

Object Placement in Parallel Object Oriented Databases

Zhen He

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Science (Honours) at
The Department of Computer Science
Australian National University

December 1998

© Zhen He

Typeset in Palatino by T_EX and L^AT_EX 2_ε.

Except where otherwise indicated, this thesis is my own original work.

Zhen He
4 August 2003

To my understanding parents.

Acknowledgements

Firstly I would like to take this opportunity to thank my two advisors Jeffrey and Steve. Both of whom, have spent enormous amounts of time helping me with my project. I am very lucky to have two advisors who have brought two different areas of computer sciences to this thesis. Jeffrey with his vast experience in object oriented databases and Steve with his in depth knowledge in scalable object storage. With the absence of either advisor this thesis would have been much poorer in quality.

I would like to thank my parents for their understanding throughout the year.

I would like to thank the other members of the UPSIDE team, Chris, Alonso, David, Gavin and John for the useful and insightful feedback throughout the year.

I would like to thank David Sitsky, David Walsh and Andrew Tridgell for their help in the implementation of the test-bed used in this project.

I wish also to thank ACSys for the financial support. In particular I would like to thank Jan for her help through out the year.

On the other spectrum of people to thank, I would like to thank the members of the zoo, Luke, Peter, Gareth, Tue and Reehaz. All of whom will be forever remembered for reasons best left unsaid. Luke deserves a special mention, for helping with many aspects of my project, from providing helpful and useful comments to helping in the implementation of the test-bed.

Abstract

This thesis explores the issue of object placement in the context of parallel object oriented databases. The particular parallel hardware architecture targeted is the shared nothing architecture where each node has its own processor, memory and disk. The particular query parallelisation method employed is intra-operator parallelism where each node works on the portion of the data set which resides on its own local disk. In such an environment the object placement algorithm must balance two conflicting objectives. The first is to minimise the number of internode object traversals and the second is to maximise the degree of computational parallelism.

This thesis proposes a new similarity-based two phased approach to object placement for the particular situation outlined above. The first phase is an adaptation of existing similarity based declustering algorithms developed for uniprocessor multiple disk systems. The objective of the first phase is to balance the two conflicting objectives outlined above. The second phase is an optimisation step in which objects are clustered into pages for optimal use of page grain caching. The objective of the second phase is to minimise the number of remote page loads.

The performance of the two phased approach was investigated experimentally. The results show that the two phase approach was able to reduce the number of internode object references and remote page loads while preserving a high degree of parallelism.

Contents

| | |
|--|-----------|
| Acknowledgements | vii |
| Abstract | ix |
| 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.2 Overview of Important Concepts | 2 |
| 1.3 Approach | 2 |
| 1.4 Contribution | 3 |
| 1.5 A Theoretical Framework | 3 |
| 1.6 Organisation | 4 |
| 2 Background | 5 |
| 2.1 Parallel Architectures | 5 |
| 2.2 Parallel Query Processing Methods | 6 |
| 2.3 Parallel relational databases | 7 |
| 2.3.1 Data partitioning | 7 |
| 2.3.2 Partial declustering | 7 |
| 2.4 Clustering for uniprocessor OODBMS | 8 |
| 2.4.1 Similarity between clustering for uniprocessor OODBMS and declustering in parallel OODBMS | 9 |
| 2.4.2 Dynamic vs Static Frequency Counts | 10 |
| 2.5 Summary | 10 |
| 3 Our Approach | 11 |
| 3.1 Similarity-Based Declustering | 11 |
| 3.2 Two Phased Approach to Object Placement | 12 |
| 3.2.1 Declustering | 12 |
| 3.2.1.1 Definition of Declustering Similarity, S_d | 12 |
| 3.2.1.2 Declustering Heuristic and Determination of Declus- tering Similarity | 13 |
| 3.2.2 Clustering | 15 |
| 3.2.2.1 Definition of Clustering Similarity, S_c | 16 |
| 3.2.2.2 Clustering Algorithm | 16 |
| 3.2.3 A Point of Failure | 18 |
| 3.2.3.1 Improved Declustering Heuristic | 19 |
| 3.3 Summary | 20 |

| | |
|---|-----------|
| 4 Experiments | 21 |
| 4.1 Experimental Methodology | 21 |
| 4.1.1 Test-bed | 21 |
| 4.1.1.1 Properties of AP1000 | 21 |
| 4.1.1.2 Approach to Delivering Appropriate Test-bed Using AP1000 | 22 |
| 4.1.2 Query Processing | 23 |
| 4.1.3 Startup and Termination | 26 |
| 4.1.4 Cold Times | 26 |
| 4.2 Measures of object placement quality | 26 |
| 4.3 Response Time vs. Average Time | 28 |
| 4.4 Scalability Experiment | 28 |
| 4.4.1 Individual Queries | 29 |
| 4.4.1.1 Query 1 | 30 |
| 4.4.1.2 Query 2 | 31 |
| 4.4.1.3 Query 3 | 32 |
| 4.4.1.4 Query 4 | 34 |
| 4.4.2 Server workload imbalance | 36 |
| 4.4.3 Scalability Comparison of Individual Queries | 37 |
| 4.4.4 Overall Results for all four queries | 39 |
| 4.4.4.1 Running Time | 39 |
| 4.4.4.2 Number of Remote Page Loads | 40 |
| 4.4.4.3 Degree of Client Workload Imbalance | 40 |
| 4.4.4.4 Scalability Experiment: Summary | 40 |
| 4.5 Varying Training Conditions Experiment | 42 |
| 4.5.1 Response Time | 42 |
| 4.5.2 Communication costs | 44 |
| 4.5.3 Degree of Parallelism | 45 |
| 4.5.4 Varying Training Conditions Experiment: Summary | 46 |
| 4.6 Summary | 46 |
| 5 Parsets | 47 |
| 5.1 Background | 47 |
| 5.2 A case for partial declustering | 48 |
| 5.3 Automatic placement of parsets | 48 |
| 5.4 Determination of Degree of Declustering | 50 |
| 5.5 Summary | 53 |
| 6 Conclusion | 55 |
| 6.1 Summary of Findings | 55 |
| 6.2 Further Work | 56 |
| A Peculiar Result Explanation Continued | 57 |
| B Scalability Comparison via Response Time and Average Time | 59 |

| | |
|--|-----------|
| C Varying Training Conditions Results Continued | 61 |
| D OO7 schema | 63 |
| Bibliography | 67 |

Introduction

This thesis explores the issue of object placement in the context of parallel object oriented databases (OODBMS). There are many parallel architectures on which parallel OODBMS are run. The particular parallel architecture explored in this thesis is the 'shared nothing' architecture where each node has its own processor, memory and disk. Communication is performed via explicit message passing. There also are many different ways queries can be parallelised in a parallel OODBMS. The particular method explored in this thesis is termed 'intra-operator' parallelism. In intra-operator parallelism each processor works on the portion of the data set which resides on its own local disk. The object placement strategy in such a situation should attempt to minimise the number of remote page loads and maximise the degree of parallelism.

We have adopted an *offline* approach to the placement of objects. In offline object placement the placement is determined offline with the use of dynamic or static frequency counts. Dynamic frequency counts record the frequency with which an object references another and static reference count records the presence or absence of object references between two objects. The placement determined by the object placement algorithm is achieved by rearranging the objects offline. This is in contrast to the *online* approach to object placement, in which object placement is determined when the database is in operation and objects are rearranged dynamically.

The focus of this thesis is to develop new object placement strategies for the particular situation outlined above. In addition the performance of the various object placement strategies developed in this thesis is investigated with different query access patterns.

1.1 Motivation

OODBMS provide rich facilities for the modelling and processing of structural as well as behavioural semantics associated with complex objects in many applications [Zdonik and Maier 1990]. Some of these applications include Computer-Aided Design (CAD), office information systems, spatial information systems, etc. In most cases query facilities which support retrieval of objects over large data sets are required. Research into parallel object-oriented database systems demonstrates that parallelism is a highly effective tool for providing high performance systems in large object oriented

data bases [Thakore and Su 1994; DeWitt et al. 1996; Imasaki et al. 1997]. Similar research into parallel relational database systems (RDBMS) has also proved successful in extracting performance [Ghandeharizadeh 1990; Mehta and DeWitt 1995; Copeland et al. 1988].

Effective data placement is crucial to the performance of any parallel database system since it is an important lever for load balancing [Ghandeharizadeh 1990; Padmanadhan and Baru 92; Mehta and DeWitt 1995; Copeland et al. 1988].

1.2 Overview of Important Concepts

Object placement in parallel OODBMS consists of two important concepts: *declustering* and *clustering*. Declustering in the context of parallel OODBMS corresponds to the partitioning of the graph representing the object base across the disks of a parallel computer. The way the graph is partitioned depends on the parallel hardware architecture used. For a uniprocessor, multiple disk hardware architecture, the aim of declustering is to load objects involved in the same query in parallel.

However for a shared nothing architecture, where each node has its own processor, disk and memory, the objective is to:

- a) reduce the number of internode object references
- b) maximise parallelism.

These objectives are in conflict because a) argues for assigning objects to a few nodes in order to group the relevant objects together, while b) advocates the distribution of the objects across as many nodes as possible.

The degree of declustering is a measure of the number of nodes the object base is declustered across.

Clustering in the context of parallel OODBMS is the grouping together in pages of objects that are required by remote nodes. This process reduces the number of remote page loads incurred during parallel query processing.

1.3 Approach

This thesis explores a new two phased similarity graph partitioning approach to object placement in a shared nothing system, using intra-operator parallelism.

The first phase is the *declustering* phase which attempts to reduce internode object references while attempting to maximise parallelism. This is accomplished by first assigning objects to nodes either in order of decreasing frequency counts or in order of reachability. The decision as to which node an object o_i is assigned to is determined as follows: object o_i is assigned to the node N_j which has the largest measure of similarity between the objects in N_j and o_i . We have chosen a similarity measure which states two objects are more similar to each other when they are accessed in a *navigational* manner but less similar when they can be accessed in a *parallel* manner.

The second phase is the *clustering* phase, which attempts to cluster objects together in order to reduce the number of remote page loads. This is accomplished by grouping together objects which are often referenced by a remote node together. Therefore in this phase objects are not moved from one node to another but rather are grouped into pages within the node which was assigned to it by the declustering phase. Again we use the similarity concept to determine which node an object should be clustered for. However the measure of similarity is redefined for this phase by dropping the requirement to place apart objects that can be accessed in a *parallel* manner. This is intuitively explained by the fact grouping objects in pages within a node does not effect the degree of parallelism to which objects can be accessed by a query.

1.4 Contribution

The key contribution of this thesis is the application of the similarity approach to object placement in a shared nothing OODBMS, using intra-operator parallelism which has previously not been tried. Two phases were identified to be necessary when applying the similarity approach to the shared nothing systems. The first phase termed the *declustering* phase is an adaptation of the similarity-based declustering approach applied by Liu and Shekhar [1996] to the problem of data placement in uniprocessor, multiple disk hardware architectures. The second phase termed the *clustering* phase was identified by the work done in this thesis to be a necessary second step in reducing remote page loads. The combination of the two phases to the placement of objects in a shared nothing parallel OODBMS is introduced for the first time in this thesis.

1.5 A Theoretical Framework

We consider an object base as a directed graph. Each node represents an object and an edge (o_i, o_j) represents a reference from o_i to o_j . The weights of edges can be assigned either by *dynamic* reference counts or *static* reference counts. Dynamic reference counts are determined by the frequency of traversal of references of the object base. Static reference counts assign a weight to the edge joining an object with another object a value of 1 if a reference exists otherwise it assigns a value of 0 [Tsangaris and Naughton 1991]. This makes object placement in an object oriented database a graph partitioning problem. For this study we use dynamic reference counts to represent weights on the edges of the graph.

Consider a shared nothing architecture with N nodes, and a set of queries Q_1, Q_2, \dots, Q_m . Also assume the queries are processed via intra-operator parallelism. We seek an object placement which will make $\sum_{i=1}^m Cost(Q_i)$ minimum where $Cost(Q_i)$ is the cost to execute query Q_i on the N node shared-nothing architecture.

The main issue here is how to balance the two conflicting goals as mentioned in Ghandeharizadeh, Wilhite, Lin, and Zhao [1994].

- The object placement strategy should assign all the relevant traversals by a query to a single node in order to minimise the number of internode object references

- It should distribute the workload of an application evenly across the nodes in order to maximise the processing capability of the system.

1.6 Organisation

The remainder of this thesis is organised as follows:

Chapter 2 provides a background of the related concepts and work done in both data placement in parallel RDBMS and clustering in uniprocessor OODBMS. The ideas present in this chapter were instrumental in the design of the two phased approach to object placement.

Chapter 3 describes an existing similarity approach to data declustering in an uniprocessor multiple disk environment. Then the chapter presents a new similarity-based two phased approach to object placement in a shared nothing parallel OODBMS.

Chapter 4 presents the results of running two experiments. The first was designed to investigate the scalability of the various object placement methods developed and the second investigates the ability for the various placement methods to adapt to non-favourable training conditions.

Chapter 5 introduces the concept of 'parsets', a way of parallelising set operations. Some issues relating to object placement in parsets are discussed and a new way of determining the optimal degree of declustering is described.

Chapter 6 concludes this thesis by summarising the findings made during this project and also outlines directions for further work.

Background

The previous chapter identified the particular parallel architecture and parallel query execution method investigated in this thesis. In addition our approach to object placement in parallel OODBMS was also explained. However the broader context from which these ideas originated was not presented.

This chapter aims to place the concepts and ideas identified in the previous chapter in context.

The different parallel architectures and parallel query execution methods available are outlined. In addition this chapter outlines how ideas from uniprocessor OODBMS and parallel RDBMS relate to object placement in parallel OODBMS.

2.1 Parallel Architectures

The objective of object placement algorithms differs vastly depending on the particular parallel architecture used. There are three main architectures for building a parallel object oriented database. They comprise of *shared memory*, *shared disk* and *shared nothing* systems.

Shared memory computers comprise of computers which share both memory and disks across all processors. They do not require message passing and are relatively easy to load balance. However adding additional processor, memory and disk may cause increased contention of memory and disk usage.

Shared disk computers consists of multiple nodes, each with own processor and memory. All nodes share a parallel file system on which files are fragmented across multiple disks. Files can be loaded in parallel. Object placement in the context of shared disk systems require objects which are likely to be accessed together to be placed on distinct node. Objects placed in this way are more likely to be loaded in parallel. The motivation for using a parallel file system is to reduce IO costs by allowing nodes to read and write fragments of a file in parallel across the shared disks. However when many processors attempt to read or write concurrently there is potential for disk contention. In general a parallel file system creates an opaque layer between the OODBMS and the storage device employed. Therefore an OODBMS built on top of a parallel file system can not rearrange the placement of objects in order to reduce disk contention. However in some shared disk systems the parallel file system can

be bypassed by writing to particular locations of the file on which the object base is stored. Once this is accomplished the shared disk system can be treated as a shared nothing system. This is in fact the approach taken in the experiments conducted in this thesis.

Shared nothing computers consists of multiple nodes, each with own processor, memory and disks. Communication is achieved via message passing across a common interconnection network. Object placement in a shared nothing system attempt to minimise remote page loads and maximise parallelism by placing objects which are accessed together in a *navigational* manner together and placing apart objects that can be accessed in a *parallel* manner. Many existing parallel relational databases are built on shared nothing systems. They include DBC/1012, Gamma, Bubba and Tandem. The shared nothing architecture may be viewed as being more scalable when compared to the other parallel architectures due to the transparency it provides with respect to placement. Therefore as the number of nodes increases, disk contention can be kept low by intelligent placement of objects on the private disks of the shared nothing nodes. The shared nothing architecture was chosen as the parallel architecture investigated in this thesis due to its popularity in the parallel database world and its scalable nature.

2.2 Parallel Query Processing Methods

There are three primary methods of processing queries in parallel on shared nothing systems: inter-query, inter-operator, and intra-operator parallelism.

inter-query parallelism Multiple queries are executed concurrently.

inter-operator parallelism The operators¹ involved in one query are processed in a pipelined manner whereby the results of one operator form part of the input to another. Load balancing is difficult, if not impossible, using this method as each processor is loaded proportionally to the amount of data it has to process. This amount cannot be chosen by the implementor or the query optimiser, and cannot be predicted very well [Goetz 1993].

intra-operator parallelism In intra-operator parallelism, one operator is processed across all the nodes of the parallel system. Typically, each node works on its own disjoint subset of the problem and its own data set. Parallelism is dependent on the way data items are positioned.

We have chosen intra-operator parallelism as the method used to process queries in this thesis since it is the most sensitive to object placement.

¹Operations involved in processing sets of objects or tuples of records (eg. join, select, etc.)

2.3 Parallel relational databases

A large amount of work has been done on data placement for parallel *RDBMS*. Two important problems identified from this previous work include data partitioning and partial decluster. This section outlines previously developed solutions to these two problems.

2.3.1 Data partitioning

Data partitioning in shared nothing parallel relational databases is dominated by three main methods: *round robin*, *hash* and *range* partitioning.

Round robin partitioning places tuples of a relation across the nodes of a shared nothing parallel *RDBMS* in a round robin fashion. This partitioning approach is ideal for a sequential scan of an entire relation, since it balances the number of tuples each node considers in a most optimal manner. For this reason round robin partitioning is also sometimes used to place objects in a parallel *OODBMS*. However, round robin object placement should only be used in a parallel *OODBMS* context when the queries accessing the objects partitioned contain no navigational component. The reason being that round robin partitioning, like all data partitioning methods used in parallel *RDBMS*, do not consider object references.

Hash partitioning for *RDBMS* involves assigning tuples of a relation to nodes based on the hash value of a particular attribute chosen by the user. This method of data partitioning is ideal for exact match queries where the join attribute is the hash attribute. However, the hash function may not place data items evenly across the nodes of the shared nothing system. Therefore, a full table scan on a relation which has been partitioned via hash partitioning may result in low performance due to the low degree of parallelism.

Range partitioning in parallel *RDBMS* involves assigning tuples of a relation to nodes based on the value of an attribute falling within a range of values assigned to each node of the parallel system. This data partitioning method allows the execution of range queries to be localised to a subset of nodes of the parallel system. The limitation of this method is that the range attribute must be pre-determined by the user.

