Delta = P_\Delta \vec{\pi}_\Delta$. This observation is important, since it allows us to view a partition of the state space as a Markov chain too. However, great care must be taken on what event constitutes a *state* of the induced process.

[2]To avoid subsequent confusion we will use $P_\Delta$ to denote the transition probability matrix for a partition $\Delta$ in contrast to $P$ for the transition probability matrix of the initial state space $S$.

[3]If there is a closed set in $\Delta$, then the initial chain contains it too. But $S$ was assumed to have a single closed set of states only.

## A.2 Calculating $WSS$

Suppose we have a sequence of object accesses that follow IID or SMC distributions (see Chapter 2 for complete definitions). $\pi(x)$ denotes the stationary probability observing $x$ at any time in the trace, and $P(x,y)$ the probability to observe $y$ next given that $x$ was seen before. Let $\pi_F(x)$ and $P_F(x,y)$ denote the corresponding quantities for the IID and SMC models of the page request stream, generated because objects are mapped to pages using the function $D$. $R^{(n)}$ is a random sequence of $n$ pages, $N_F$ is the total number of pages the whole object population maps to through $D$.

### A.2.1 $WSS$ of an IID Stream

We will define the $n$-indicator random variable that assumes two possible values:

$$U(n,x) = \begin{cases} 0 & \text{if } x \in R^{(n)} \\ 1 & \text{otherwise} \end{cases}$$

The cardinalty of the set of $w$ requests $K^{(w)}$ is simply:

$$K^{(w)} = w - \sum_{x=1}^{N_F} U(w,x)$$

Because of independence, the probability not to see $x$ in $n$ consecutive references is:

$$U(n,x) = (1 - \pi_F(x))^n$$

and the expected cardinality of $n$ requests will become:

$$K^{(w)} = w - \sum_{x=1}^{N_F} (1 - \pi_F(x))^w \tag{A.8}$$

It is easy to show that $K$ is minimized for every value of $w$ when each frame $f$ contains objects placed using the probability ranking. Suppose that a partition $F$ that contains the frames:

$$F = \{f_1, f_2, f_3, ...f_i, ..f_j, ...f_{N_F}\}$$

such that $\pi_F(i) \geq \pi_F(j)$, $\forall\, i \leq j$. Now suppose that there are two states $s_1 \in f_i, s_2 \in f_j (i < j)$ such that:

$$\pi(s_1) < \pi(s_2)$$

Then we can define a new partition $F'$ the same as $F$ with $s_1, s_2$ exchanged. $F'$'s cardinality will differ from $F$'s only on $U(n,i)$ and $U(n,j)$ terms. Now:

$$\begin{aligned} U(n,i)+U(n,j) &\leq U(n,i)'+U(n,j)', \text{ where:} \\ U(n,i)+U(n,j) &= (1-\pi_F(i))^n+(1-\pi_F(j))^n \\ U(n,i)'+U(n,j)' &= (1-(\pi_F(i)+\gamma))^n+(1-(\pi_F(j)-\gamma))^n \end{aligned}$$

with $\gamma = \pi(s_2) - \pi(s_1)$. This can be derived from the next formula, that can be shown true for any $n > 0$ by induction:

$$\begin{aligned} x^n+y^n &\leq (x-z)^n+(y+z)^n \\ \text{where } z \leq\ & x \leq y \end{aligned}$$

### A.2.2 $WSS$ of an SMC Stream

An auxiliary random variable used here, will measure the probability of occurrence of a particular state in a sequence of $n$ consecutive states of a SMC model:

$$W(n,x) = \begin{cases} 1 & \text{if } x \in R^{(n)} \\ 0 & \text{otherwise} \end{cases}$$

