Dynamic Database Object Horizontal Fragmentation

C.I. Ezeife*
School of Computer Science
University of Windsor
cezeife@cs.uwindsor.ca
http://www.cs.uwindsor.ca/users/c/cezeife

Jian Zheng
School of Computer Science
University of Windsor

Abstract

Applying an object based horizontal fragmentation scheme to an object oriented database system, creates subsets of class extents which are allocated to sites where they are most needed. As new class instances are created or database schema evolves or application queries access patterns change at distributed sites, system performance may drop. Restoring the system performance requires re-running the object horizontal fragmentation scheme after conducting a full static system requirements analysis. This paper proposes an object horizontal distributed design architecture which determines system performance threshold, monitors changes in distributed design input which may affect performance and dynamically triggers a re-fragmentation of the system if performance drops below the system performance threshold.

Keywords: Object-oriented databases, Horizontal fragmentation, Distribution, Performance analysis

1 Introduction

Horizontally fragmenting database classes and placing class fragments at distributed sites has the advantage of reducing the amount of irrelevant data accessed locally by applications as well as reducing the amount of remote access made to data by applications running at other sites. Other advantages of data fragmentation include increased throughput since more than one application can run on fragments of a class placed at distributed sites concurrently. Fragmentation also reduces data transmission costs by transmitting only part of a class and not the whole class. Recent works on fragmentation of database relations and classes include [3, 4, 8]. All of these earlier distributed database design works are static designs because input to the design process, which include queries accessing database and their frequencies as well as the database schema, are obtained from an earlier full requirements analysis. This implies that when the system undergoes sufficient changes, a new requirements study is carried out to re-run the fragmentation and allocation schemes. In order to make these systems more acceptable by users, a mechanism for determining what change is considered sufficient to trigger a re-fragmentation is needed.

This paper adopts the technique of setting a distributed system performance threshold as a factor of the measured system performance immediately after fragmentation and allocation and a system determined variable which measures the average change in system input over time.

*This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

The paper proposes an architecture that includes a monitor component which keeps track of changes in the system and periodically re-computes the system performance values using the new input states. A re-fragmentation is triggered if the performance values fall below the threshold. The component used to evaluate the performance of the distributed system is the object horizontal partition evaluator presented in [5], which extends the initial work presented in [2] for vertical fragmentation of relations. Chakravarthy *et al.* in [2] presents an objective function for measuring the performance of vertical fragments of relations. [5] extends this work to measure object horizontal fragmentation including features of encapsulation, inheritance, class composition and method nesting hierarchies.

1.1 Data Model

Object-oriented databases represent data entities as objects supporting features of encapsulation. Thus database objects have a set of attributes ($\mathcal{A}$) and a set of methods ($\mathcal{M}$) for accessing these attributes. Objects with common attributes and methods belong to the same class C with a class identifier K. Every object has an object identifier corresponding to its class identifier prefixed to a unique object number. All objects of a class form its instance objects ($\mathcal{I}$) or class extents. Thus, we can represent a class as an ordered relation C = (K,$\mathcal{A}$,$\mathcal{M}$,$\mathcal{I}$). For example, a class *Person* has class identifier Person. Its attributes are social security number, name, age. Its methods are getname, getage etc. The instances of the class have names, John, Mary etc. Object-oriented database systems also support inheritance whereby a class B called subclass of class A can be defined as a specialization of class A. Class A is called the superclass of class B and class B can inherit all attributes and methods of its superclass A but not all of its extents. Thus if class B is a subclass of class A, A is represented as $C^A = (KA, A^A, M^A, I^A)$ while B can be represented as $C^B = (KB, A^A \cup A^B, M^A \cup M^B, I^B \odot I^A)$ where the symbol $\odot$ stands for object pointer, meaning that every $I \in I^B$ is connected to some $I \in I^A$. For example, in a database with class *Person* and *Student*, *Student* is a subclass of class *Person*. Object-oriented database systems also support class composition and method nesting hierarchies. Each horizontal fragment (C_h) of a class contains all attributes and methods of the class but only some instance objects ($I' \subseteq I$) [4]. Vertical fragments of a class contain subsets of attributes and methods but all instance objects of the class, while hybrid fragments contain subsets of both attributes, methods and instances of the class.