All the methods described above maybe applied to *OODBMS* as was indicated by DeWitt, Naughton, Shafer, and Venkataraman [1996] on their 'parset' paper. However all the above partitioning strategies are only ideal for particular types of queries run and thus require a high level of user interaction. Further, the above partition strategies when applied to parallel *OODBMS* suffer from a lack of consideration of internode object references.

2.3.2 Partial declustering

Much work in data placement in parallel relational databases has centered around the problem of partial declustering. Partial declustering in shared nothing parallel relational databases consists of two steps: determining the degree of declustering;

and determining the subset of nodes to which fragments of the declustered relation is to be assigned to.

The degree of declustering is a measure of the number of nodes the object base is to be declustered across. The startup and termination costs of parallel query processing increases as the degree of declustering increases. However, as the degree of declustering increases, the computation time of each processor decreases due to a higher degree of parallelism. Therefore, to extract optimal performance, the degree of declustering must be chosen so that the benefits derived from the higher degree of parallelism outweighs the penalty incurred from the startup and termination cost of running the parallel query.

In this thesis we outline a method of partial declustering for parallel OODBMS which borrows ideas from partial declustering of parallel RDBMS presented in this section.

Ghandeharizadeh [1990] developed a partitioning strategy for parallel relational databases called multidimensional partitioning strategy (MDPS). He concluded that the execution of queries with minimal resource requirements should be localised to a few processors in order to avoid the communication overhead associated with parallelism. In addition he concluded that for queries with a greater demand for resources, a higher degree of parallelism is required. It was to accomplish this balance between localisation and parallelism he proposed MDPS.

MDPS partitioned the relation based on the requirements of queries which constitute the workload and the processing capability of the system. Using this information he construct a grid directory.

More precisely the grid directory construction requires the following input parameters:

1. the cardinality of each fragment.
2. the splitting strategy.
3. the relation to be declustered.
4. the partitioning attributes.

Given these inputs, the algorithm constructs a grid directory which records the processor to which each fragment of the relation is assigned. A query optimiser can potentially use this information to localise the execution of a query to only the processors which contain the desired tuples of a relation. MDPS was demonstrated to be superior when compared to range and hash partitioning strategies. It is a variation on MDPS that we proposed as a new way of determining the optimal degree of declustering in a parallel OODBMS.

2.4 Clustering for uniprocessor OODBMS

A considerable amount of work has been done on clustering of objects into pages in uniprocessor OODBMS. The algorithms proposed in this thesis have their roots in

uniprocessor clustering algorithms.

2.4.1 Similarity between clustering for uniprocessor OODBMS and declustering in parallel OODBMS

Clustering for uniprocessor OODBMS attempts to place objects onto pages such that for a given application the number of page faults becomes minimal. Uniprocessor clustering algorithms are divided into two categories — sequence based and partition based — both of which treat the object base as a clustering graph (CG) [Gerlhof et al. 1993].

The vertices and edges of the graph are labelled with weights. Vertex weights represent object size and edge weights represent frequency of traversal of one object to another (dynamic object reference counts) or presence of reference between objects (static reference counts).

The first category of uniprocessor clustering algorithm is sequence based mapping algorithms which first sorts the objects in the CG to some criteria (eg. order by decreasing reference count). Once sorted, the CG is traversed according to a traversal algorithm which produces a sequence that is used to order the storage of objects onto pages.

The second type of uniprocessor algorithm is partition based clustering algorithms which considers the object placement problem as a graph partitioning problem in which the min-cut criteria is to be satisfied for page boundaries. This second method is analogous to the approach proposed in this thesis to decluster the object base for a parallel OODBMS. For declustering in parallel OODBMS the object base is also considered as a graph partitioning problem but the min-cut criteria is to be satisfied between *node* boundaries instead of *page* boundaries. In the uniprocessor case, satisfying the min-cut criteria results in a reduction in the number of page faults, which is analogous to the parallel case in which a reduction in the number of remote page loads results.

There are two types of partition based algorithms for clustering in uniprocessor OODBMS: *iterative improvement* and *constructive partitioning*. Iterative improvement algorithms such as the Kernighan-Lin Heuristic (KL) [Kernighan and Lin 1970], iteratively improve partitions by swapping objects between partitions attempting to satisfy the min-cut criteria. Constructive algorithms such as greedy graph partitioning (GGP) [Gerlhof et al. 1993] attempt to satisfy the min-cut criteria by first assigning only one object to a partition and then combine partitions in a greedy manner.

In a study done by Gerlhof, Kemper, Kilger, and Moerkotte [1993] a number of existing uniprocessor clustering algorithms were compared and a new greedy graph partitioning algorithm with look ahead (GGPl_a) was proposed. They also conclude that constructive partitioning algorithms were considerably faster than iterative improvement algorithms since they ran in linear time and the iterative improvement algorithms involved a large amount of useless swaps which had to be discarded. The constructive algorithms only produced clustering results which were of slightly poorer quality when compared to their iterative counterparts. Therefore they recommended that for large databases the greedy constructive approach should be taken for

clustering the object base is a uniprocessor OODBMS. This supports our choice of a greedy algorithm for the purpose of declustering a parallel OODBMS, given the analogies outlined previously between clustering for uniprocessor OODBMS and declustering for parallel OODBMS.

2.4.2 Dynamic vs Static Frequency Counts

The weights assigned to the edges of the clustering graph (CG) as stated in the previous section can either be dynamic reference counts or static reference counts. Dynamic reference counts contain historical information on query access patterns which can be used to predict the likely pattern in which the object base will be traversed. Static reference counts, on the other hand, only contain information on the structure of the object references present in the object base and hence does not contain information of query access patterns.

This thesis uses dynamic frequency counts to assign weights to the CG graph².

2.5 Summary

Methods developed for data partitioning in parallel RDBMS can be applied to place objects in a parallel OODBMS. However they are desirable only for particular types of queries and thus require a high level of user interaction. In addition methods developed for RDBMS when used to place objects involved in a query where navigation is present suffer from a lack of consideration of internode object references.

The application of the graph partitioning approach to place objects in a parallel OODBMS can be seen to be analogous to the application of the same approach to cluster objects of a uniprocessor OODBMS.

²In this thesis dynamic reference weights are used to determine object placement offline

Our Approach

The background material presented in the previous chapter placed into context the particular problem this thesis is attempting to solve. However, the similarity approach upon which we based our object placement algorithms were omitted.

In this chapter we address this omission by first presenting the way in which the similarity based approach has previously been used to solve the problem of data declustering in a uniprocessor, multiple disk environment. We then present a new similarity based two phased approach for solving the different problem of object placement in a shared nothing parallel OODBMS using intra-operator parallelism.

3.1 Similarity-Based Declustering

Fang, Lee, and Chang [1986] introduced similarity graph partitioning to decluster data across multiple disks of a uniprocessor, multiple disk system.

Fang, Lee, and Chang [1986] considered two groups of data items G_i and G_j similar if, for every point p_k in G_i there exists a point p_l in G_j in which p_k is a nearest neighbour of p_l , or p_l is a nearest neighbour of p_k . Minimal spanning trees and shortest spanning paths were proposed to divide a set of objects into two 'similar groups'.

Liu and Shekhar [1996] proposed a variation of this approach in which query frequency distribution information was used to define similarity. Again they were attempting to solve the problem of declustering in a uniprocessor, multiple disk environment. They proposed a general approach to declustering which they applied to grid files of a spatial database. Their aim was to maximise the chances that a pair of atomic data-items that are frequently accessed together by queries are allocated to distinct disks, they termed this the max-cut criteria. They defined similarity between two items as the weight of the edge joining two data items. The weight is assigned as the likelihood that the pair of data-items will be accessed together by queries in the query set of interest.

Liu and Shekhar [1996] proposed two heuristic techniques to perform the similarity based declustering. The first is a *incremental* technique which is faster than the second *global max-cut graph* partitioning technique.

The incremental method selects an unallocated data item and then retrieves all the data items that are in a window around the selected data item. The window around

a data-item o_u is a set of data items which are likely to be accessed together with the data-item o_u . The algorithm iterates through the unallocated objects in the window in a greedy manner placing the data-items into disks based on the max-cut criteria.

The global max-cut graph partitioning technique initially partitions the data-items by some less optimal method like the incremental method and then repeatedly compares and swaps pairs of data items for possible improvements. This is analogous to iterative improvement algorithms used in uniprocessor clustering as described in Section 2.4.1.

Liu and Shekhar [1996] concluded that the first method provides the best trade-off between the parallel response time for queries and the cost of declustering. This supports the conclusion given by Gerlhof, Kemper, Kilger, and Moerkotte [1993] for clustering in uniprocessor OODBMS (see Section 2.4.1). It is a variation on the first method that we have adopted to decluster our database. However, instead of declustering with the aim of parallelising IO, this chapter explores declustering with the aim of minimising internode object references and maximising computational parallelism.

3.2 Two Phased Approach to Object Placement

The object placement algorithm outlined in this chapter consists of two phases. The first phase is a declustering phase in which a greedy similarity graph partitioning algorithm is used to assign objects into nodes of the shared-nothing system. The aim of the first phase is to minimise internode traversals and maximise parallelism. Once assigned to nodes, the second phase attempts to cluster objects residing at each node into pages with the aim of reducing the occurrence of remote page requests. Therefore, objects that are likely to be accessed by another remote node are clustered together. Both phases use query frequency distribution information stored in dynamic frequency counts to decide the placement of objects.

3.2.1 Declustering

We use a variation of the similarity based approach proposed by [Liu and Shekhar 1996] to *decluster* the objects in the data set. Unlike [Liu and Shekhar 1996] who considered data placement in a uniprocessor, multiple disk environment, we target a share-nothing architecture. Our algorithm differs from [Liu and Shekhar 1996] in that we attempt to place objects that have a higher degree of similarity *together* in the disk of the same node in contrast to their goal of placing similar data items on *different* disks.

3.2.1.1 Definition of Declustering Similarity, S_d

The goal of the declustering similarity equation presented in this section is to balance the two conflicting objectives: minimised number of internode object references and maximised operation parallelism as discussed in Section 1.2.

Our definition of declustering similarity states that two objects are more similar if they are accessed together in a *navigational* manner but less similar if the two objects can be accessed together in a *parallel* manner. This enables the algorithm to preserve a degree of parallelism while decreasing internode object references.

The *declustering similarity equation* between objects o_i and o_j is defined as follows:

$$S_d(o_i, o_j) = \alpha \left(w_{link}(o_i, o_j) + w_{link}(o_j, o_i) \right) - (1 - \alpha) w_p(o_i, o_j) \quad (3.1)$$

where α is a user defined weight that allows the user to balance the relative importance of reducing internode object references as compared to maximising execution parallelism. A higher value of α corresponds to placing higher importance on reducing internode object references. A possible reason for doing so is the fact that in an environment where the communication overhead is very high it may be appropriate to place higher importance on reducing internode object references. A lower α value may be appropriate for situations where most queries run contain complex computations on objects processed. In such a situation balancing computation workload may be more important when compared to reducing internode object references.

The term $w_{link}(o_i, o_j)$ is used to denote the frequency of traversal of the pointer link from o_i to o_j . $w_p(o_i, o_j)$ denotes the frequency with which the two objects o_i and o_j can be processed on different nodes in parallel. Two objects are said to be accessed in parallel when the placement of the two objects determine the parallelism of query processing. In other words, in this study, $w_p(o_i, o_j)$ will be n if both o_i and o_j belong to the same set which is accessed n times. Otherwise, $w_p(o_i, o_j)$ will be treated as zero.

The *declustering similarity equation* of an object o_i with a partition P_j of objects is defined as follows:

$$S_d(o_i, P_j) = \alpha \sum_{o_k \in P_j} \left(w_{link}(o_i, o_k) + w_{link}(o_k, o_i) \right) - (1 - \alpha) \sum_{o_k \in P_j} w_p(o_i, o_k) \quad (3.2)$$

The declustering algorithm proposed in the next section considers each object in the object base with a frequency count greater than zero and places the object in the partition, P_j , that results in the largest value for $S_d(o_i, P_j)$.

3.2.1.2 Declustering Heuristic and Determination of Declustering Similarity

The heuristic used in the declustering phase is a greedy approach. Objects are first sorted into an array based on non-increasing frequency of reference. Then each object o_i is placed, one at a time, into the partition P_j which has the largest declustering similarity value with the object o_i . In the case where a tie occurs, the object is placed in the partition to which the least number of ties have been assigned. The reason for this is that if objects involved in ties are distributed randomly or to a particular partition then there is potential for an imbalance, which may lead to a bottle neck, since nodes with abnormally large number of tied objects may receive abnormally large number of remote page requests.

A simple method of determining the declustering similarity of an object with re-

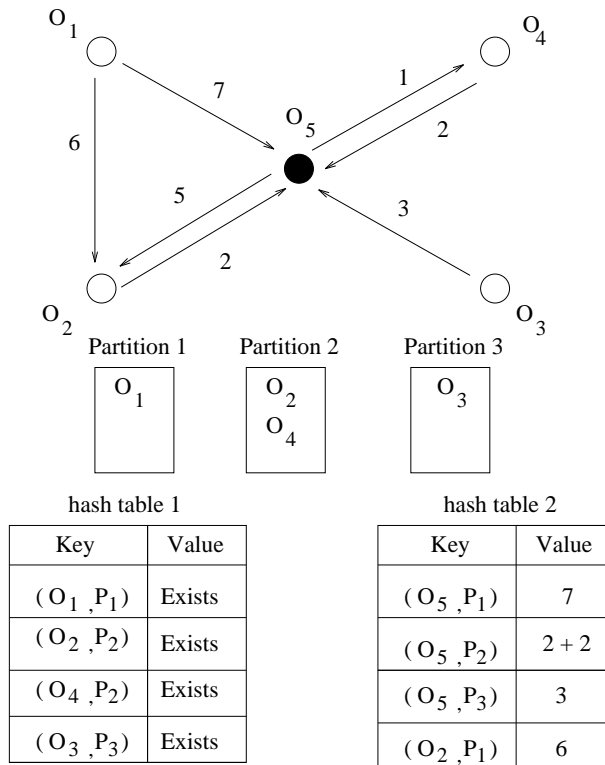


Figure 3.1: Illustration of the two hash table technique

spect to a partition is to add the declustering similarity between each object in the partition and the currently considered object. Such an approach would not be feasible for large numbers of objects. We have developed a faster method that uses two hash tables to accumulate the information required to calculate the first summation in the declustering similarity partition equation 3.2. See Figure 3.1, where object O_5 is the object that is currently being considered for placement, while the other objects have been partitioned.

Both hash tables hash object identifier and partition number pairs. The first hash table indicates whether an object exists in a partition. For each object o_i being considered the presence in partition P_j of each of the objects o_k referenced by o_i is determined by checking for the existence of an entry (o_k, P_j) in the first hash table. If the entry exists, then $S_d(o_i, P_j)$ is incremented by $w_{link}(o_i, o_k)$. The first term in the first summation of the declustering similarity equation 3.2 is thus calculated.

The second hash table contains the set of object identifiers which is being referenced by objects already in the partition. The value field in this hash table is the accumulated frequencies of all objects o_k in the partition P_j which reference the object o_i . This is the second term in the first summation of the declustering similarity equation defined in the previous section.

Algorithm 1 A declustering algorithm

```

let  $N$  be the number of nodes;
let  $P_j$  and  $o_i$  be a partition and an object;
let  $set(o_i)$  be the set that object  $o_i$  belongs to;
let  $load(P_i, S_j)$  be the load balancing factor for the node  $P_j$ 
for objects of set  $S_j$ ;
sort all objects with frequency greater than zero into non-increasing order
of frequency
for each object  $o_u$  in sorted order do
  for each partition  $P_j$ , where  $1 \leq j \leq N$  do
    for each object  $o_v$  that  $o_u$  references to do
      if  $(o_v, P_j)$  exists in hash table 1 then
         $S_d(o_u, P_j) \leftarrow S_d(o_u, P_j) + \alpha \times w_{link}(o_u, o_v)$ ;
      endif
    endfor
     $S_d(o_u, P_j) \leftarrow S_d(o_u, P_j) + \alpha \times$  value in hash table 2 for  $(o_u, P_j)$ ;
     $S_d(o_u, P_j) \leftarrow S_d(o_u, P_j) - (1 - \alpha) \times load(P_j, set(o_u))$ ;
  endifor
  if no ties occurs then
    let  $P_k$  be the partition that has  $\max_{1 \leq i \leq N} (S_d(o_u, P_i))$ ;
  else
    let  $P_k$  be the partition that has received least number of ties;
  endif
   $P_k \leftarrow P_k \cup \{o_u\}$ ;
  update hash tables;
  update load balancing factors;
endifor

```

Pseudo code for the two hash table method of determining declustering similarity is presented in Algorithm 1.

3.2.2 Clustering

The second phase termed the *clustering* phase was identified by the work done in this thesis to be a necessary second step in reducing remote page loads.

The clustering phase is composed of two steps. The first step clusters objects of the same type together. This benefits intra-operator parallelism since each node iterates through the portion of objects that reside in its own disk. Once clustered together based on type, the second step reduces remote page loads by clustering objects which are likely to be accessed by another node together.

While the implementation of the first step is straight forward, the second is not as obvious. Our approach is to partition the objects to be clustered at a particular node into N partitions where each partition is reserved for a node. Each object is assigned

to the partition which has the largest similarity value with respect to the node that the partition is clustered for. This requires a slightly different definition of similarity.

3.2.2.1 Definition of Clustering Similarity, S_c

The clustering similarity equation between objects o_i and o_j is defined as follows:

$$S_c(o_i, o_j) = w_{link}(o_i, o_j) + w_{link}(o_j, o_i) \quad (3.3)$$

The term w_{link} is defined in the same way as in Equations 3.1 and 3.2. Note the term w_p present in Equations 3.1 and 3.2 is not required in this equation since clustering does not effect parallelism in the query processing techniques used in this document.

The clustering similarity equation between object o_i and partition P_j is defined as follows:

$$S_c(o_i, P_j) = \sum_{o_k \in P_j} (w_{link}(o_i, o_k) + w_{link}(o_k, o_i)) \quad (3.4)$$

The clustering algorithm proposed in the next section use the clustering equation 3.4 to determine the partition P_j which contains the most number of objects that reference object o_i or objects which is referenced by o_i . Each partition contains a set of objects which is clustered for a particular node. Therefore partition P_j contains a set of objects which will be stored together in pages with the aim of reducing the number of remote page requests by node N_j .