Then the $K^{(w)}$ is simply the sum of the occurrence variables over all states:

$$K^{(w)} = \sum_{x=1}^{N_F} W(w,x)$$

For $w = 1$, $K$ is always 1. For $w = 2$ there is a simple formula for $K^{(w)}$:

$$\begin{aligned} W(2,x) &= \pi_F(x) + \sum_{y \neq x} \pi_F(y) P_F(y,x) \\ &= \pi_F(x) + \pi_F(x) - \pi_F(x) P_F(x,x) \\ &= 2\pi_F(x) - \pi_F(x) P_F(x,x) \end{aligned}$$

substituting this result in the formula for $K^{(2)}$:

$$\begin{aligned} K^{(2)} &= \sum_{x=1}^{N_F} W(2,x) = \sum_{x=1}^{N_F} 2\pi_F(x) - \pi_F(x) P_F(x,x) \\ &= \sum_{x=1}^{N_F} 2\pi_F(x) - \sum_{x=1}^{N_F} \pi_F(x) P_F(x,x) = 2 - \sum_{x=1}^{N_F} \pi_F(x) P_F(x,x) \end{aligned}$$

and finally:

$$K^{(2)} = 2 - \sum_{x=1}^{N_F} \pi_F(x) P_F(x,x) \tag{A.9}$$

# Appendix B

# Simulation

## B.1 CLAB: The Clustering Laboratory

CLAB stands for CLustering LABoratory, and is an experimental system we have designed and implemented for our performance study. CLAB consists of a set of Unix tools that can be glued together to many configurations:

- `gen`, a synthetic object oriented application that implements both the Tektronix Hypermodel Benchmark [And90], and the Sun Engineering Benchmark [Cat88]. `<clab.h>`, a set of C++ classes and a corresponding library, that facilitates instrumentation of any object oriented application, to extract its object graph and its access traces. `gen` produces an `object-graph` and a `object-trace` object.

- `smooth`, an access model generator. Given an input trace and an optional object graph, `smooth` performs statistical processing of the trace. It can generate a graph or hyper-graph representation of any of the following access models: IID,SMC,HMC,SG, or PSM (see Chapter 2 for the model definitions).

- `cl`, a tool to cluster objects to pages using an appropriate `access-model`. Currently `cl` implements PRP, DFS, BFS, WDFS, CACTIS, WISC, PIPE (organ pipe), KL (Kernighan Lin), and PT (Placement trees) clustering algorithms (all defined in Chapter 4). The output is a *cluster* object (object to page mapping).

- `sim`, a general purpose cache/disk simulator and performance monitor. Currently `sim` can simulate variable size LRU, Working Set, and OPT (an optimal cache based on Belady's algorithm [Bel66]). `sim` takes as input a `trace` object and optionally a `cluster`. It generates a page fault stream (`trace` object) of a cache of specified size and policy, periodic statistics of that cache (`stats` object), or an interleaving of faults and statistics.