1.2 Related Work

In horizontally fragmenting relations, primary fragmentation is performed on all owner relations while derived fragmentation is performed on all member relations of links according to a selection operation specified on its owner relation. Ozsu and Valduriez [8] give a formal definition of simple predicates as $P_j : A_i \ \theta \ Value$ (e.g. Studentgpa $\geq$ 4.0) and that given a set of simple predicates $P_{ri} = \{P_{i1}, P_{i2}, ..., p_{im}\}$, the set of minterm predicates $M_i = \{m_{i1}, m_{i2},...,m_{iz}\}$ is defined as $M_i = \{m_{ij} | m_{ij} = \wedge P^*_{ik}\}$, $P_{ik} \in P_i$, $1 \leq k \leq m$, $1 \leq j \leq z\}$ where $P^*_{ik} = P_{ik}$ or $P^*_{ik} = \neg P_{ik}$. For example, with P_1 as studentgpa $\geq$ 4.0 and P_2 as major ="Math", the defined minterms are $m_1 = P_1 \wedge P_2$, $m_2 = P_1 \wedge \neg P_2$, $m_3 = \neg P_1 \wedge P_2$, $m_4 = \neg P_1 \wedge \neg P_2$. They also propose an iterative algorithm (COM_MIN) that basically eliminates minterm predicates which are either redundant or meaningless.

Object-oriented database systems that support some form of distribution or based on client/server model include GOBLIN [6], THOR [7] and for improved performance of such systems, techniques for proper fragmentation of objects are needed. Thus, Ezeife and Barker in [4] present a set of horizontal fragmentation algorithms (called object oriented horizontal fragmentation scheme OOHF), for four types of class models including those with both simple attributes and methods, simple attributes and complex methods, complex attributes and simple methods, and complex attributes and methods. Since the model with complex attributes and methods is the most encompassing, this algorithm is summarized next. The algorithm defines primary horizontal fragments of the class using predicates from queries accessing this class directly. Then, derived fragments of this class are defined using all primary fragments of all its owner classes from all link graphs to accommodate inheritance, class composition hierarchies and possibly method nesting hierarchies. Finally, the sets of primary and derived fragments of the class are combined following the rule that every derived fragment should be merged with a primary fragment it has the highest affinity count with. However, if a class has no primary fragments, its derived fragments become its final fragments ensuring all instance objects are placed in some fragment. Other object horizontal fragmentation schemes include the class dependence graph techniques in [9] and an approach by Bellatreche, Karlapalem and Simonet in [1] which defines primary horizontal fragmentation on simple queries while derived horizontal fragmentation is defined on component queries. Chakravarthy *et al.* in [2] presents an objective function called Partition Evaluator (PE) for evaluating the goodness of any vertical partitioning algorithm. The PE is used to measure the total costs of processing all applications at all distributed sites if each access to a data fragment amounts to a unit cost. It is assumed that there are no data redundancies and following fragmentation, the fragments have been allocated to sites where they are most needed. The two components of PE are irrelevant local method access cost which minimizes the square error for a fixed number of fragments and assigns a penalty factor whenever irrelevant attributes are accessed in a particular fragment. Ezeife and Barker extend the relational vertical Partition Evaluator (PE) to handle measuring the costs of local and remote accesses incurred by the object vertical fragments. Ezeife and Zheng in [5] presents an object horizontal partition evaluator algorithm which computes the processing cost of horizontally distributed objects by first computing the application instance object sets. Then, the performance of each class of the object base is measured by its partition evaluator value (PE). The PE of a class is the sum of its local irrelevant access cost (E_M^2) and its remote relevant access cost (E_R^2). To accommodate features of encapsulation, inheritance, class composition and method nesting hierarchies, access to a class, includes all queries accessing the class instances directly, those accessing it indirectly through its descendant classes, containing classes and complex method classes.