3.2.2.2 Clustering Algorithm

The algorithm for clustering is given in Algorithm 2. In this step we simply reuse the information in the two hash tables to calculate clustering similarity as defined in equation 3.4 and place the objects in the partition that returns the largest similarity. Note since the clustering process follows declustering, the hash tables will have already been constructed.

Note we do not need to break ties since placing objects involved in ties together in one partition is not necessarily worse than evenly distributing ties. Tied objects occur when more than one partition contain objects that either reference it or is referenced by it. One partition may reference many tied objects in which case loosely grouping the tied objects together may be superior to evenly distributing the ties.

Once all the objects are clustered into partitions, the objects are stored into the database one partition at a time. If the similarity value of an object with respect to all other nodes is zero, then the object is placed in the partition reserved for the node that the object is on.

Algorithm 2 A clustering algorithm

```

let  $N$  be the number of nodes;
let  $n_j$  be node number  $j$ ;
let  $P_j$  be the partition for the  $j$ -th node;
for each node  $n_j$  for  $1 \leq j \leq N$  do
    for each object  $o_u \in n_j$  do
        for each partition  $P_k$  where  $P_k \neq P_j$  do
            for each object  $o_v$  that  $o_u$  references to do
                if  $(o_v, P_k)$  exists in hash table 1 then
                     $S_c(o_u, P_k) \leftarrow S_c(o_u, P_k) + w_{link}(o_u, o_v)$ 
                endif
                 $S_c(o_u, P_k) \leftarrow S_c(o_u, P_k) + \text{value of the hash table 2 for } (o_u, P_j)$ 
            endfor
        endfor
    endfor
    if  $\max_{P_k \neq P_j}(S_c(o_u, P_k)) = 0$  then
         $P_j \leftarrow P_j \cup \{o_u\}$ ;
    else
        let  $P_k$  be the partition that has  $\max_{P_k \neq P_j}(S_c(o_u, P_k))$ ;
         $P_k \leftarrow P_k \cup \{o_u\}$ ;
    endif
endfor
endfor

```

3.2.3 A Point of Failure

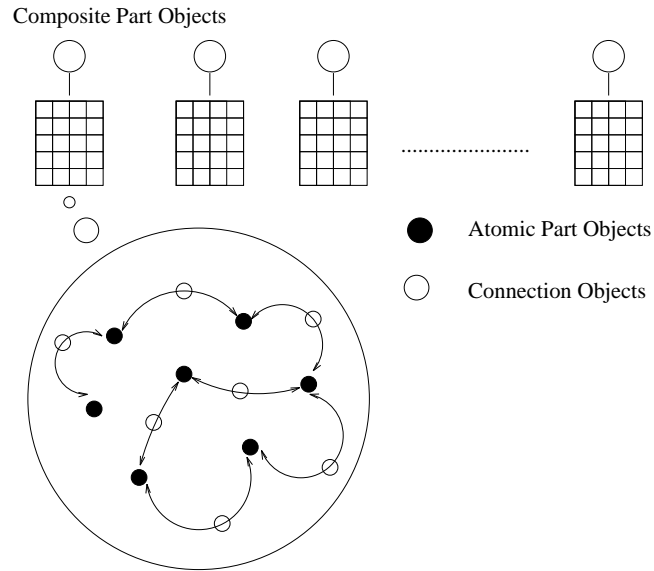


Figure 3.2: Parallelised atomic part graph traversal

The clustering and declustering phases proposed in Section 3.2.1 and 3.2.2 were found to fail for a query involving traversal of a complex graph of objects.

The query was run on the objects of the medium size OO7 benchmark [Carey et al. 1993]. Firstly the assembly hierarchy is traversed. At the completion of the assembly hierarchy traversal, a set of composite part objects are selected. This set of composite part objects are then distributed to the different nodes of the parallel OODBMS. Each node operates on the composite objects that reside on its disk and performs a depth first search on its atomic part graph. Each composite part object has a graph of 200 atomic parts with 600 connection objects arranged in a random way. The parallelisation of the atomic graph traversal is shown diagrammatically on Figure 3.2.

Figure 3.3 depicts the results of running the various object placement methods on this type of query. The following is an explanation of the labels given to the various object placement algorithms presented in Figure 3.3:

- RND: **R**andom partitioning
- RR : **R**ound robin partitioning
- DN : Similarity **d**ecustering with **n**o page clustering (Algorithm 1).
- DC : Similarity **d**ecustering with **p**age clustering (Algorithm 1 and Algorithm 2).

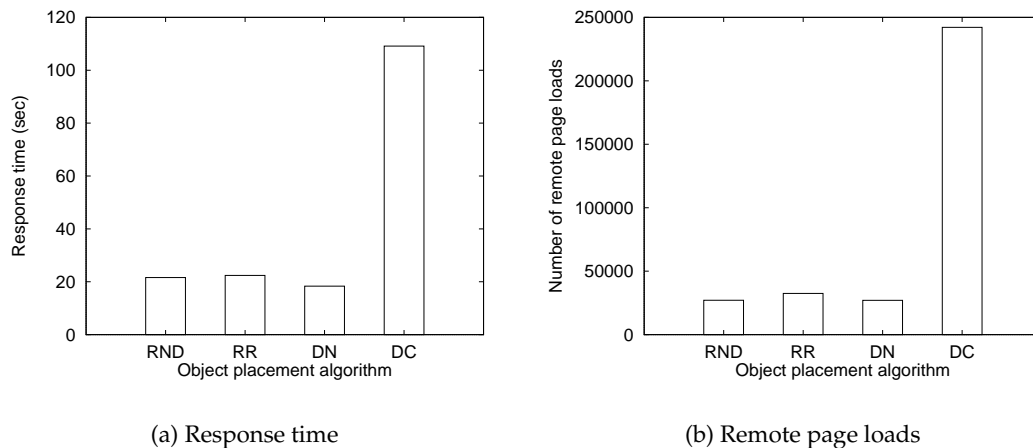


Figure 3.3: Place of failure

The two phased approach with both declustering and clustering can be seen to perform worst in both response time and the number of remote pages loaded. To explain this result, the order with which the objects are considered for placement in this query needs to be analysed. The atomic part objects are referenced the most often in this query and thus atomic part objects are considered first for placement. Considering all atomic part objects for placement before the connection objects joining them handicaps the declustering algorithm. This may be explained by the fact that in such a case the declustering algorithm does not know to group atomic part objects that reference each other via connection objects together in one node. Hence the declustering process tends to produce random placements for this query.

The clustering algorithm only clusters for objects which are one reference away from the object being considered. However here we need to cluster the atomic part and connection objects for the composite part object which may be up to 200 references away for some of the atomic parts. Therefore the clustering process tends to add an additional random process to the object placement. The poor performance of DC is thus explained.

3.2.3.1 Improved Declustering Heuristic

The solution to the problem outlined in the previous section is simple. The order in which objects are considered for placement in the declustering phase needs to be changed. The improved heuristic considers objects for placement in order of reachability. This is analogous to the window concept in similarity based declustering proposed by Liu and Shekhar [1996] and explained in section 3.1. We define the window of objects around object o_i as the objects which reference o_i or are referenced by o_i .

This new heuristic was tested and the results are shown in Figure 3.4. Where WIN corresponds to the improved two phased approach with the improved declustering

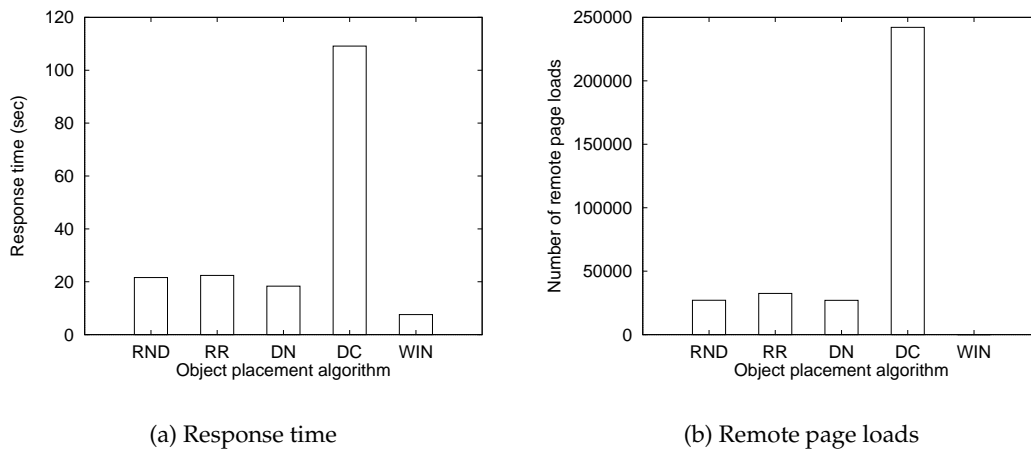


Figure 3.4: Results with Improved Heuristic WIN

heuristic and the unchanged clustering heuristic.

The new results show WIN as the best performer with zero remote page loads and the lowest response time. Zero remote page loads is achieved for this small experiment since only one query is involved in the experiment and therefore there is no potential for conflict between maximising parallelism and minimising internode object references. More complex experiments will be presented in the next chapter where conflict exists between maximising parallelism and minimising internode object references.

3.3 Summary

This chapter proposed a new similarity based approach to object placement in shared nothing parallel OODBMS. The approach is an adaptation of the similarity based declustering approach proposed by [Liu and Shekhar 1996] for the problem of declustering in a uniprocessor multiple disk environment. The change in environment required a new definition of similarity and the inclusion of a second phase termed the clustering phase. The two phased approach appears to not have been tried before.

Experiments

The primary purpose of this chapter is to investigate the performance of the new algorithms proposed in the previous chapter. To achieve this purpose two experiments were conducted. The first experiment investigates the scalability of the various placement algorithms. The second experiment investigates the ability for the various algorithms to adapt to varying training and execution conditions.

The focus of the discussion for the two experiments will be centered around the comparative performance of the various placement algorithms employed.

4.1 Experimental Methodology

The methodology used to conduct the experiments described in this chapter consists of three steps. The first step is termed the *training step*. In the training step a set of queries are run to determine the frequency distribution of object references. This information is stored in dynamic frequency counts. The training step may use either the complete set of queries to be used in the subsequent execution step or a subset of those queries. Once this step completes the second step termed the *placement step* uses information stored in the dynamic frequency counts to run the various object placement algorithms. The output of the placement step is the objects arranged and stored in the way determined by the particular object placement algorithm used. The third step called the *execution step*, where the queries used by the training step is executed to obtain performance results.

4.1.1 Test-bed

The hardware device used to conduct the experiments presented in this thesis is the AP1000. Firstly the properties of the AP1000 is presented and then our approach to modifying the AP1000 to delivering the appropriate test-bed is presented.

4.1.1.1 Properties of AP1000

The hardware platform used is the 128 node Fujitsu AP1000 [Ishihata et al. 1990]. Each node has a 25 MHz SPARC CPU with 16MB of memory and 128 KB of direct-mapped cache memory. There are 16 disks in the system.

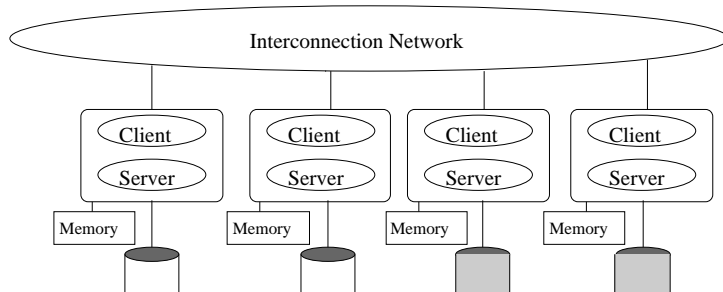


Figure 4.1: Parallel architecture used

Communication is achieved via message passing. The message passing interface MPI [Sitsky et al. 1994] has been implemented on the AP1000 and may be called via C function calls to achieve communication.

The disks of the AP1000 are managed by the HiDIOS parallel file-system. This allows fragments of one file to be loaded in parallel across the multiple disks.

The AP1000 uses a lightweight operating system called CellOS which does not provide multiple threads. Therefore programs running on the AP1000 can only operate within a single thread.

The AP1000 has a short lived kernel meaning the system is rebooted when a new program is executed.

4.1.1.2 Approach to Delivering Appropriate Test-bed Using AP1000

We have simulated a shared-nothing system on the hardware architecture of the AP1000. The first limitation to be overcome when simulating a shared nothing system on the AP1000 is the disk contention caused by the HiDIOS parallel file system when multiple nodes read at once. This was overcome by de-stripping the disk. De-stripping involves writing fragments of the file for each node into particular positions in a file. The positions chosen cause the reading of a file for a node to be free from contention when the file is loaded onto the parallel file system.

A second limitation of the AP1000 was the fact it has only 16 disks and our experiments involve running on 128 shared nothing nodes. This second limitation was overcome by the adoption of an 'air-disk'. The air-disk simulates reading from disk by pre-loading the entire file into a block of memory and then copying requested pages from the block into the application buffer on request. When a page is to be read from the air-disk, the system delays for the amount of time equivalent to the disk IO cost incurred when loading a page from disk. The disk IO cost is derived by first running an experiment in which a set of queries (the same as queries used to conduct the experiments) are run on the disk de-stripped and recording response times of page loads. The response times are then used to arrive at a model of the IO costs involved in loading a page from disk into memory. To simulate the variability in the IO cost incurred in reading a page, a small random factor was included in the model.

The client-peer architecture was simulated on the AP1000 to allow queries to be processed in parallel. The clients process the queries and the server is responsible for satisfying page requests. When a client requires a page from another node it sends a request and the server on the remote node replies with the requested page. This setup is depicted on Figure 4.1, where the grey disks depict air-disks. It is important to note this setup calls for the server and client to be operating on different threads. In this way the server can wait for remote page requests to arrive in one thread as the client processes the query in a different thread.

Due to the inability of the operating system to support multiple threads, the server had to be implemented in the same thread as the client. This means the execution step had to balance client and server workload so that the server can service remote page requests as close as possible to their arrival time. This was accomplished by using a non blocking receive [Sitsky et al. 1994]. When a page request message arrives it is buffered and at frequent intervals the receive buffer is checked. For the application we are running the receive buffer is checked between the processing of only a small number of objects. In addition each object involves only a very small amount of computation. For these reasons the receive buffer is checked very often. When a request message is found to have arrived it is serviced immediately. This way a client-peer architecture is simulated for our experiments.

4.1.2 Query Processing

Intra-operator parallelism is used to process queries for all experiments presented in this thesis. Each node works on the objects which reside on its own disk. Thus for the experiments presented in this thesis the distribution of objects across the nodes defines the degree of parallelism.

Four queries were used to conduct the experiments in this chapter. The queries were chosen to conflict with each other in terms of placement requirements. In addition the queries were chosen to represent the type of queries likely to be encountered in a typical parallel OODBMS. However, some of them maybe encountered more often than others in a typical parallel OODBMS.

The queries were run on the objects of the OO7 benchmark [Carey et al. 1993]. The actual size of the OO7 benchmark used varies according to the experiment run and thus will be explained in more detail in the relevant sections. However, subsequent descriptions of the database used for the two experiments will be presented in terms of a slightly, modified, medium sized, OO7 benchmark, hence in this section we will outline the modifications we made to the medium sized OO7 benchmark. In addition, the individual queries used in both experiments are also described in point form below and diagrammatically on Figure 4.2. The OO7 benchmark schema used in this thesis is included in appendix D.

Query-1 (a range query) The first query iterates through a set of atomic part objects and counts the number of atomic part objects with build date greater than 1500.

Query-2 (a value-based join query) The second query is based on query 8 of the OO7

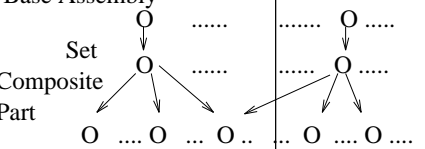
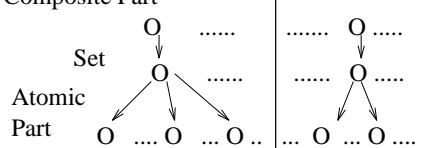
| | | | |
|--|--|--|--|
| <p>Query 1 (range query)</p> <p>Atomic Part</p> <p>O O O</p> | | | <p>eg.</p> <p>Select count(*)</p> <p>From a in AtomicPart</p> <p>Where a.buildDate > 1500</p> |
| <p>Query 2 (hash join)</p> <p>Atomic Part</p> <p>O O O</p> <p>Document</p> <p>O O O</p> | | | <p>eg.</p> <p>Select count(*)</p> <p>From a in AtomicPart</p> <p> d in Document</p> <p>Where a.docId = d.id</p> |
| <p>Query 3 (nav. sharing)</p> <p>Base Assembly</p> <p>Set</p> <p>Composite</p> <p>Part</p>  <p>O O O</p> <p>O O O</p> <p>O O O</p> | | | <p>eg.</p> <p>Select count(*)</p> <p>From b in BaseAssembly</p> <p>Where b.buildDate ></p> <p> b.compositePart.buildDate</p> |
| <p>Query 4 (no sharing)</p> <p>Composite Part</p> <p>Set</p> <p>Atomic</p> <p>Part</p>  <p>O O O</p> <p>O O O</p> <p>O O O</p> | | | <p>eg.</p> <p>Select count(*)</p> <p>From c in CompositePart</p> <p>Where c.buildDate <</p> <p> c.atomicPart.buildDate</p> |

Figure 4.2: Queries represented diagrammatically

benchmark. The query involves finding all pairs of documents and atomic parts where the document ID in the atomic part matches the ID of the document [Carey et al. 1993]. The query is processed by performing a parallel value-based hash join. The objects in two sets which are to be joined are first hashed on their join attribute and sent to nodes based on the value of the hashed attribute, this phase may be named the **distribution phase**. Note the distribution phase involves all-to-all communication and synchronisation. Once all objects have been distributed the **probing phase** starts. The probing phase receives the objects sent in the distribution phase and joins the two sets locally.