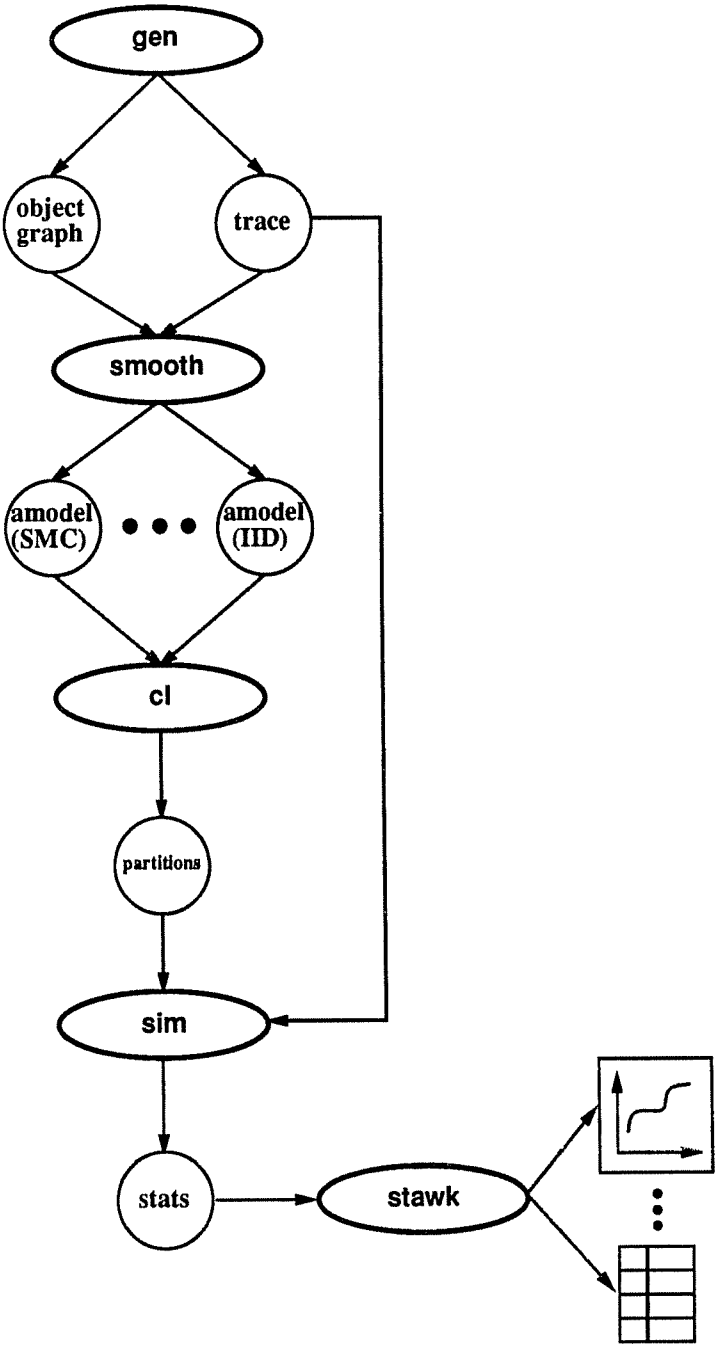



Figure B.1: A typical configuration of CLAB

*CLAB in a typical configuration to test the performance of a client cache using mappings generated by any clustering algorithm of the study.*

- `stawk`, a tool that resembles the Unix tool **awk**, providing a relational like interface to all objects produced by the previous tools. **stawk** supports frequently needed statistical processing functions (like averaging and standard deviation). Typically it is used for manipulating raw performance statistics (i.e., `stat` objects), and for generating graphs that can be displayed by standard Unix packages like **xgraph**, or **gnuplot**.

There are a few more auxiliary tools like **trmerge**, to merge and mix object traces, and `trmap` to transform and change single event traces.

All tools were written in C++ and communicate using objects. Every such object is written to (and read from a) Unix text file/pipe. We have chosen to represent objects in a textual format, to facilitate visual inspection, and to enable us use standard Unix tools to manually change or inspect them. Figure B.1 shows a typical configuration of CLAB, used for most of the experiments presented in this thesis.

## B.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.

| Level ($t$) | $N_t$ | $A_t$ | (%) | $TN_t$ | $TA_t$ | $\rho$(%) |
|---|---|---|---|---|---|---|
| 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 B.1: 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 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 $TN_t$:

$$TN_t = \sum_{h=h_0}^{t} N_h = \frac{p^t - 1}{p - 1}$$

$$TA_t = \sum_{h=h_0}^{t} A_h$$

$$\rho = \frac{TA_t}{TN_t}$$

where $\rho$ gives the ratio of accessible objects to the total objects.

Table B.1 shows $\rho$ 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.

## B.3 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 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, possibly with some objects appearing multiple times in a traversal. 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 $a/2$. 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 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 maybe 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 B.2 illustrates the effect, by marking the probabilities of objects that are accessed by a single mn2 traversal. As can be seen, 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; in addition, suppose that the training trace consists of a long sequence of randomly selected traversals; finally, suppose 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 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 TBOG with the given parameters), the