1.3 Contributions

This paper contributes by proposing an architecture for dynamic object horizontal fragmentation. Components of the dynamic object horizontal distribution design include: (1) a mechanism for determining the performance threshold of an object horizontal distributed system which is based on the partition evaluator measure. (2) a monitor algorithm for keeping track of changes in the system. This component eliminates the need for a full static requirements analysis and provides input for measuring system performance in order to determine when a re-fragmentation should be initiated. Dynamic fragmentation saves processing cost by eliminating the cost of full requirements study before every re-fragmentation. It also improves on overall system performance because it is able to detect early when the system performance drops below acceptable threshold. Furthermore, user confidence in the system is increased since the user is more reassured to know there is a mechanism in place for detecting and correcting low system performance.

1.4 Outline of the Paper

Section 2 presents the architecture of the dynamic object horizontal distributed database design and the algorithm for determining performance threshold (OOThreshold algorithm). Section 3 presents the distributed object-oriented monitor algorithm (DOBS-Monitor algorithm). Section 4 presents an example while section 5 presents conclusions.

2 Dynamic Distributed Objectbase Horizontal Design Architecture

This section presents the architecture of the dynamic object horizontal distributed database design (section 2.1), including the algorithm for determining the system performance threshold (section 2.2).

2.1 Dynamic Distributed Objectbase Architecture and Components

Figure 1 shows the architecture of the dynamic distributed object horizontal design. At the onset of the design, the input of application queries (AQ) and the sites where they run, query access frequencies (AF), class inheritance hierarchy (CH), class composition hierarchy (CCH), method nesting hierarchies (MNH) and the objectbase class instances (IS) are fed to an object horizontal fragmentation scheme like OOHF in [4] or Bell's in [1].

The effect is that horizontal fragments of objects is defined and allocated to distributed sites. The instance object fragments allocated to distributed sites, and other inputs to distribution design (AQ, AF, CH, CCH and MNH) now constitute input to the object horizontal partition evaluator algorithm [5]. The object horizontal partition evaluator algorithm immediately following a fragmentation and allocation, computes the best system partition evaluator using the input data. Next, the dynamic architecture system computes the system performance threshold using the computed best partition evaluator and system change variable (S). The system change variable measures or represents how frequently the system's input set that may affect performance change. The S is computed as the inverse of the average percentage change in system input, multiplied by 10 to allow a reasonable margin of degrading performance. The inverse is used because a frequently changing system would require more frequent re-fragmentation and thus a lower S value since a lower S value means a threshold that is closer to the best PE value. The lower the threshold, the more likely it is that the system computed PE value at any other check time is higher than threshold and would call for a re-fragmentation. The implication is that if a system's set of input change frequently, then its average change is very high making its inverse very low and close to zero. Then, the system threshold which is computed as the addition of the best PE following a fragmentation and its product with the system change variable S (that is (threshold = best PE + (best PE * S))) is close to best PE value. The margin of lower performance allowed by the system before calling for a re-fragmentation is the fraction of the best PE determined by S. This margin is closer to best PE if S is closer to 0 and S is closer to 0 as the system average input change increases. The more frequently a system changes, the lower its S value; while a high S value represents a slowly changing system with a low average percentage change in its input. Note that S has a positive value and a higher PE value represents a lower performance by the system.