Query-3 (a navigational query with sharing) The third query is based on query 5 of the OO7 benchmark. Query 5 of the OO7 benchmark involves finding all base assemblies that use a composite part with a build date later than the build date of the base assembly. This query involves accessing shared objects. The OO7 benchmark is slightly modified for this query. The number of composite parts is increased from 500 (in original medium size OO7 benchmark) to 5000 and the size of the composite parts is enlarged. The reason for this is that testing with only 500 small composite parts on 32 nodes with a 4 KB page size results in all composite parts fitting in 1 page per node. Given that the objects are first clustered by type, if at least one composite part object is needed from every node then all the pages in the system which contain composite parts end up being loaded. This is likely to happen when both the degree of sharing is high and a large proportion of the objects are selected by the query. Such a scenario would be difficult to optimise. We did test such a case (original medium sized OO7 benchmark) and the results showed our two phased approach did not offer any improvement in speed.

Query-4 (a navigational query without sharing) The final query involves finding all composite parts with a build date earlier than its atomic part build date. It is a navigational query which does not share objects. The OO7 benchmark was slightly changed here as well. The number of atomic parts per composite part was changed from 100 to 20. The reason for this change was the number of composite parts was increased 10-fold thus the number of atomic parts needs to be decreased 10-fold to keep the size of the database at near the same size as the medium sized OO7 benchmark.

We call Query-1 and Query-2 **set queries**, and call Query-3 and Query-4 **navigational queries**. The distinction between *set* queries and *navigational* queries is that in set queries a set of objects are iterated through but no object references are traversed whereas navigational queries involve reference traversals. This is depicted diagrammatically in Figure 4.3.

It is important to note that there exist conflicts between maximising parallelism and minimising internode object references between the different queries. For Query-1 and Query-2, atomic parts need to be evenly placed on the nodes to allow a higher degree of parallelism, however Query-4 requires atomic parts to be placed together in

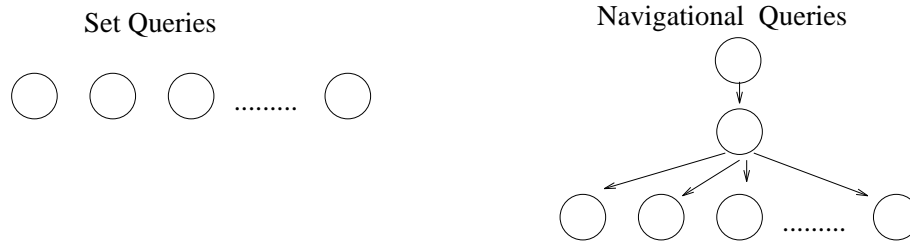


Figure 4.3: Set versus Navigational Queries

order to decrease the number of internode object traversals. Similarly, optimal placement of objects for Query-3 requires composite part objects to be placed together but composite part objects in Query-4 should be spread apart.

4.1.3 Startup and Termination

To simulate a more realistic query processing environment we have included a startup and termination cost to every query. At the start of every query a startup process is performed in which one server node sends a startup message to every client node. At the completion of every query a termination process is initiated in which every node sends a message to the server reporting the results of executing the query.

4.1.4 Cold Times

To allow better comparison of the effects of the various object placement methods on individual queries, the cache is flushed between query executions. Therefore the timing measurements presented in this chapter are derived from cold runs.

4.2 Measures of object placement quality

The response time and average time observed from running queries gives a good overall view of the performance of the various object placement methods. However there are other measures of object placement quality, including: number of internode object references; number of remote page loads; degree of client workload imbalance; and degree of server workload imbalance.

Response time Response time is taken as the total time taken to execute a query. A query does not complete until all clients have communicated their results in the termination processes. Therefore the slowest node determines the response time taken to execute a query.

Average Time Average running time is an average of the completion times of all the nodes processing the query.

Number of internode object references The number of internode object references is the number of objects requested by a particular client which does not reside on the same node as the client. This is a measure of the ability of the object placement methods to group objects which are accessed in a navigational manner together on one server.

Number of remote page loads The number of remote page loads determines the bulk of the communication costs between the client and server during the execution of a query. The other communication costs include the startup and termination costs. The clustering phase of the algorithm proposed in this thesis attempts to minimise this measure of object placement quality.

Degree of server workload imbalance The degree of server workload imbalance is a measure of the imbalance in remote page requests serviced at a particular server. Therefore, a high server workload imbalance means a disproportional number of remote page requests are targeted at one or only a few servers.

We define *server workload imbalance* as:

$$\text{server workload imbalance} = \frac{\max_{1 \leq i \leq N}(\text{server_work}(P_i)) - \min_{1 \leq i \leq N}(\text{server_work}(P_i))}{\max_{1 \leq i \leq N}(\text{server_work}(P_i))}$$

where $\text{server_work}(P_i)$ is the total number of remote pages requested from the server residing on node P_i and N is the number of nodes in the shared nothing system.

Note set queries do not contain a navigational component and thus do not request pages from remote servers. Hence server workload imbalance for set queries are always zero.

Degree of client workload imbalance Client workload imbalance is a measure of the degree of query execution parallelism for the clients. Therefore, a client workload imbalance of zero means every client processes the same number of *root objects*. The term *root object* is given to objects that are at the root of a graph traversal when the query being processed contains navigational components. Therefore, for set queries which do not contain a navigational component, the objects belonging to the set of objects iterated form the set of *root objects*.

We define *client workload imbalance* as:

$$\text{client workload imbalance} = \frac{\max_{1 \leq i \leq N}(\text{client_work}(P_i)) - \min_{1 \leq i \leq N}(\text{client_work}(P_i))}{\max_{1 \leq i \leq N}(\text{client_work}(P_i))}$$

where $\text{client_work}(P_i)$ is the number of objects in the set of *root objects* processed by the client residing on node P_i and N is the number of nodes in the shared nothing system.

4.3 Response Time vs. Average Time

Response time is a useful measure of performance when synchronisation is required before queries are permitted to terminate. In such a situation all nodes must wait for the slowest node to complete before the next query can begin its execution. In contrast average running time is useful when a node is permitted to begin execution of successive queries as soon as it has completed the query it is currently running.

For scalability experiments average time and response time often exhibit increasing difference as the number of processors increase. This is the case since as the number of processors increase then so does the probability of finding outliers¹.

4.4 Scalability Experiment

In this experiment we investigate the scalability of the various algorithms proposed in this thesis. The scalability of an algorithm depicts its potential usefulness as a way of placing objects in a parallel OODBMS. The reason is the scalability of an algorithm is a measure of the ability of the algorithm to perform efficiently as the problem size increases in proportion to the number of processors.

For this experiment the four queries described in Section 4.1.2 were tested on the objects of a varying sized OO7 benchmark [Carey et al. 1993]. Varying the size of the OO7 benchmark allowed us to test the scalability of our algorithm by observing response time as the number of processors increase in proportion to problem size. As a base case a slightly modified (as explained in Section 4.1.2) medium sized OO7 benchmark was used for 32 nodes. Since the problem size was increased in proportion to the number of processors, a perfectly scalable system would depict a constant response time as the number of processors increase.

In this experiment the training conditions were the same as the execution conditions. Therefore this experiment depicts the perfect situation in which the query distribution information stored in the dynamic frequency counts reflect perfectly the distribution of future object accesses.

For this experiment the declustering phase was biased in favour of reduced internode object references as opposed to increased parallelism (by setting α from Equations 3.1 and 3.2 to 0.9). The reason behind this decision was, from previous experiments the performance of setting α to less than 0.9 was observed to cause all the algorithms to degrade in performance for navigational queries and only slightly improve the performance of set queries. This is evident in the second experiment presented in this chapter where both α settings of 0.1 and 0.9 are tested.

All the results presented in this experiment was collected by executing each query 10 times and storing the accumulated values of the 6 measures of clustering quality. The accumulated values were then normalised with respect to the base case (random placement with 2 processors). This was done to allow better comparison of results between the different queries.

¹nodes completing with abnormally high or low completion times.

The scalability information present in the running time graphs in this experiment convey a combination of the scalability of the particular object placement method employed and query execution method employed. Wherever applicable, the distinction will be made when discussing these results.

In preceding sections the various object placement methods will be referred to as follows:

- RND: **R**andom partitioning
- RR : **R**ound robin partitioning
- DN : **S**imilarity declustering with no page clustering (Algorithm 1).
- DC : **S**imilarity declustering with page clustering (Algorithm 1 and Algorithm 2). Objects in the declustering phase are considered in order of non-increasing frequency of traversal.
- WIN: **S**imilarity declustering with page clustering. Objects in the declustering phase are considered in order of reachability. Analogous to the **w**indow-based approach to similarity declustering [Liu and Shekhar 1996].

General notes to keep in mind while investigating the results presented for this experiment:

Running time graphs Response time is represented in dotted lines, average time is represented in solid lines.

Client workload imbalance graphs DC was removed from this graph since the clustering phase does not attempt to balance the client workload and hence DC and DN exhibit the same degree of client workload imbalance.

Internode object reference graphs DC was removed from this graph since the clustering phase does not change the number of internode object references, hence DC and DN exhibit the same number of internode object references.

The following sections will contain: four separate individual investigations of the four queries tested; investigation of server workload imbalance results for queries 3 and 4; a comparison of the four queries relative scalability; and overall measurements from running all four queries.

4.4.1 Individual Queries

During analysis of the results for this experiment it was observed that the behaviour of the different queries varied vastly from each other and many interesting observations can only be made when the queries are analysed individually. Therefore in this section we present the results obtained from separate investigations of the four queries involved in this experiment. However it is important to note all four queries were involved in the training step and thus the placement algorithms still had to balance the conflicting objectives in the four queries as explained at the end of Section 4.1.2.

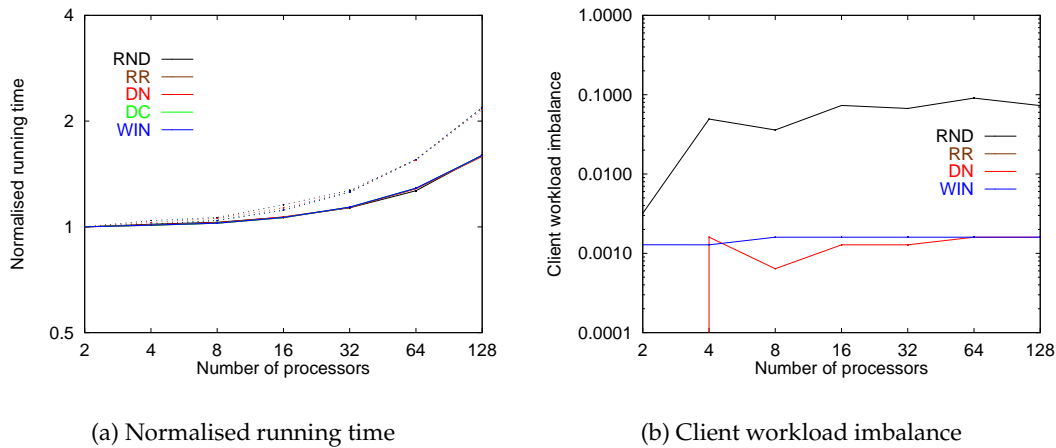


Figure 4.4: Scalability results for query 1

4.4.1.1 Query 1

Query 1 is a simple set query in which a set of objects are iterated through across the nodes of the shared nothing system in parallel. Figure 4.4 shows the performance of query 1 for both running time and client workload imbalance.

To explain the running time results we must first examine the client workload imbalance graph on Figure 4.4 (b). The degree of client workload imbalance is not large for any of the placement methods investigated. Even for RND, the maximum client workload imbalance is only 0.1. The reason for RND to be so low is that the random number generator is not programmed to discriminate against any set of nodes when assigning objects to nodes. When the number of objects placed is large (as is the case for this query) then the number of objects assigned to each node is close to the expected number of objects per node². RR is absent from the graph since for this query the number of processors divided exactly into the number of objects iterated through. It is encouraging to note that DN and WIN both perform well at around 0.001 client workload imbalance despite the conflict between query 1 and query 4 as outlined in the end of Section 4.1.2. However, from these observations it should be noted the magnitude of client workload imbalance is not large for any of the placement methods investigated.

From Figure 4.4 (a), the response time (dotted line) and average time (solid line) of executing query 1 can be observed. The performance of the different object placement methods was around the same for both response time and average time. This may be explained by the absence of a navigational component to query 1 and the low degree of client workload imbalance for all the object placement methods investigated. The absence of a navigational component to query 1 handicaps the placement algorithms

²total number of objects divided by the number of nodes.

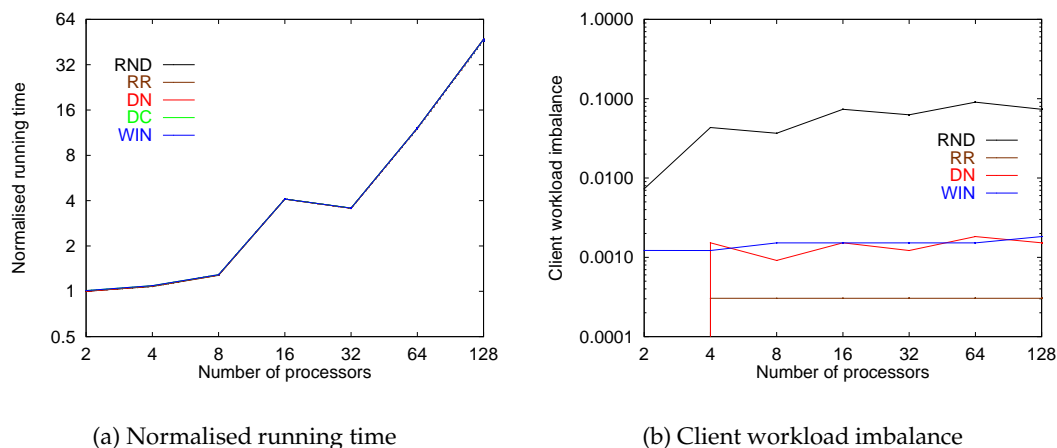


Figure 4.5: Scalability results for query 2

proposed in this thesis by removing the requirement to decrease internode object references.

Both average time and response time results for query 1 show almost perfect scalability. This is due to the simplicity of query 1. Query 1 simply required each node to iterate through a set of objects that resided on its own disk. No communication was required other than during the startup and termination processes. However the scalability results are not perfect, this may be explained by the fact as the number of processors increase then so does the startup and termination costs. However, if the number of objects processed was increased to a point where the query execution time dominated the startup and termination time, then an even more scalable running time graph would result.

4.4.1.2 Query 2

Query 2 is a parallel value-based hash join in which two sets of objects are joined on a particular attribute. Figure 4.5 shows the performance of query 2 for both running time and client workload imbalance.

As for query 1, the running time of the various placement algorithms did not differ by much. There are two reasons for this. The first is the same as for query 1 — namely the absence of a navigational component handicaps the object placement algorithms developed in this thesis. The second reason is more particular to query 2. Query 2 is a parallel value-based hash join. Parallelism for this query is not solely determined by the object placement algorithms. In the distribution phase the objects are redistributed via a hash function. The probing phase follows the distribution phase and thus the degree of parallelism for the probing phase is determined by the hash function used in the distribution phase and not the object placement algorithms. Therefore query

2 is not very sensitive to client workload imbalance caused by the object placement algorithms.

Query 2 does not appear to be scalable for any of the object placement algorithms employed. The reason is that the distribution phase requires synchronisation and all to all communication. As the number of processors increases, the communication costs increases rapidly.

The average and response times do not differ by much since the synchronisation required at the completion of the distribution phase causes all the nodes to finish at about the same time.

4.4.1.3 Query 3

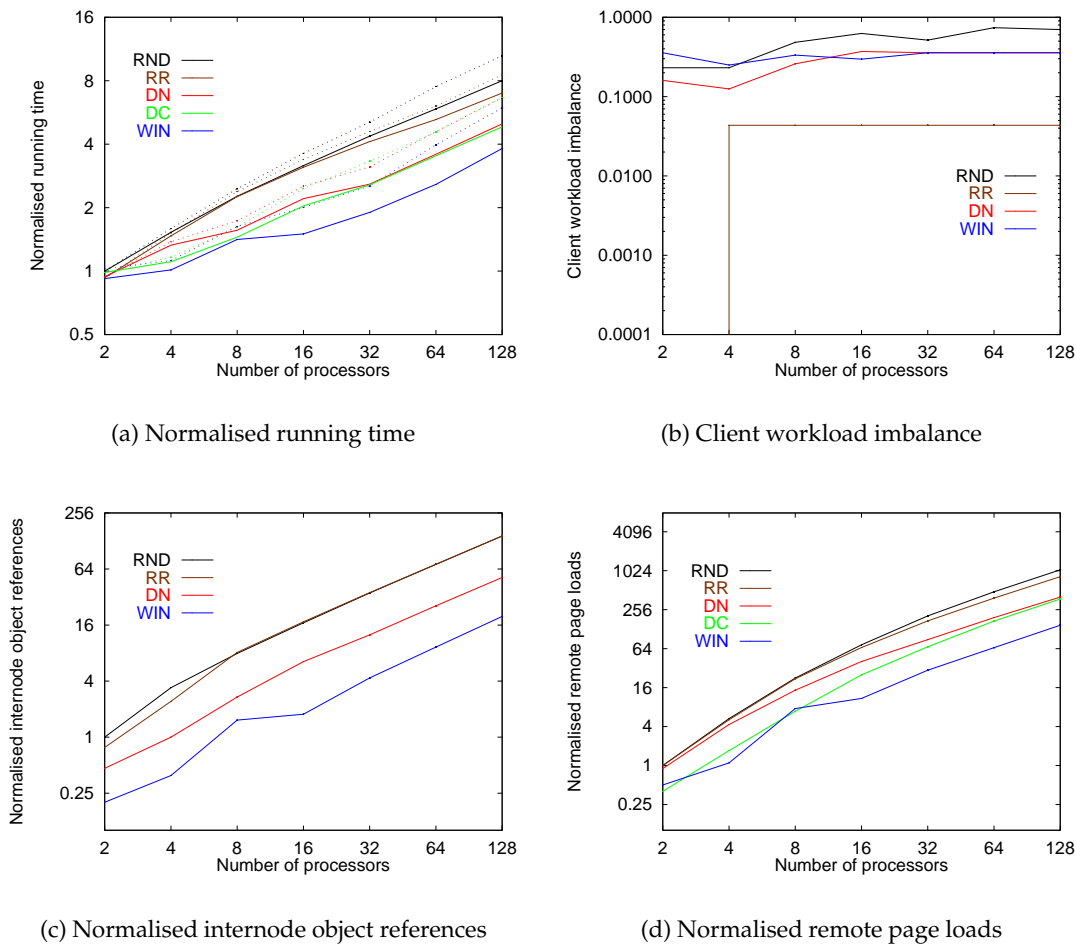


Figure 4.6: Scalability results for query 3

Query 3 is the first query which contains a navigational component. In addition,

query 3 contains sharing of objects referenced.