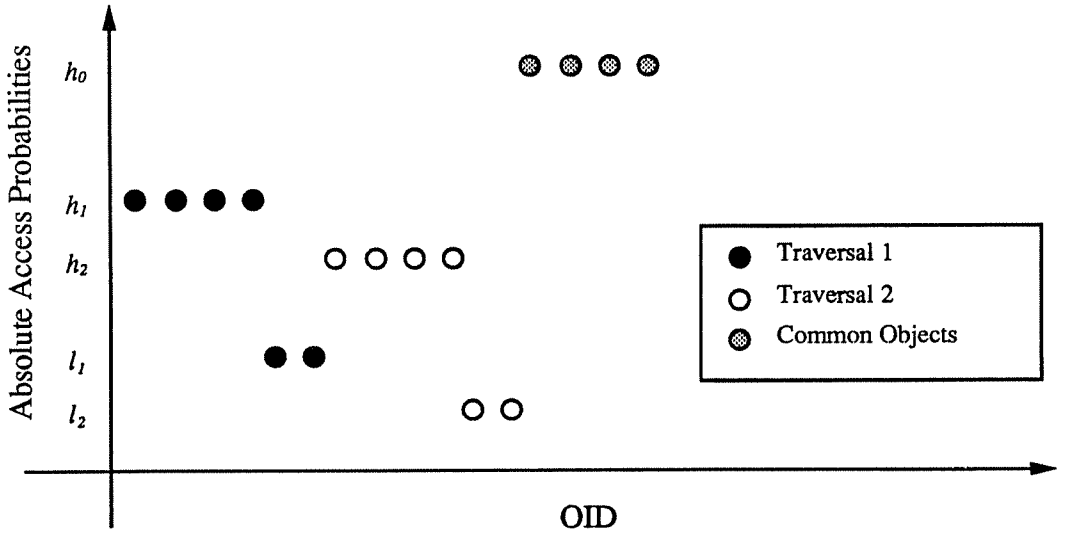



Figure B.2: Object Probabilities on mn2

*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.*

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.

# Bibliography


[ADM+90] Malcolm Atkinson, David DeWitt, David Maier, Francois Bancilhon, Kalus Dittrich, and Stanley Zdonik. The Object–Oriented Database System Manifesto. Technical report, University of Wisconsin, Comp. Sciences Dept., 1990.

[ADU71] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of Optimal Page Replacement. *Journal of ACM*, 18(1):80–93, January 1971.

[AG91] Rakesh Agrawal and N. H. Gehani. Rationale for the Design of Persistence and Query Processing in Database Programming Language O++. Technical report, AT&T Bell Laboratories, 1991.

[Aga88] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Kluwer Academic Publishers, 1988.

[All78] Arnold O. Allen. *Probability, Statistics, and Queueing Theory, with Computer Science Applications.* Academic Press, 1978.

[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.

[Bar84] Earl R. Barnes. Partitioning the Nodes of a Graph. In *Proceedings of the Fifth Quadrennial International Conference on the Theory of Graphs with special emphasis on Algorithms for Computer Science Applications*, pages 57–72, Kalamazoo,Michigan, June 1984.

[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.

[Bel66] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[BKKG88] Jay Banerjee, Won Kim, Sung-Jo Kim, and Jorge F. Garza. Clustering a DAG for CAD databases. *IEEE Transactions on Software Engineering*, 14(11), 11 1988.

[BS89] Bruno Braban and Peter Schlenk. A Well Structured Parallel File System for PM. *Operating Systems Review*, 23(2):25–38, 1989.

[Bud90] Timothy A. Budd. *An introduction to Object Oriented Programming.* Manuscript, May 1990.

[BVJ84] Earl R. Barnes, A. Vannelli, and J.Q.Walker. A New Procedure for Partitioning the Nodes of a Graph. Technical Report RC 10561, IBM Thomas J. Watson Research Center, June 1984.

[CAC+84] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Managment System. *Software Practice and Experience*, 1984.

[Cat88] R. G. G. Cattell. Object Oriented Performance Measurement. In *Proceedings of the 2nd International Workshop on OODBMS*, pages 364–367, FRG, September 1988.

[Cat91] R. G. G. Cattell. *Object Data Management: Object-oriented and Extended Relational Database Systems.* Addison-Wesley Publishing Co., Reading, Mas., 1991.

[CD73] E. G. Coffman and P. J. Denning. *Operating Systems Theory.* Prentice–Hall Inc, Englewood Cliffs, NJ, 1973.

[CDKK85] H-T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug. The wisconsin storage system software. *Software—Practice and Experience*, 15(10):943–962, October 1985.

[CDRS89] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. The EXODUS Storage Manager. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* Addison-Wesley Publishing Co., 1989.

[CFL+91] M. Carey, M. Franklin, M. Livny, , and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Denver, CO, May 1991.

[CH81] R. W. Carr and J. L. Hennessy. WSCLOCK–A Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the 8-th SOSP, Operating Systems Review*, volume 15, pages 87–95, December 1981.

[CK89] Ellis E. Chang and Randy H. Katz. Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object Oriented DBMS. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 348–357, Atlantic City, NJ, May 1989.

[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.

[DFMV90] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Velez. A study of Three Alternative Workstation Server Architectures for Object Oriented Database Systems. Technical Report 936, University of Wisconsin, Computer Sciences Dept., May 1990.

[DG92] David DeWitt and Jim Gray. Parallel Database Systems: The future of high performance Database Systems. *Communications of ACM*, 35(6):85–98, 1992.