The dynamic architecture also includes a Monitor algorithm which keeps track of all changes in the input data and runs periodically to determine if the change is big enough to call for a re-evaluation of the system performance. A re-evaluation of the system performance is

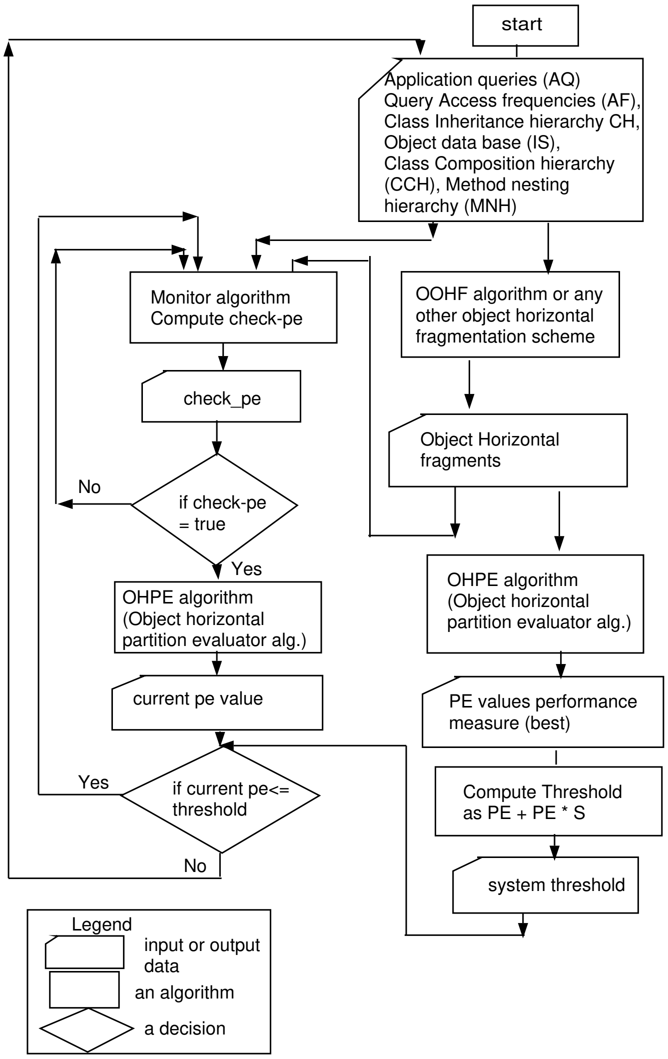

Figure 1: Dynamic Distributed object horizontal database design architecture

accomplished by computing the system's current PE value using the object horizontal partition evaluator algorithm. It is only when the current PE value of the system is computed that the current PE value is compared with performance threshold value. A current PE value bigger than the performance threshold value calls for the system to be re-fragmented using the new input data from the monitor. The monitor keeps track of the system average change for each input type over time and compares the new change in this input type with the average change. Its output is a value (check-pe) to recommend whether to run the object horizontal partition evaluator for evaluating system performance or not. It also outputs the new system average change for each input data as the average of the old and new changes. In Figure 1, the algorithms are shown in rectangles while the solid input and output data are shown in left-corner trimmed rectangles. In addition to the overall system architecture, the algorithms, OOMonitor and Compute Threshold are contributions of this paper.

2.2 The Algorithm Compute Threshold

Determining the system performance threshold is necessary if changes in the input that would drop system performance below acceptable limits are to be detected. Thus, the system change variable S is used to determine the performance threshold value of the system. We propose a method for computing the threshold value as the sum of the best system PE obtained immediately after a re-fragmentation and a fraction of this best PE value determined by S. This means that threshold is computed as best PE + (best PE * S). The value of S is determined as the inverse of the average change in the system input data. Since it is not easy deciding how to measure changes in such system inputs as class inheritance, class composition hierarchies, we define a technique for obtaining the changes in the input data in section 3. This computes the change in an input data like access frequency as a percentage of the original access frequency, while creation, destruction or re-location of a query each counts as a change measured against the total number of queries in the system. More detailed and formal presentations of these definitions are discussed later since it is the monitor program that is responsible for computing those changes.