Figure 4.6 (a) depicts the running time behaviour of this query. The first observation that can be made is that for both response time and average time, all the algorithms proposed in this thesis, DN, DC and WIN outperform both RR and RND with WIN being the best performer. This may have been expected since DN, DC and WIN attempt to reduce communication costs by reducing the number of remote pages loaded whereas RR and RND do not.

The second observation that can be inferred from Figure 4.6 (a) is that the performance difference between DN and DC as the number of processors is increased above 16 is almost non-existent. This behaviour is supported on Figure 4.6 (d), where the number of remote page loads of DN and DC converge as the number of processors increases. This seems to suggest the clustering phase of the algorithm become, less beneficial as the number of processors increases. After investigating this observation the following explanation is arrived at:

- The number of internode object references *per* processor stays close to constant as the number of processors is increased.
- The problem size for this query is small with only 110 objects per node.
- There is only a small number of internode object references for both DN and DC in this query. This is caused by a combination of the high quality of the declustering phase present in both DN and DC and the small problem size of this query.
- As the number of processors increase the number of internode object references directed at a particular server decreases. Beyond 32 nodes many servers contain only one object for a particular client.

From the points stated above it can be seen the number of internode object references per processor does not scale up as the number of processors increase. This means as the number of processors increase, internode object references for a particular node becomes more dispersed across the increasing number of servers. When the number of objects residing at a particular server for a particular client is only one (as is the case for 32 or more nodes), the object placement algorithms can no longer be optimised by clustering. Therefore after the declustering phase the number of internode object references for this query was too small for the clustering phase to be of any real benefit.

The degree of client workload imbalance for DN and WIN in query 3 as depicted on Figure 4.6 (b) is up to 300-fold larger than for either of the set queries 1 or 2 shown earlier. A possible reason for DN and WIN to perform worse than before is that there is sharing of objects in this query. When two objects both reference the same object being shared then they are likely to be placed in the same node even if a larger client workload imbalance may result. This is more likely to happen when α (from equations 3.1 and 3.2) is set to 0.9 as is the case for this experiment. A possible reason why the client workload imbalance of RND is higher for this query is that the problem size of

this query is small and hence the actual number of objects assigned to each node may tend to be further from the expected number³.

Finally from Figure 4.6 (a) query 3 appears to be only moderately scalable, with an 8-fold increase in running times of the various algorithms, corresponding to a 64-fold increase in the number of processors. There are two reasons for this. The first is the execution time of this query is only in the order of 0.2 seconds, hence the execution time is small enough for the startup and termination time to play a large part in determining the scalability of the query. The second reason is as the number of processors increase the number of remote page loads also increase, causing increased network contention.

4.4.1.4 Query 4

Query 4 is another navigational query but with no sharing of objects. This query is larger than the previous query with a problem size of 3280 objects per node instead of 110 objects per node.

Figure 4.7 (a) shows the response time of running query 4. Again the best performers are the algorithms proposed in this thesis. This can be again explained by the fact the algorithms proposed attempt to reduce remote page loads while the others do not.

When comparing Figures 4.7 (c) and (d) a very counter-intuitive result can be observed, WIN results in more internode object references when compared to DC⁴ but the reverse is true when the number of remote page loads is observed. The graphs seem to suggest the more objects requested from remote servers the fewer number of pages is loaded. A top-level explanation for this peculiar result is that WIN declusters the objects in a way which benefits the clustering phase more than the declustering phase present in DC.

Firstly, an explanation needs to be given for the declustering phase of WIN to perform worse than DC in reducing the number of internode object references. This can be explained by the order in which objects are considered for placement by WIN. By considering objects in order of reachability, WIN attempts to place all the atomic part objects referenced by a particular set object in succession. This causes the algorithm to think that there is likely to be client workload imbalance for query 1 and 2 (where atomic part objects are parallelised), hence WIN incorrectly places many of the atomic objects referenced into other nodes. However this is not as likely to be the case for DC where objects are not considered in order of reachability.

Secondly, an explanation needs to be given for the clustering phase⁵ to benefit more from the declustering phase of WIN when compared to the declustering phase of DC. This can be explained by observing Figure 4.8. From Figure 4.8 it can be seen that WIN places set objects and composite part objects in the same node and hence the

³Total number of objects iterated divided by the number of nodes.

⁴Note as stated earlier DC and DN share the same first phase and hence have the same number of internode object references. Therefore results for DN in Figure 4.7 can be used in-place of results for DC

⁵Both WIN and DC share the same clustering phase

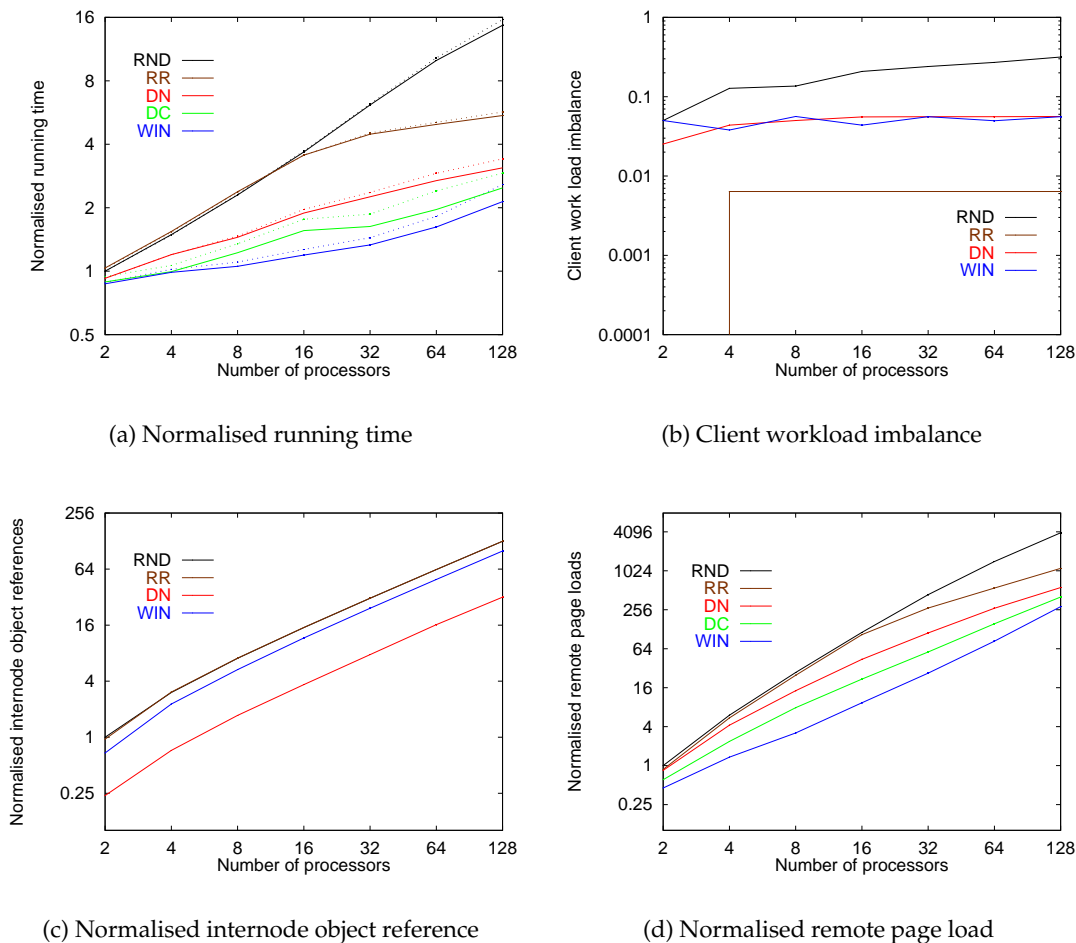


Figure 4.7: Scalability results for query 4

atomic part objects are *clustered* for the *correct* node. However DC places set objects and composite part objects in different nodes and hence the atomic part objects are *clustered* for the *incorrect* node. This, however, does not explain why the two placement algorithms WIN and DC behave in the way described in Figure 4.8. For the curious a detailed explanation of the reason why WIN and DC behave in the way exhibited in Figure 4.8 is included in Appendix A. It may be concluded from this peculiar result that the number of internode object references is not always a good indication of the relative number of remote page loads of two placement algorithms when using page grain caching.

The degree of client workload imbalance for query 4 is exhibited in Figure 4.7 (b). The degree of client workload imbalance for DN and WIN in query 4 is smaller than for query 3. This may be explained by the absence of object sharing and a larger problem size.

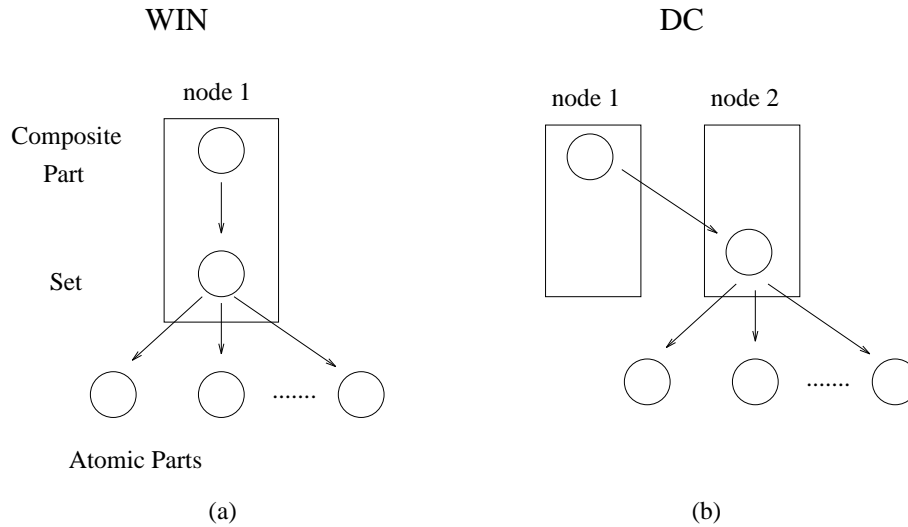


Figure 4.8: Diagram to aid the explanation of the peculiar result observed for query 4

The average time and response times of the various algorithms for query 4 can be seen on Figure 4.7 (a). The average and response time results for this query do not differ as much as was for query 3. This may be due to the lower client workload imbalance for query 4, exhibited in Figure 4.7 (b) when compared to that of query 3 in Figure 4.6 (b).

This is the first query in which the scalability of the various placement algorithms differs noticeably. The algorithms proposed in this thesis are clearly more scalable when compared to RND and moderately more scalable when compared to RR. A possible reason for this is that this query contains a large percentage of navigational component and as the number of processors increase then so does the difference in the amount of network contention between the different algorithms.

4.4.2 Server workload imbalance

Only the navigational queries 3 and 4 require the servers to serve remote page requests. Therefore this section only presents the server workload imbalance of queries 3 and 4.

The first observation that can be seen from the two graphs presented in Figure 4.9 is that the degree of server workload imbalance appears to be random for algorithms proposed in this thesis, namely DN, DC and WIN. This may be explained by the fact the issue of server workload imbalance was not fully addressed by the algorithms proposed in this thesis. DN, DC and WIN attempted to reduce server load imbalance by placing tied objects on nodes which have received the least number of ties. However this is not a satisfactory solution since tied objects occur for many reasons. Server workload imbalance is only one among many. For instance tied objects may occur

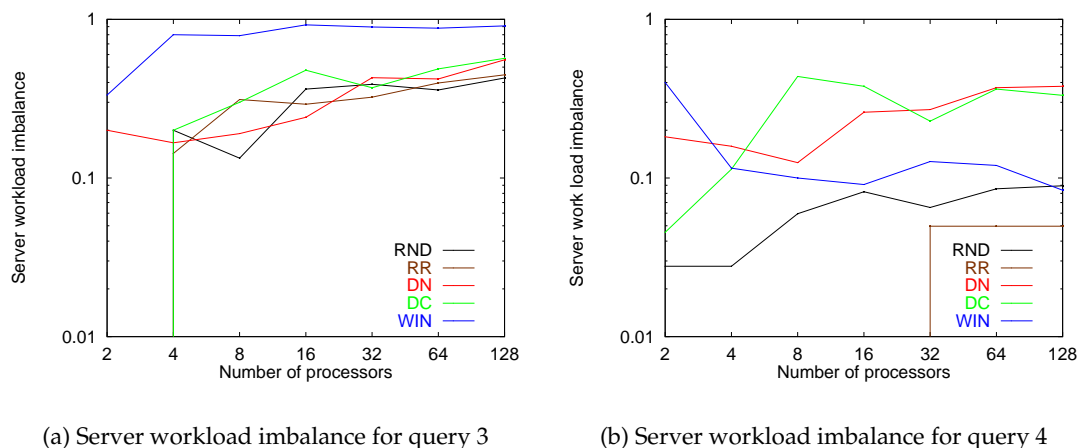


Figure 4.9: Server workload imbalance results

because the algorithm can not decide the best node to place an object to, in order to balance client workload.

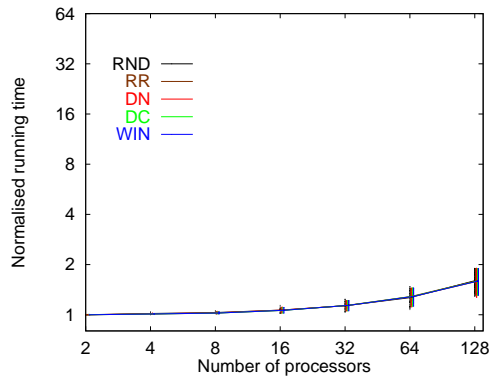
Future enhancements of the algorithms proposed in this thesis may include heuristics that address server workload imbalance in a more satisfactory manner.

4.4.3 Scalability Comparison of Individual Queries

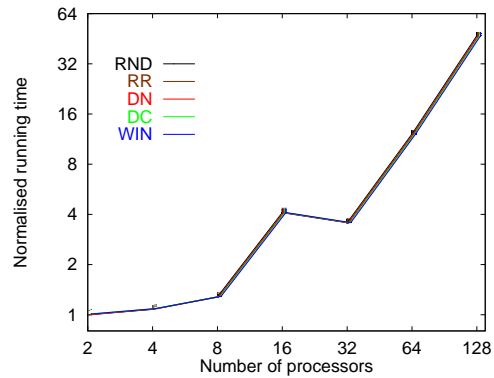
In this section we present graphs containing the individual completion times of every node involved in the query processing. In addition average running time is also depicted on the same graphs. The graphs are all adjusted to have the same x and y range to allow better comparison of results. These graphs are depicted on Figure 4.10. For an alternative comparison containing response times, see Appendix B.

The scalability of the queries vary vastly, from a near perfectly scalable query 1 to a very non-scalable query 2. From these graphs it can be seen that in general the particular type of query run is more important in determining scalability than the particular placement method employed. However Query 4 may be an exception, where the scalability of RND is noticeably worse than the other placement methods. A possible explanation is query 4 when compared to the other queries contains the largest percentage of object references. As the number of processors increase and along with it the total number of objects to be processed, the network contention starts to play an increasingly larger part in determining the scalability of the system.

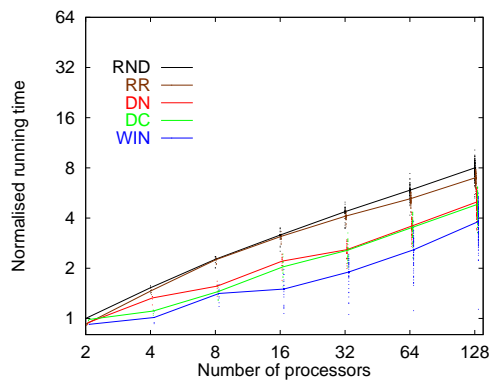
These graphs also makes clear the fact that for set queries 1 and 2, the difference in performance between object placement methods is small. However for the navigational queries 3 and 4, a larger performance difference can be seen. This seems to indicate object placement methods should only be used to place objects in an application when navigational queries occur frequently. It may be argued that parallel OODBMS



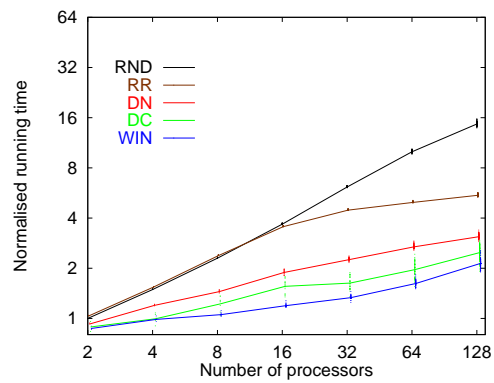
(a) Query 1



(b) Query 2



(c) Query 3



(d) Query 4

Figure 4.10: Comparison of scalability of individual queries. This figure depicts the individual completion times of every query along with the corresponding average running times (note that individual times for each query are slightly offset on the x-axis to avoid overlap).

are dominated by navigational queries since object references occur frequently.

4.4.4 Overall Results for all four queries

So far we have only shown the individual performance of the four queries tested. However in this section we show the combined effects of executing them.

Note as explained earlier at the completion of every query the buffer was flushed, thus the results shown in this section depicts the effect of executing the four queries with buffers initially being empty.

The results presented in this section depict the summation of the normalised contribution of each query.