[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.

[Fer74] Domenico Ferrari. Improving Locality by Critical Working Sets. *Communications of the ACM*, 17(11):614–620, Nov 1974.

[Fer75] Domenico Ferrari. Tailoring Programs to models of program behavior. *IBM Systems Journal*, 19(3):244–251, May 1975.

[FST88] Shel Finkelstein, Mario Schkolnick, and P. Tiberio. Physical Database Design for Relational Databases. *ACM Transactions on Data Base Systems*, 13(1):91–128, March 1988.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W.H.Freeman and Company, 1979.

[Gol81] A. Goldberg. Introducing Smalltalk-80 system. *BYTE magazine*, August 1981.

[GS73] David D. Grossman and Harvey F. Silverman. Placement of Records on a Secondary Storage Device to Minimize Access Time. *JACM*, 20(3):429–438, July 1973.

[HBD91] Gilbert Harrus, Veronique Benzaken, and Claude Delobel. Measuring Performance of Clustering Strategies: the CluB-0 Benchmark. Technical Report 66-91, Altair, January 1991.

[HG71] D. J. Hatfield and J. Gerald. Program Restructuring for Virtual Memory. *IBM Systems Journal*, 10(3):168–192, May 1971.

[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.

[HN76] M. Hammer and B. Niamir. Index Selection in a Self Adaptive Database. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1–8, May 1976.

[Hol90] Mark A. Holliday. A Program Behavior Model and its Evaluation. Technical Report CS-1990-9, Dept. of Computer Science, Duke University, 1990.

[HZ87] M. F. Hornick and S. B. Zdonick. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1), 1987.

[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.

[Men91] Carlos G. Mendioroz. Graph Clustering and Caching. Master's thesis, Computer Systems Research Institute– University of Toronto, Toronto, CANADA, 1991.

[MJLF84] M.McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[Mos89] J. Eliot B. Moss. Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 269–285, Gleneden Beach, Oregon, June 1989.

[Mos90] J. Elliot B. Moss. Working with objects: To Swizzle or Not to Swizzle. Technical report, University of Massachusetts, Amherst, Computer and Information Science, 1990.

[MS88] J. Eliot B. Moss and Steven Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. In *Lecture Notes in Computer Science: Advances in Object-Oriented Database Systems*, volume 334, pages 298–316. Springer-Verlag, September 1988.

[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.

[RC88a] E. Richardson and M. J. Carey. E Language: Issues and Implementation. Technical Report 791, University of Wisconsin-Madison, CS Department, March 1988.

[RC88b] J. Richardson and M. Carey. Programming Constructs for Database System Implementation in EXODUS. In *Proc. of the ACM SIGMOD Int'l. Conf*, San Francisco, CA, May 1988.

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

[Riv76] R. Rivest. On Self Organizing Sequential Search Heuristics. *ACM Communications*, pages 63–67, 1976.

[Sch77] Mario Schkolnick. A Clustering Algorithm for Hierarchical Structures. *ACM Transactions on Data Base Systems*, 2(1):27–44, March 1977.

[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.

[Sto87] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Pub. Company, 1987.

[SWT89] H. S. Stone, J. L. Wolf, and J. Turek. Optimal Partitioning of Cache Memory. Technical Report RC14444 (64697), IBM, March 1989.

[SWZ92] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Automatic tuning of data placement and load balancing in disk arrays. In *Database Systems for Next Generation Applications - Principles and Practice, Advanced Database research and development series.* World Scientific Publications, 1992.

[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.

[TN92] Manolis M. Tsangaris and Jeffrey F. Naughton. On the performance of object clustering techniques. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 144–153, San Diego, California, June 1992.

[VC90] Paul Vongsathorn and Scott D. Carson. A System for Adaptive Disk Rearrangement. *Software Practice and Experience*, 20(3):225–242, March 1990.

[Won80] C.K. Wong. Minimizing expected head movement in one-dimensional mass storage systems. *Computer Surveys*, 12(2), June 1980.

[Won83] C.K. Wong. *Algorithmic Studies in Mass Storage Systems.* Computer Science Press, 1983.

[YLT83] C. T. Yu, M. K. Lam, and F. Tai. Adaptive clustering schemes: A general framework. *COMPAC*, pages 81–89, November 1983.

[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.