The average changes are gathered by the monitor algorithm and passed in as input to the compute threshold algorithm. Once the input changes are computed, the S variable is simply computed as the 10 divided by the average of the percentage changes in all of these input data. For example, in system with a 25% change in class inheritance hierarchy, 35% change in class composition hierarchies, 10% change in application queries, 20% change in application frequencies, no change in method nesting hierarchies and 10% change in object instance fragments, the S value is computed as $10/((25 + 35 + 10 + 20 + 10)/5) = 0.5$. On the other hand, if the average percentage change is 50%, the S is computed as 10/50 or 0.2. This shows that a higher change produces lower S while lower percentage changes in system input produces higher S value. The threshold value is now determined as best PE value + (best PE value * S). This means that any input change less or equal to the average input change would not trigger a re-fragmentation. The formal presentation of the Compute Threshold algorithm is given as Figure 2.

Algorithm 2.1 *(Algorithm (OOThreshold- computes the system performance threshold))*

Algorithm OOThreshold
Input: (input changes: $\triangle AF$, $\triangle AQ$, $\triangle CH$, $\triangle CCH$, $\triangle MNH$, $\triangle IS$ fragments),
best PE
Output: Performance threshold
Begin
S = 10/(($\triangle AF + \triangle AQ + \triangle CH + \triangle CCH + \triangle MNH + \triangle IS$)/(number of inputs with change > 0))
Threshold = best PE + (best PE * S)
end // of oothreshold //

Figure 2: The Algorithm Compute Threshold

3 The Distributed Object-oriented Monitor Algorithm (OOMonitor)

The object-oriented monitor algorithm serves to compute the input changes used to determine the system change variable S by the Compute Threshold algorithm (OOThreshold). The monitor also determines whether the system performance should be checked or not. By determining the times it is necessary to re-compute the system current PE value, the overhead incurred doing the checks is reduced. The monitor would recommend that the system PE values be re-computed only if during any system input change check, it encounters an input change that is higher than the average system input change. It re-computes the average system input change only after a re-fragmentation. The other question is determining how frequently the monitor itself should run. We recommend that the monitor be run periodically, once a day for some applications but may be set to suit application requirements.

The input to the monitor include the previous AQ, AF, CH, CCH, MNH and IS fragments and their sites. Other input to the algorithm are current AQ, AF, CH, CCH, MNH and IS fragments. An additional input is periodic wait time which is used to determine how often the monitor should run. In order to compute the average changes in the input data, the monitor needs to remember the previous change in each input data so that it can compute the average change by finding the sum of the previous change and the current change. The monitor output includes a boolean variable which is either true or false where true means that the system PE value should be re-computed and false means that it should not. The other output data from the monitor are the average changes in the input data of AQ, AF, CH, CCH, MNH and IS fragments. These average input changes are used to compute the system change variable S by algorithm OOThreshold. A change in any input data of AQ, AF, CH, CCH, MNH and IS fragments is computed as the percentage of the original data. This means that in a class hierarchy with 8 classes, if a class gets added, is dropped or moves to another location in the hierarchy, the change is defined as 1/8*100 = 12.5%. If a total of two classes is either created, destroyed or relocates, the change is defined as 2/8 * 100 = 25% change. This definition applies to all hierarchies. In the case of object instance fragments, if a new object gets created in any site, it stays in that site until a re-fragmentation. To compute the change in object instance fragments, we count any new object that is created at any site as 1, any destruction or relocation of an object also counts as 1. The total change in the object instance fragments of a class is the sum of all changes at all sites divided by the number of instance objects in the class * 100. The change in application queries is defined as the percentage of new queries that access a class, stop accessing a class or change the site of access to a class. Thus, the change in application queries accessing a class is obtained as the sum of the number newly accessing it, or no longer accessing it divided by the total number of queries originally accessing the class multiplied by 100. The change in access frequency of queries is defined as the maximum of the percentage change in any one query at all sites. Thus, the change in access frequency of queries is computed as maximum of the total change at all sites for each query divided by total access at all sites for that query multiplied by 100. The formal presentation of the Monitor algorithm is given as Figure 3. The monitor computes change in each input data during each run as the average of the previous change and the new change.