4.4.4.1 Running Time

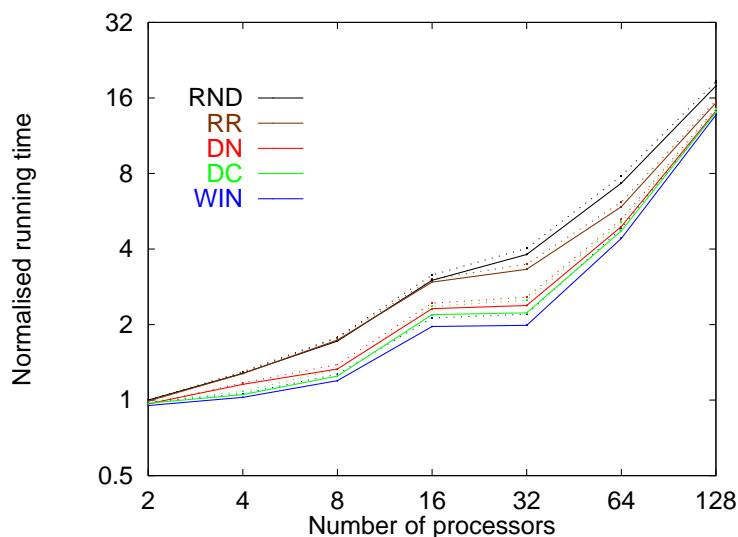


Figure 4.11: Running time vs. number of processors for all four queries

Figure 4.11 depicts both the response time (dotted lines) and average time (solid lines) results obtained from running all four queries.

Two general observations can be made from Figure 4.11, the first is the object placement algorithms presented in this thesis outperform both RR and RND. However, the margin of improvement is not large and the difference between DN, DC and WIN is even smaller. The second observation is the combination of the four queries run do not seem to be scalable for any object placement algorithms tested. The reason behind both of these observations is the fact that the parallel value-based hash join (query 2) dominates the execution time of all four queries even after the contribution of each query was normalised. The reason being that query 2 was so much less scalable when compared to the other queries.

4.4.4.2 Number of Remote Page Loads

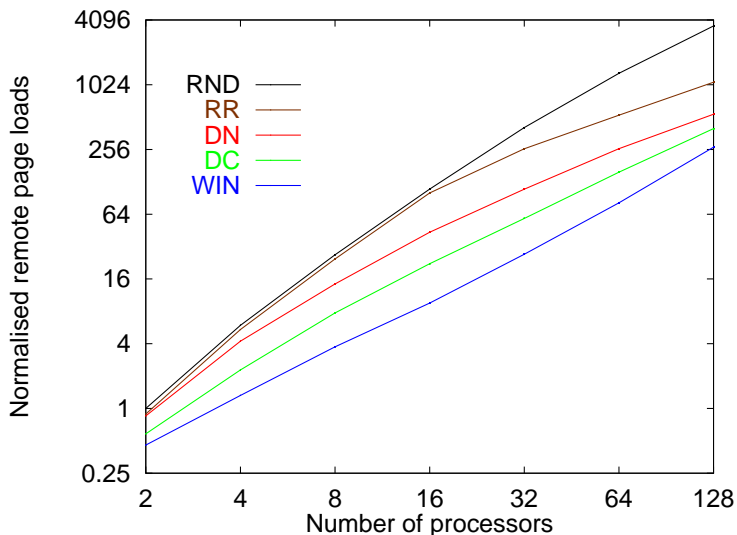


Figure 4.12: Remote page loads vs. number of processors for all four queries

Figure 4.12 depicts the overall performance of the object placement methods investigated in terms of the number of remote page loaded. Overall WIN is clearly ahead with up to 16 times reduction in the number of remote pages loaded when compared to RND. Note that only the navigational queries 3 and 4 contribute to this graph. The graph shows the potential for a placement algorithm to reduce the amount of communication during processing of navigational queries.

4.4.4.3 Degree of Client Workload Imbalance

Figure 4.13 shows the overall client workload imbalance from the combined effects of executing all four queries.

The overall observation that can be made is that the degree of client workload imbalance was not large for any of the placement methods and therefore did not play a major part in separating the execution time performance of the various placement methods. It is, however, encouraging to note that at an α^6 setting of 0.9 (biasing heavily in favour of reduced internode object references), the algorithms proposed in this thesis still managed to balance client workload so well.

4.4.4.4 Scalability Experiment: Summary

The results presented for this experiment indicate the placement methods developed in this thesis improves the overall performance for the four queries on which the ex-

⁶From Equations 3.1 and 3.2

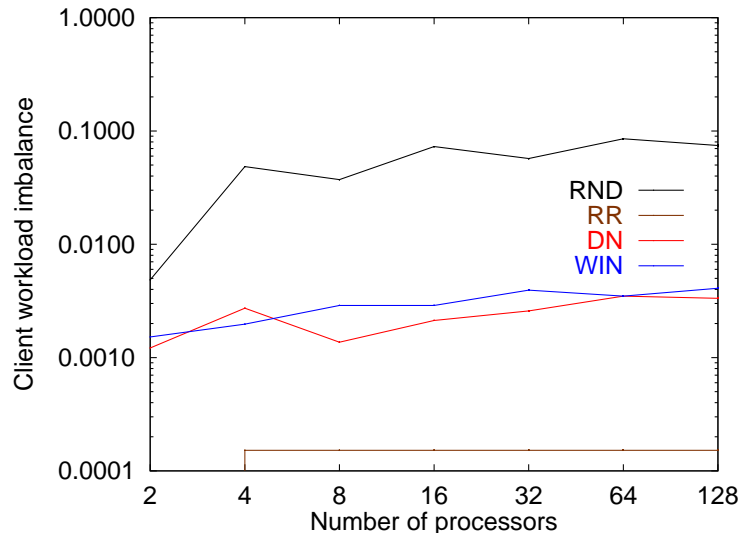


Figure 4.13: Client Workload Imbalance vs. number of processors for all four queries

periment was conducted. The overall performance improvement was not as large as we would have liked. This was mainly due to the parallel value based hash join (query 2) dominating the overall performance. However when the navigational queries 3 and 4 were considered in isolation, the algorithms proposed in this thesis performed significantly better than RR and RND which do not attempt to reduce remote page loads.

When comparing placement algorithms the number of internode object references is not always a good indication of the number of remote page loads. As was exemplified by the fact that for query 4 the number of internode object references was higher for WIN than DC but the reverse is true when the number of remote page loads was observed for the same two placement algorithms.

The scalability of the different queries differ greatly but the scalability of the different algorithms differ only for query 4 where the problem size is large and the percentage of navigational component present is large.

Overall the degree of client workload imbalance did not appear to separate the performance of the different algorithms by much. This was due to the small magnitude of client workload imbalance exhibited by all algorithms investigated and the small amount of computation required per object processed. However it is important to note that the queries tested in this experiment had conflicting requirements in terms of one query requiring objects to be grouped together while another query requiring the same objects to be placed apart. It is encouraging to note in such an environment the placement algorithms proposed in this thesis still managed to maintain a high degree of parallelism.

Considering objects for declustering in order of reachability as exhibited by WIN seems to be the best placement algorithm overall. The alternative, declustering in order of decreasing frequency count as exhibited by DN and DC appears to perform

worse since often objects which reference each other indirectly, are placed in separate nodes.

4.5 Varying Training Conditions Experiment

This experiment investigates the dependency between the percentage of set queries used for training the object base and the percentage of set queries used during execution. This is a test of the ability of the various object placement algorithms to adapt to conditions where the training conditions differ from execution conditions. Each run involved execution of a total of 100 queries with varying percentage of set and navigational queries. This experiment was run on 16 nodes with the parallel file-system de-striped, therefore disk contention was avoided.

The size of the database used for this experiment is half the slightly⁷ modified medium sized OO7 benchmark.

For this experiment α ⁸ settings of 0.1 and 0.9 were both investigated. Placing α at 0.1 corresponds to biasing the declustering phase in favour of maximised parallelism. An α value of 0.9 corresponds to biasing the declustering phase in favour of reduced internode object references.

Note that in all the results presented in this chapter dotted lines are used to depict an α setting of 0.1 and solid lines are used to depict an α setting of 0.9. Also DC results are omitted from the internode object reference results and client workload imbalance results as was the case for the scalability experiment.

4.5.1 Response Time

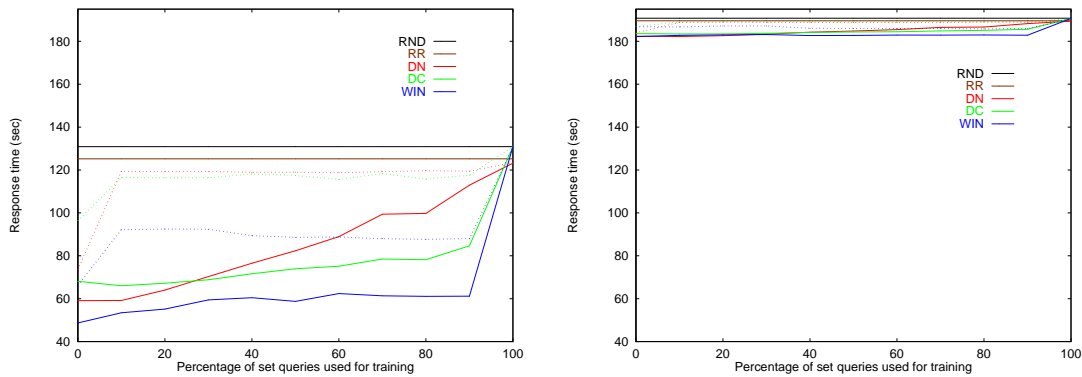
Figure 4.14 shows the effect on response time of varying training conditions for two cases of execution conditions. The first case depicts the result of executing 10% set (90% navigational) queries and the second case depict results from executing 90% set (10% navigational) queries.

When comparing Figures 4.14 (a) and 4.14 (b) it can be observed, for the 10% set queries case, the response time vary vastly depending on the placement method employed while for 90% set queries the response time is around the same for all placement methods. This reinforces the observation made in the scalability experiment, where, navigational queries were found to be more sensitive to the different algorithms when compared to set queries.

For the 10% set queries (90% navigational queries) case, the object placement algorithms proposed in this thesis with α set to 0.9 in general outperform the same placement methods with α set to 0.1. This is the case since a large α value of 0.9 corresponds to placing greater importance in reducing remote page requests and hence benefits navigational queries.

⁷For an explanation of reason for modifying medium size OO7 benchmark, see Section 4.1.2.

⁸From equations 3.1 and 3.2



(a) Executing with 10 % set queries, 90% navigational queries

(b) Executing with 90 % set queries, 10% navigational queries

Figure 4.14: Response time results. The dotted line curves for DN, DC and WIN represent response time results with an α setting of 0.1. The solid line curves for DN, DC and WIN represent response time results with an α setting of 0.9.

An important observation from Figure 4.14 (a) is that both WIN and DC at an α setting of 0.9 exhibit almost constant response time as the training conditions move further from execution conditions. This seems to suggest WIN and DC at an α of 0.9 can adapt to conditions where the training data used does not reflect future execution conditions well. This is due to the presence of the clustering phase in both WIN and DC. The clustering phase does not attempt to balance parallelism with internode object references and thus is not sensitive to the changing training conditions. However for DN at an α value of 0.9 the response time increases rapidly as training conditions tend further from running conditions. This may be explained by DN only running the declustering phase, which needs to balance the two conflicting objectives of maximised parallelism and minimised internode object references and hence is more sensitive to the training conditions moving further from the running conditions.

Note that at training of 100% set queries all the placement algorithms proposed in this thesis converge at a high response time. This can be explained by the absence of information required by the declustering and clustering algorithms to reduce remote page loads and internode object references.

Note that both RR and RND exhibit constant response time on both response time Figures 4.14 (a) and (b). This may be explained by the fact RND was generated using the same seed for each run. Both RR and RND do not require training and thus the response time results for RR and RND are independent of training.

The last observation to note from Figure 4.14 (a) is that WIN at an α setting of 0.9 appears to outperform all other object placement algorithms, as was the case for the navigational queries in the scalability experiment.

4.5.2 Communication costs

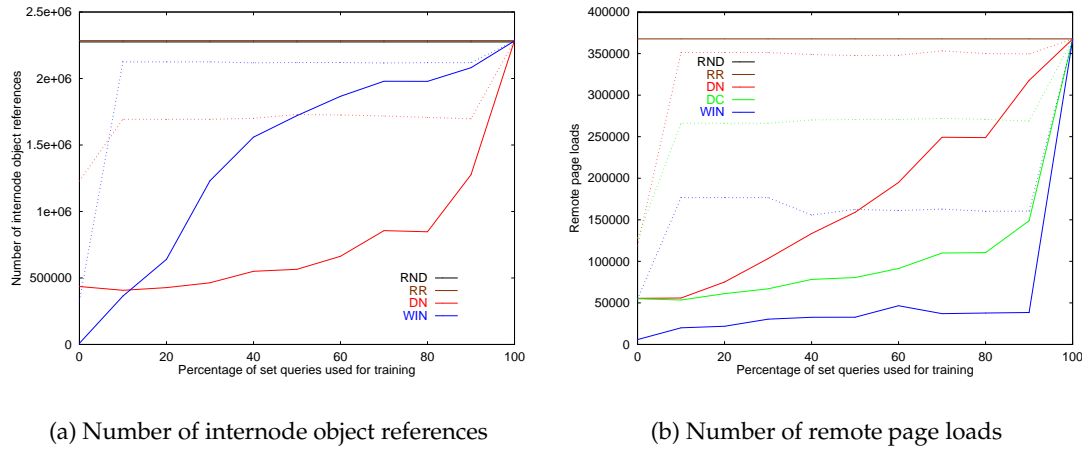


Figure 4.15: Executing with 10% set queries, 90% navigational queries

For the purposes of this section the term ‘communication cost’ will be used to refer to the number of internode object references and remote page loads incurred by the various algorithms. The communication cost behaviour of the 10% and 90% set queries cases were observed to be identical. The reason being that only navigational queries contain communication costs, therefore the 10% set queries (90% navigational queries) case would have a communication cost of 90x as opposed to the 90% set queries (10% navigational queries) case which would have a cost of 10x, where x is the communication cost involved in executing 1% navigational queries (99% set queries). Hence, only the results for the 10% set queries case are used for discussion in this section. The graphs for the 90% set queries case is included in Appendix C.

Figure 4.15 presents the communication cost graphs for the 10% set (90% navigational) queries case. A general observation that can be made is that the algorithms proposed by this thesis outperformed both RR and RND. This is not surprising since RR and RND do not attempt to reduce remote page loads or internode object references. Again better results are obtained when the value of α of the various algorithms are set at the higher value of 0.9. The explanation for this observation was given in the previous section.

The most interesting observation that can be made from Figure 4.15 (a) is that for WIN at an α setting of 0.9 the number of internode object references increase rapidly as training conditions become increasingly non favourable. Whereas DC⁹ at the same α setting exhibit a much smaller rate of increase in the number of internode object

⁹Note as stated earlier DC and DN share the same first phase and hence have the same number of internode object references. Therefore results for DN in Figure 4.15 (a) can be used in-place of results for DC

references. However, in figure 4.15 (b) the number of remote page loads for WIN at an α setting of 0.9 stays constantly lower than DC at the same α setting. This seems to be similar to the observation made for the peculiar behaviour of WIN in query 4 of the scalability experiment, namely that a high number of internode object references does not necessarily mean a higher number of remote page loads. This may be explained by the following points:

- The queries included in this experiment only include two navigational queries 3 and 4.
- Equal portions of query 3 and query 4 were used in this experiment.
- Query 4 involves a much larger problem size when compared to query 3. Hence query 4 dominates the communication cost behaviour of the queries run in this experiment.

Therefore the same arguments presented for the discrepancy in communication cost results for WIN in query 4 in the previous experiment can be reapplied to explain the behaviour of WIN with an α value of 0.9 in this experiment.

4.5.3 Degree of Parallelism

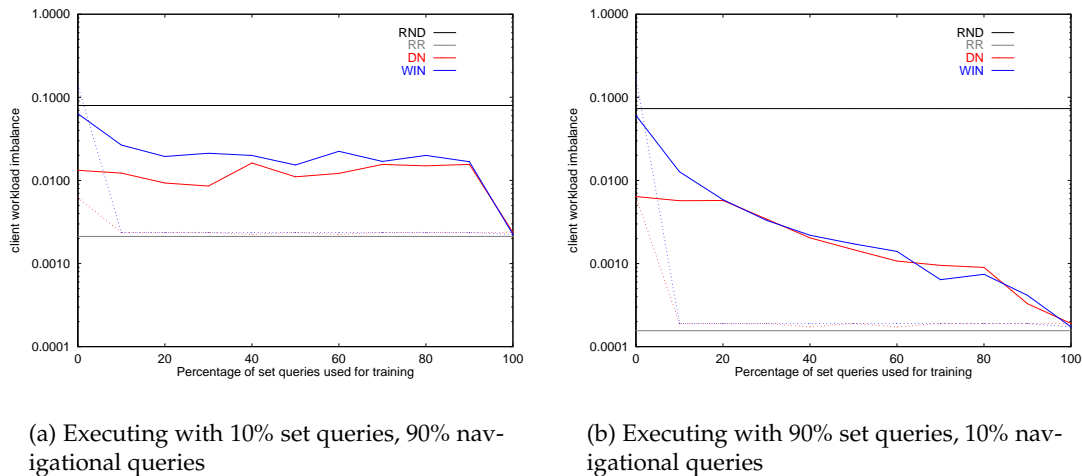


Figure 4.16: Client workload imbalance results

Figure 4.16 depicts client workload imbalance for both the 10% set queries case and the 90% set queries case. The algorithms proposed in this thesis when given an α setting of 0.1 resulted in an object placement which gave lower client workload imbalance when compared to the higher α setting of 0.9. This is to be expected since assigning an α value of 0.1 gives more importance to maximising parallelism. However, even for an α value of 0.9 the algorithms still performed relatively well with

around a 0.02 client workload imbalance for the 10% set queries case and even lower for the 90% set queries case. This suggests for the set of four queries investigated in this chapter assigning more importance to maximising parallelism may not be of as much benefit as assigning more importance to reduced internode object references.

4.5.4 Varying Training Conditions Experiment: Summary

Placing more importance on reduced internode object references by setting α to a higher value produces much better response time for the 10% set queries (90% navigational queries) case. This seems to suggest the biasing term α is working as it was intended to.

The algorithms which had a clustering phase present resulted in placements that yielded near constant response time with varying training conditions. Whereas algorithms which only had the declustering phase present produced rapid increase in response time when training conditions moved further from running conditions. The explanation for this observation is that the declustering phase had to balance the two conflicting objectives of reduced internode object references and increased parallelism whereas the clustering phase did not. Therefore this seems to suggest that if training conditions are likely to be different from running conditions then the clustering phase should be used when placing objects.

Again it was observed that the number of internode object references is not an accurate indication of the number of remote page loads. This was shown by WIN at an α value of 0.9 exhibiting very rapid increase in the number of internode object references but resulting in a low constant number of remote page loads.

This experiment, like the scalability experiment exhibited, only a small degree of client workload imbalance for all the placement algorithms. Hence, for this experiment the degree of client workload imbalance did not play a important role in separating the performance of the various object placement methods investigated.

Lastly, this experiment reinforced the observation made from the scalability experiment, which was that set queries were not sensitive to object placement methods. This was shown to be the case in this experiment when the percentage of set queries were set to a high of 90% and the resultant response times of the various placement algorithms did not differ by much.

4.6 Summary

In this chapter we conducted two investigations into the performance of the two phased approach to object placement developed in this thesis. The findings indicate that the object placement algorithms proposed in this thesis are beneficial for navigational queries. However, set queries were found to be insensitive to the object placement methods investigated.

When the training conditions were varied from execution conditions, the clustering phase was found to be important in keeping the response time constant with varying training conditions.

Parsets

Until now we have only considered operations on objects belonging to the same type set. However programmers often perform operations on a sub set of objects belonging to a type set. In such cases a parallel set can be defined on which a method can be invoked on every object in the set in parallel.

This chapter outlines the issues that arise from parallelising a sub set of a type set. In addition it also provides a method for determining the optimal degree of declustering of such a set.

5.1 Background

This idea of parallel sets was first explored by Kilian [1992] for adopting the data parallel approach to C++. However similar ideas has appeared prior to Kilian [1992] for different languages. Bancilhon, Briggs, Khoshafian, and Valduriez [1987] proposed a similar facility for Bubba (a parallel RDBMS).

DeWitt, Naughton, Shafer, and Venkataraman [1996] proposed a parallel set design for parallelising C++ code on the SHORE Persistent Object Store [Carey et al. 1994]. Their goal was to provide system primitives that make it easy for a programmer to consciously and explicitly parallelise his or her own OODBMS application. They left object placement the responsibility of the programmer. A programmer had to specify the placement strategy employed on each parset. They included, hash, range, random and unspecified. Unspecified object placement allows the user to manually specify the placement of the objects of the parset. The motivation for this option was that there are cases when no mapping exists from a value of an object to a processor number. For example, if objects are stored on the nodes where they were created there need not be any logical mapping from object values to nodes. Automatic declustering was not supported.

We believe automatic object placement is critical for the efficient use of parsets. We justify this statement in Section 5.3.

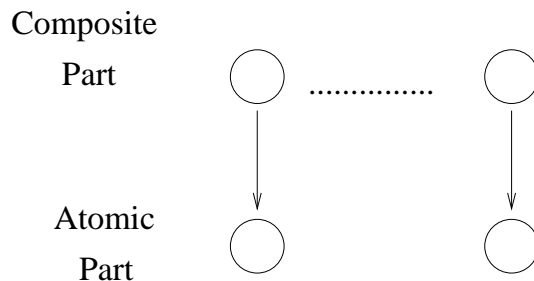


Figure 5.1: Partial declustering query, it involves 5000 composite parts with 1 atomic part per composite part.

5.2 A case for partial declustering

The goal of object placement in parsets is no different to the general problem of object placement in parallel OODBMS. Namely the internode objects references need to be decreased and the optimal degree of parallelism is dependent on the average workload performed on the parset. In fact one of the main motivating factors behind our investigation of partial declustering is the possibility of encountering small parsets which may also have small average workloads. In such cases the optimal degree of declustering is often smaller than the total number of processors in the parallel computer.

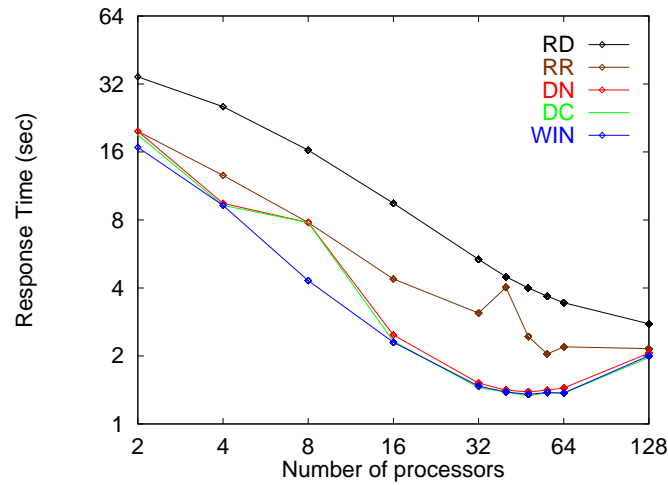
Figure 5.1 depicts a query using the objects of the OO7 benchmark. The query required a total of 10000 objects to be processed. These two queries were run on the AP1000 with the number of nodes varied from 2 to 128. The resulting response time graph is depicted on Figure 5.2.

The first observation that can be made is that an optimal degree of declustering does exist. Secondly it can be observed the optimal degree of declustering occurs at around the same place for DN, DC and WIN. More tests need to be carried out but from this graph it appears the optimal degree of declustering maybe independent of the object placement method employed.

5.3 Automatic placement of parsets

It is in general difficult for the user to manually specify the optimal degree of declustering. The reason is that the degree of declustering is dependent on the speed of the processors, IO and communication of the parallel computer on which the parset is declustered over. An automated placement scheme can however use a formula such as the one described in section 5.4 to determine the optimal degree of declustering.

After determining the optimal degree of declustering, the objects needs to be assigned to particular nodes. This is termed the assignment problem. The ideal assignment of an object residing in a particular parset is dependent on the placement of other



(a) smaller query

Figure 5.2: Graph depicting existence of optimal degree of declustering

objects which interact with it. For the purposes of this chapter an object o_i is defined to interact with an object o_j , if o_i references o_j or o_j references o_i . This interaction can be very complex. Many objects may reference the objects in the same parset and vice versa. In addition the frequency of reference is also important. It would be unrealistic for the user to consider all these factors when manually placing objects in a parset.

Figure 5.3 is an example of a situation which is difficult to optimise via manual object placement. Part a of Figure 5.3 describes the optimal state of placement of parset 1 before the creation of parset 2. When parset 2 (a subset of objects of parset 1) is created, the user may specify round robin declustering on parset 2 (in an attempt to parallelise parset 2). This causes the placement in 5.3 (b) to result. The placement depicted in 5.3 (b) is no longer optimal, since parset 2 is no longer perfectly load balanced. However an optimal solution does exist and is depicted in 5.3 (c). Therefore for optimal placement, the parsets can not be considered in isolation. It is however unreasonable for the user to manually place objects with consideration for more than one parset. Therefore automated placement is essential for the optimal placement of objects in this situation.

Figure 5.4 depicts another difficult situation to optimise via manual placement. In this situation, a new object is added to the database. At this point the user can reasonably make two choices and request the system to perform them. The first is to cluster the objects that O_1 references into one node as shown on Figure 5.4 (b) and the second is to place O_1 in the node which results in the least amount of internode

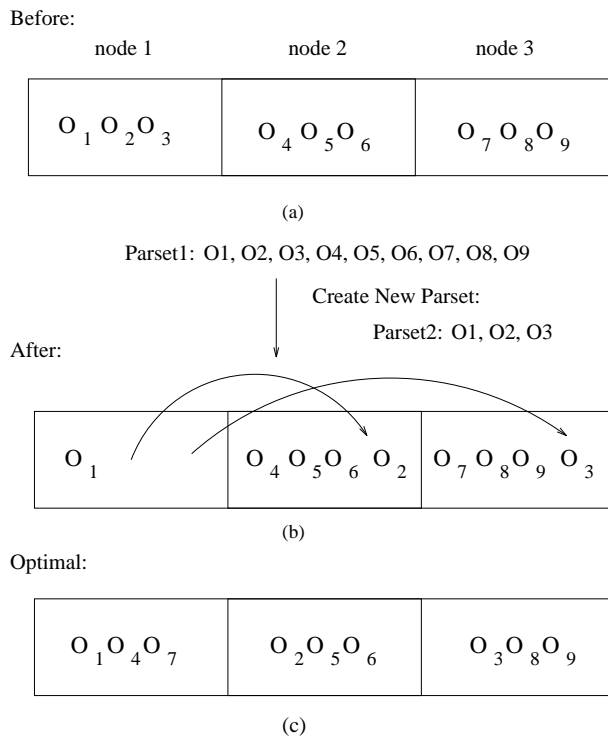


Figure 5.3: First illustration of Difficulties with manual placement

references as shown on Figure 5.4 (c)¹. However, note in case 1, parset 1 is no longer perfectly load balanced. In case 2, there is a internode reference from object 1 residing on node 1 to object 6 residing on node 3. Therefore both cases result in a non-optimal placement. Note however a optimal placement does exist and is depicted on Figure 5.4 (d). These are only two of many possible situations which are difficult to optimise via manual object placement. The reason is that object placement of a particular object can not be considered in isolation to the requirements of other parsets that this object interacts with.

5.4 Determination of Degree of Declustering

As mentioned in Section 5.3, the degree of declustering is an important parameter in determining the optimal placement of objects in a parallel OODBMS. In this section we outline a method developed in this thesis for determining the optimal degree of declustering in a parallel OODBMS in the context of parsets. The method outlined is an adaptation of the MDPS strategy developed by Ghandeharizadeh [1990] for rela-

¹Note here a tie occurs between node 1 and node 3. For the situation depicted in Figure 5.4 (c), the tie is broken arbitrarily by placing O_1 in node 1.

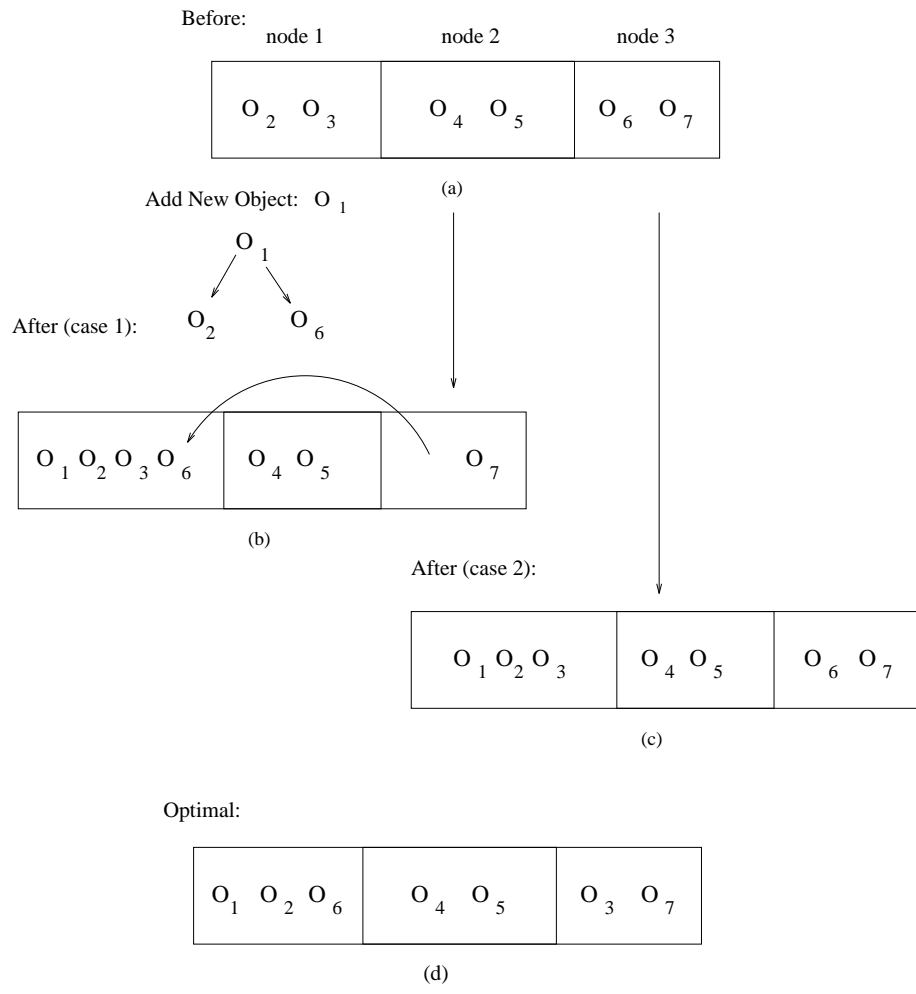


Figure 5.4: Second illustration of difficulties with manual placement

tional databases. Please see Section 2.3.2 for a brief description of MDPS. The main idea behind this method is the optimal degree of declustering is dependent on the workload size.

The average response time of running queries is depicted as a function of the number of processors (M) and the parset (P) being considered. This is depicted below:

$$RT_{avg}(M, P) = \frac{CPU_{avg}(P) + Disk_{avg}(P)}{M} + M \times CP(|P|) \quad (5.1)$$

Where the terms $CPU_{avg}(P)$ and $Disk_{avg}(P)$ are determined using the following equations:

$$CPU_{avg}(P) = WL_{avg}(P) \times CPU_{obj} \quad (5.2)$$

$$Disk_{avg}(P) = WL_{avg}(P) \times Disk_{obj}(|P|) \quad (5.3)$$

Where $Disk_{obj}(|P|)$:

$$Disk_{obj}(|P|) = \frac{Disk_{page}}{Obj_{page}} \quad (5.4)$$

$Disk_{obj}$ is the average time that it takes to retrieve a desired object from disk into memory. $|P|$ is the number of objects in parset P . $Disk_{page}$ is the time that it take to load a page from disk to memory. Obj_{page} is the average number of objects in a page. The first fraction of the equation determines the time that it takes to load an object from disk to memory if all the objects in the page are used by the query. Note this is only true if the objects in the parallel OODBMS are well clustered for parsets.

CPU_{obj} is a constant particular to the parallel computer being used. CPU_{obj} is the average time that it takes for the computer to process one object. This can be well approximated if the amount of computation done per object does not vary vastly between objects. Therefore if methods of vastly varying complexity are invoked on the objects, this equation may no longer be valid.

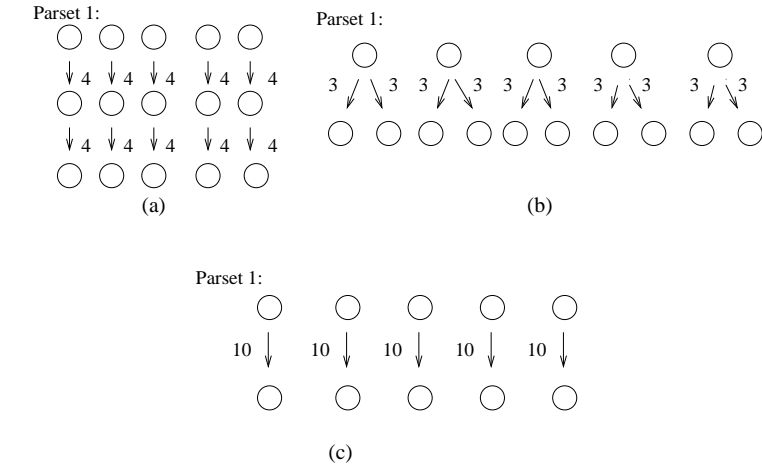
$WL_{avg}(P)$ is the average workload performed for the parset P and is determined as follows:

$$WL_{avg}(P) = \frac{\sum_{i=1}^n freq(P, i) \times num(P, i)}{\sum_{i=1}^n nfreq(P, i)} \times |P| \quad (5.5)$$

$freq(P, i)$ is the frequency of traversal of queries of kind i for parset P . Queries which access the same number of objects from the same parset are grouped into the same kind. $num(P, i)$ is the number of objects that are accessed by kind i of parset P . Note two queries need not have the same traversal structure but only the same number of objects traversed to be classified as the same kind. Figure 5.5 illustrates this concept diagrammatically. Figure 5.5 (a) and (b) both belong to the same kind.

$CP(|P|)$ is the overhead incurred when an extra node is used to process parset P . This overhead includes the startup and termination cost associated with adding an extra node.

When the first derivative of Equation 5.1 is computed with respect to M and equated to zero the following equation is derived:



$$\begin{aligned}
 WL_{avg}(\text{Parset 1}) &= \frac{(4 + 3) \times 3 + 10 \times 2}{4 + 3 + 10} \times 5 \\
 &= \frac{205}{17}
 \end{aligned}$$

Figure 5.5: Illustration of the computation of $WL_{avg}(P)$

$$M = \sqrt{\frac{CPU_{avg}(P) + Disk_{avg}(P)}{CP(|P|)}} \tag{5.6}$$

From Equation 5.6 we are able to compute the optimal number of processors to decluster the parset P across. Once the value of M is determined, the assignment of the parset objects to nodes need to be done. The assignment process is however not covered in this thesis mainly due to time constraints.

5.5 Summary

In this chapter we took a look at the issues arising from object placement in parsets. A method was outlined for determining the optimal degree of declustering for a parset. It is an adaptation of a similar method used in parallel relational databases called MDPS [Ghandeharizadeh 1990]. Unfortunately due to time constraints the method developed was not able to be tested.

Conclusion

In this chapter we summarise the findings made during the course of the project. We then discuss the merits of the two phased approach to object placement developed in this thesis. Finally we describe possible directions for further work.

6.1 Summary of Findings

The objective of this project was to find the object placement which minimises the total execution time of an arbitrary set of queries, for a shared nothing parallel OODBMS. To achieve this objective we developed a new similarity based two phased approach to object placement. It comprised of two phases: declustering, and clustering. The first phase was an adaptation of the similarity based declustering approach proposed by Liu and Shekhar [1996], for the different problem of data declustering in an uniprocessor multiple disk environment. The second phase was identified by this thesis to be a necessary second step to object placement in a shared nothing parallel systems using intra-operator parallelism.

The declustering phase was found to fail for the atomic part graph traversal of the OO7 benchmark. After examination, the problem was found to lay in the greedy manner in which objects are considered for placement. An improved heuristic was developed in which objects are considered in order of reachability. The improved heuristic was developed from the window approach to declustering used in Liu and Shekhar [1996]. This improved heuristic was found to perform well for the atomic part graph traversal (the problem it was developed to solve). However during subsequent experiments this improved heuristic was found to outperform all other placement algorithms when ever navigation was present in the query. From the findings made in this thesis it may be concluded if the length of navigation is long considering placement in order of reachability is essential.