4 An Example

This section shows an example object database system with its class inheritance hierarchy similar to the example in [4], which is first fragmented horizontally using the object horizontal fragmentation scheme OOHF in [4]. The sample object database represents a university database with classes *Person, Prof, Student, Grad, Undergrad and Dept.* Classes Prof and Students are subclasses of Person while classes Grad and Undergrad are subclasses of class Student. Both classes Person and Dept are subclasses of the Root class. This section shows how the performance threshold of the distributed system is measured and how a change in the access frequencies of queries accessing these classes at distributed sites is used by the monitor system proposed in this paper to trigger a re-fragmentation of the classes whose re-computed PE values are higher than the threshold PE values. Using the given query access frequency and other input data, the object horizontal fragmentation scheme (OOHF) produces the following fragments for the classes in this database, where the I's represent the classes' instance objects in each fragment.

Class Person

$$F_1^h = \{I_1, I_7\},\ F_2^h = \{I_3, I_4\},\ F_3^h = \{I_2, I_8\},\ F_4^h = \{I_{10}\},\ F_5^h = \{I_6, I_9\},\ F_6^h = \{I_5\},$$

Class Student

$$F_1^h = \{I_1, I_5\},\ F_2^h = \{I_2, I_4\},\ F_3^h = \{I_3, I_6\},$$

Class Prof

$$F_1^h = \{I_2, I_3, I_4\},\ F_2^h = \{I_1\},$$

Class Dept

$$F_1^h = \{I_2, I_3\},\ F_2^h = \{I_1\},\ F_3^h = \{I_4\},$$

Class Grad

$$F_1^h = \{I_1\},\ F_2^h = \{I_2, I_3\},$$

Algorithm 3.1 *(Algorithm (OOMonitor - computes when to re-evaluate the system))*

Algorithm OOMonitor
Input: Previous AQ, AF, CH, CCH, MNH, IS
Current AQ, AF, CH, CCH, MNH, IS
monitor_wait_time(length of time in seconds)
Previous system change averages ($\triangle AQ$, $\triangle AF$, $\triangle CH$, $\triangle CCH$, $\triangle MNH$, $\triangle IS$)
Output: check_current_PE (boolean)
check_current_PE (boolean)
new system change averages ($\triangle AQ$, $\triangle AF$, $\triangle CH$, $\triangle CCH$, $\triangle MNH$, $\triangle IS$)

```
begin
//Initialize variables //
check_current_pe = false
time = systemtime - lastruntime
Read all input data
While not(check_current_pe) and (time >= monitor_wait_time)
Begin
    Current change AQ = |currentAQ − previousAQ|
    Current change AF = |currentAF − previousAF|
    Current change CH = |currentCH − previousCH|
    Current change CCH = |currentCCH − previousCCH|
    Current change MNH = |currentMNH − previousMNH|
    Current change IS = |currentIS = previousIS|
    //Monitor first determines if the system pe should be recomputed//
    //before re-computing the new average system changes. //
    if current change AQ > previous change AQ then check_current_pe =true
    else if current change AF > previous change AF then check_current_pe =true
    else if current change CH > previous change CH then check_current_pe =true
    else if current change CCH>previous change CCH then check_current_pe =true
    else if current change MNH>previous change MNH then check_current_pe =true
    else if current change IS>previous change AQ then check_current_pe =true
    average △AQ = (current △AQ + previous △AQ)/2
    average △AF = (current △AF + previous △AF)/2
    average △CH = (current △CH + previous △CH)/2
    average △CCH = (current △CCH + previous △CCH)/2
    average △MNH = (current △MNH + previous △MNH)/2
    average △IS = (current △IS + previous △IS)/2
    lastruntime = systemtimenow
end // of while //
end //of monitor //
```