In general the type of query executed played a larger part in determining the scalability of query processing when compare to the placement method employed. However when the query contains a large percentage of navigation as is the case in the forth query examined in Chapter 4, the scalability of the various placement methods varied significantly.

Set queries where found to be insensitive to the particular placement algorithm

employed. Whereas navigational queries showed varying performance depending on the object placement algorithm employed. This is due to the fact set queries handicap the placement algorithms proposed in this thesis by removing the requirement to reduce the number of internode object references and the number of remote page loads. Whereas navigational queries do require these measures of placement quality to be reduced.

When the training and execution conditions were varied in respect to each other, the clustering phase was found to be important in keeping response time low as training conditions became increasingly non-favourable.

In conclusion both the declustering and clustering phases were found to be important in reducing the number of remote page loads for navigational queries. For navigational queries the performance of query execution can be significantly improved by adopting a similarity-based approach to placement, such as the two phase approach developed in this thesis. For the set of queries investigated in Chapter 4 the two phased approach produced a large reduction in the number of remote page loads while preserving a high degree of parallelism.

Finally a way of determining the optimal degree of declustering for small sets of objects was presented in Section 5.4.

6.2 Further Work

In this thesis we explored a two phased approach to *offline* object placement. Therefore objects are placed when the database is not in operation. However this limits the application of the algorithm to situations where 24 hour access to the database is not required. This limitation can be overcome if online object placement is employed. Therefore an interesting direction for further work would be to investigate how the two phased approach can be applied to online object placement.

There is always room for improvement for the algorithms proposed in this thesis. Simple modifications such as changing the order in which objects are considered for declustering can produce significant improvement in performance as shown in Section 3.2.3

Peculiar Result Explanation

Continued

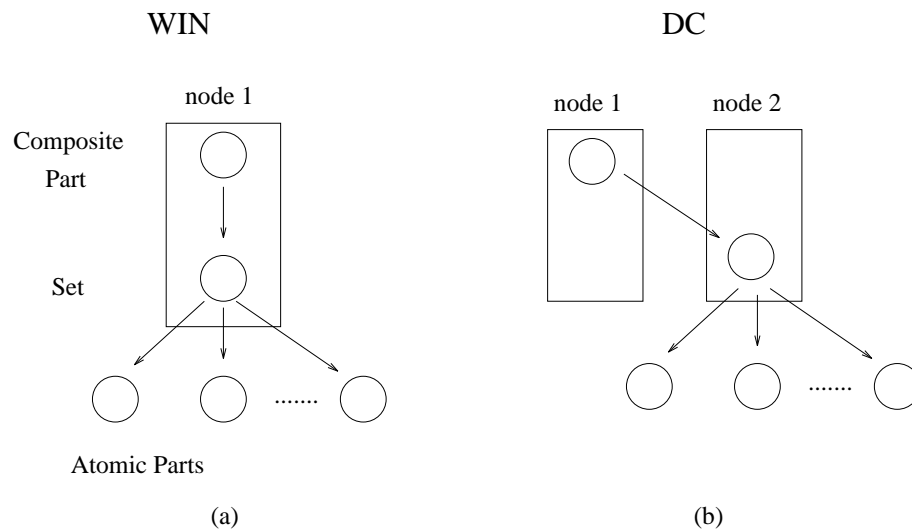


Figure A.1: Diagram to aid the explanation of the peculiar result observed for query 4

The purpose of this appendix is to explain the behaviour of the DC and WIN placement algorithms as depicted in Figure A.1, which is a reproduction of Figure 4.8. To understand how set objects are misplaced for DC but not for WIN, the traversal pattern of both query 3 and query 4 needs to be studied together. Figure A.2 depicts the objects traversed by query 3 and query 4, how they are related. For the scalability experiment described in Section 4.4, set 1 and set 2 objects are referenced more often than composite part objects. Therefore during the declustering phase of DC (greedy approach), the set 1 and set 2 objects are placed before composite part objects. This causes the declustering algorithm to often place set 1 and set 2 objects (which indirectly reference each other) into different nodes. Hence when it is time to place composite part objects, a tie situation often develops. The declustering algorithm does not

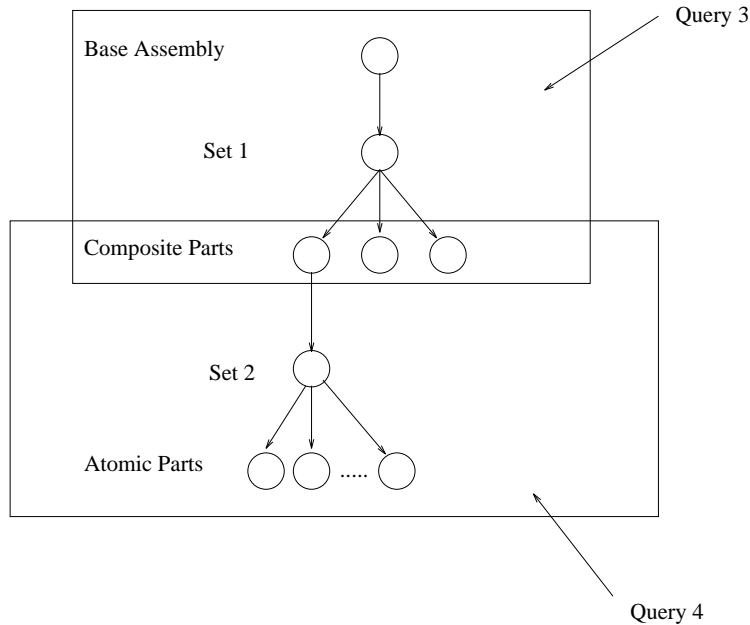
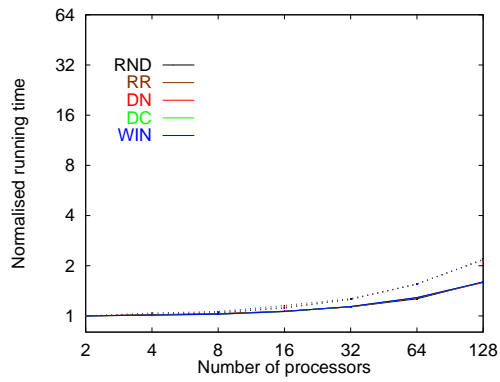


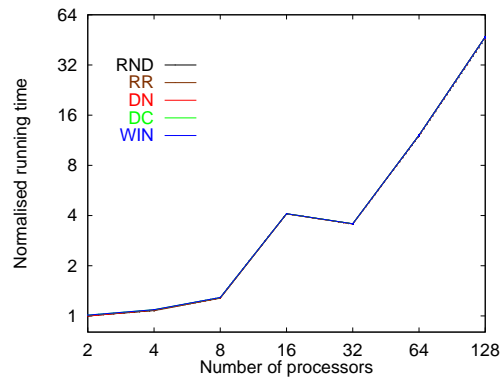
Figure A.2: Objects traversed by both query 3 and query 4

know whether to place the composite part object with the set 1 object or set 2 object. If the set 1 object is chosen then the situation depicted in Figure A.1 (b) results. In contrast WIN considers objects for placement in order of reachability. Therefore set 1, composite part, and set 2 objects tend to be placed in the same node. The reason being the declustering algorithm never considers set 1 and set 2 objects before composite part objects are considered. This explains the placement behaviour of WIN depicted in figure A.1 (a).

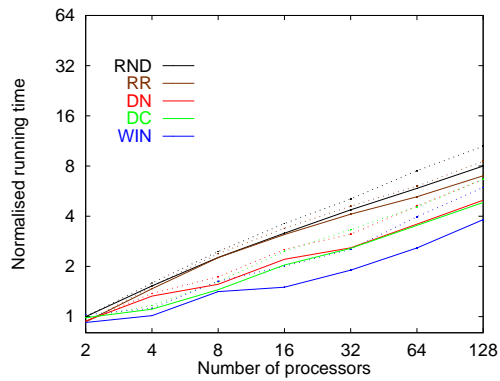
Scalability Comparison via Response Time and Average Time



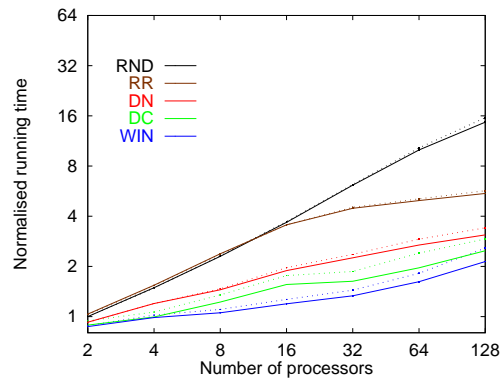
(a) Query 1



(b) Query 2



(c) Query 3

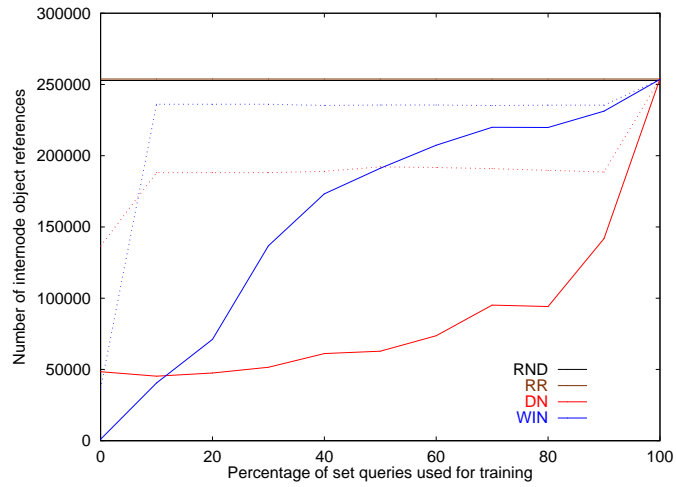


(d) Query 4

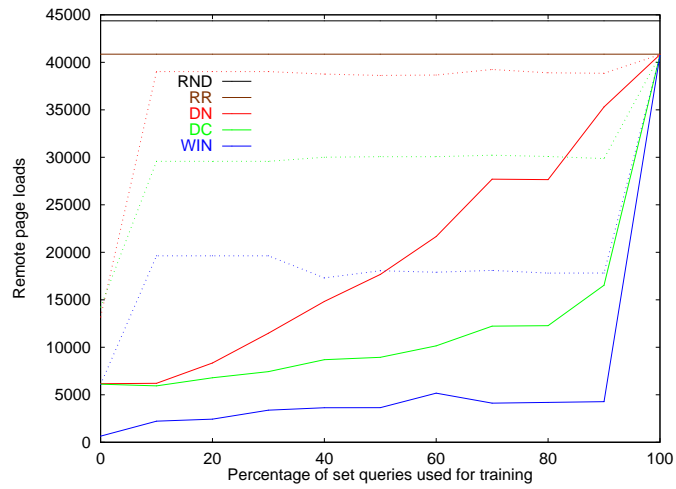
Figure B.1: Scalability comparison

Varying Training Conditions Results Continued

These are some figures which were omitted in the varying training condition experiment (Section 4.5). These graphs show the number of internode object references and remote page loads for the 90% set (10% navigational) queries case.



(a) Number of internode object references



(b) Number of remote page loads

Figure C.1: Running with 90% set queries, 10% navigational queries

OO7 schema

The following is a description of the OO7 benchmark schema as described in Carey, DeWitt, and Naughton [1993].

The description uses the C++ based data definition languages (DDLs). Inter-object references are denoted using C++ pointers notation, eg., the **partOf** field of each atomic part object is a reference to the composite part objects that the atomic part is a part of, and its type indicated as **CompositePart***. The expressions **Set(T*)** and **Bag(T*)** denote sets and bags (multisets) of references to objects of type T. For example, the **to** field of an atomic part object is a set containing references to connection objects that hold data about (outgoing) connection between the atomic part and other atomic parts. In addition, inverse relationships are denoted in the schema using **< - >**, pronounced “inverse of”. The fact that an atomic part has an inverse relationship with the connection objects that it is connected to is captured in the DDL description of the atomic part class via: **Set(Connection*) to < - > Connection::from**. This says that the field **to** of an atomic part is inversely related to the **from** field of the connections that it references. Finally, it should be noted that OO7 schema includes instances of 1:1, 1:N, and M:N relationships; these are modelled as inverse relationships between a pair of reference fields, and between two bags of references, respectively.

```
class DesignObj{
    int4    id;
    char    type[10];
    int4    buildDate;
};

class AtomicPart: DesignObj{
    int4          x, y;
    int4          docId;
    Set(Connection*)    to <-> Connection::from;
    Set(Connection*)    from <-> Connection::to;
    CompositePart*      partOf <-> CompositePart::parts;
}

class Connection {
```

```

char                type[10];
int4                length;
AtomicPart*        from <-> AtomicPart::to;
AtomicPart*        to <-> AtomicPart::from;
};

class CompositePart: DesignObj {
  Document*         documentation <-> Document::part;
  Bag(BaseAssembly*) usedInPriv
                    <-> BaseAssembly::componentsPriv;
  Set(AtomicPart*) parts<-> AtomicPart:: partOf;
  AtomicPart*      rootPart;
}

class Document {
  char                title;
  int4                id;
  String              text;
  CompositePart*     part <-> CompositePart::documentation;
}

class Manual {
  char                title[40];
  int4                id;
  String              text;
  int4                textLen;
  Module*             mode <-> Module::man
};

class Assembly: DesignObj {
  ComplexAssembly*   superAssembly
                    <-> ComplexAssembly::subAssemblies;
  Module*            module <-> Module::assemblies;
};

class ComplexAssembly: Assembly {
  Set(Assembly*)     subAssemblies
                    <-> Assembly::superAssembly;
}

class BaseAssembly: Assembly {
  Bag(CompositePart*) componentsPriv
                    <-> CompositePart::usedInPriv;
}

```

```
class Module: DesignObj {
  Manual*          man <-> Manual::mod;
  Set(Assembly*)  assemblies <-> Assembly::module;
  ComplexAssembly* designRoot;
} with extent (id indexed);
```

Bibliography

- BANCILHON, F., BRIGGS, G., KHOSHAFIAN, S., AND VALDURIEZ, P. 1987. A powerful and simple database language. In *Proceedings of the VLDB Conference* (Brighton, UK, 1987). (p.47)
- CAREY, M., DEWITT, D., AND NAUGHTON, J. 1993. The OO7 benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*. (pp.18, 23, 25, 28, 63)
- CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O., WHITE, S., AND ZWILLING, M. J. 1994. Shoring up persistent applications. In *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data* (Minneapolis, Minn, May 1994). (p.47)
- COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. 1988. Data placement in bubba. In *Proceedings of the 21st VLDB Conference* (1988), pp. 99–108. (p.2)
- DEWITT, D. J., NAUGHTON, J. F., SHAFER, J. C., AND VENKATARAMAN, S. 1996. Parallelizing OODBMS traversals: a performance evaluation. *The VLDB Journal* 5, 3, 3–18. (pp.2, 7, 47)
- FANG, M., LEE, R., AND CHANG, C. 1986. The idea of declustering and its applications. In *Proceedings of twelfth International conference on Very Large Databases*, 181–188. (p.11)
- GERLHOF, C., KEMPER, A., KILGER, C., AND MOERKOTTE, G. 1993. Partition-based clustering in object bases: From theory to practice. In *Proceedings of the International Conference on Foundations of Data Organisation and Algorithms (FODO)*. (pp.9, 12)
- GHANDEHARIZADEH, S. 1990. *Physical Database Design in Multiprocessor Database Systems*. PhD thesis, University of Wisconsin. (pp.2, 8, 50, 53)
- GHANDEHARIZADEH, S., WILHITE, D., LIN, K., AND ZHAO, X. 1994. Object placement in parallel object-oriented database systems. *Proceedings of the Tenth International Conference on Data Engineering*, 253–262. (p.3)
- GOETZ, G. 1993. Survey of query evaluation techniques for large databases. *ACM Computing Surveys* 15(2), 75–170. (p.6)
- IMASAKI, K., ONO, T., HORIBUCHI, K., MAKINOCHI, A., AND AMANO, H. 1997. Design and evaluation of the mechanism for object references in a parallel object-

- oriented database system. In *Proceedings of International Database Engineering and Application Symposium* (August 1997). (p.2)
- ISHIHATA, H., HORIE, T., INANO, S., SHIMIZU, T., AND KATO, S. 1990. CAP-II architecture. In *Proceedings of the First Fujitsu-ANU CAP Workshop* (Kawasaki, Japan, November 1990). Fujitsu Laboratories Ltd. (p.21)
- KERNIGHAN, B. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 291–307. (p.9)
- KILIAN, F. M. 1992. *Parallel Sets: an object-oriented methodology for massively parallel programming*. PhD thesis, Harvard Center for Research in Computing Technology, Cambridge, Mass. (p.47)
- LIU, D. AND SHEKHAR, S. 1996. Partitioning similarity graphs: A framework for declustering problems. *Information Systems* 21, 6 (September), 475–496. (pp.3, 11, 12, 19, 20, 29, 55)
- MEHTA, M. AND DEWITT, D. J. 1995. Managing intra-operator parallelism in parallel database systems. In *Proceedings of the 21st VLDB Conference* (1995), pp. 382–394. (p.2)
- PADMANADHAN, S. AND BARU, C. K. 92. Data placement in shared-nothing parallel database systems. Technical report, The University of Michigan.
- SITSKY, D., WALSH, D., AND JOHNSON, C. 1994. An efficient implementation of the message passing interface (MPI) on the Fujitsu AP1000. In *Proceedings of the Third Parallel Computing Workshop* (Kawasaki, Japan, 25–26 1994). Fujitsu Laboratories Ltd. (pp.22, 23)
- THAKORE, A. K. AND SU, S. Y. 1994. Performance analysis of parallel object-oriented query processing algorithms. *Distributed and Parallel Databases* 2, 1, 59–100. (p.2)
- TSANGARIS, M. M. AND NAUGHTON, J. F. 1991. A stochastic approach for clustering in object bases. In *Proceedings of the ACM SIGMOD conference on Management of Data* (1991), pp. 12–21. (p.3)
- ZDONIK, S. B. AND MAIER, D. 1990. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann. (p.1)