Figure 3: The Distributed Object-oriented Monitor Algorithm

Class	Site 1	Site 2	Site3
Person	F1={I1, I7} F5=(I6, I9} F4={I10}	F6={I5} F2={I3, I4}	F3={I2, I8}
Student	F1={I1, I5}	F2={I2, I4}	F3={I3, I6}
Prof	F1={I2, I3, I4}	F2={I1}	
Dept	F1={I2, I3}	F2={I1}	F3={I4}
Grad	F1={I1}	F2={I2, I3}	

Figure 4: Allocation of Fragments to Distributed Sites

	S1	S2	S3	Running directly on
q1	10	20	0	Grad
q2	20	30	15	Prof
q3	25	10	0	Prof
q4	15	25	10	Student
q5	20	20	20	Dept

Figure 5: Access Frequencies of queries at distributed sites

The fragments above are allocated to three distributed sites using the simple allocation scheme that places fragments at sites where they are most needed by queries, while Figure 4 shows the placement of above fragments to sites and Figure 5 shows the access frequencies of queries at the three sites.

The local irrelevant access costs, remote relevant access costs and the PE values obtained after executing the OHPE algorithm are shown in the table below:

Class	Local irrelevant	Remote relevant	PE value
Person	250	0	250
Student	250	0	250
Prof	0	0	0
Grad	0	0	0
Dept	0	0	0

Assume that overtime the change in the system change variable is determined by the OOThreshold algorithm to be 0.3, then the OOthreshold computes the system performance threshold as best PE of the class plus best PE of the class multiplied by S. This gives for classes *Person* and *Student* the threshold values of 333 and for the rest of the classes threshold values of 0. This is because the algorithm computes S as 10/30, and threshold as best PE + (S * best PE), which gives 325 + (10/30 * 325). Thus, if every other input to the system remains constant but there is a change in the access frequencies of the five queries to give the new access frequencies shown in Figure 6, then, the OOMonitor algorithm will compute the new change

	S1	S2	S3	Running directly on
q1	20	30	10	Grad
q2	25	35	20	Prof
q3	30	15	5	Prof
q4	25	35	20	Student
q5	25	25	25	Dept

Figure 6: The new Access Frequencies of queries at distributed sites

in query 1 as ((20-10)+(30-20)+(10-0))/30 * 100 which gives a percentage change in the access frequency of 100% or 1.0. Since this change is greater than the S value of 0.3, the Monitor algorithm will recommend that the system PE-value be re-computed. The system PE values are again re-computed with the OHPE algorithm of [5] using the new access frequencies and the results obtained are given below:

Class	Local irrelevant	Remote relevant	PE value
Person	650	0	650
Student	650	0	650
Prof	0	0	0
Grad	0	0	0
Dept	0	0	0

5 Conclusions

The importance of dynamic object fragmentation had been argued by many earlier works. The value of such a technique increases when it is horizontal fragmentation which is more commonly used in the industry. A dynamic horizontal fragmentation approach provides the means to detect when the distributed system performance degrades below acceptable limit. The advantage of such a design includes reduced processing cost by eliminating the need for full systems requirements analysis, user confidence because system is participating in alerting the user when there is low performance, and higher productivity.

A re-design of the distribution technique may be needed when due to changes in application access pattern or object database schema, system performance has dropped below an acceptable threshold. This paper contributes by first presenting a dynamic distribution design architecture and then, a set of algorithms that would determine the system performance threshold based on the computed partition evaluator value and the average rate of change of system input. The paper also uses the average system change variable and the threshold to determine when a re-design is necessary. A re-design is necessary when the computed current partition evaluator value of the class is higher than the performance threshold value. A re-design would entail running the object horizontal fragmentation scheme with the most current input data before allocation to distributed sites. Future work should investigate reducing the cost of re-design of the system by performing incremental re-fragmentation whereby only the changes in the input data are used to re-fragment the system.

References

[1] L. Bellatreche, K. Karlapalem, and A. Simonet. Horizontal Class Partitioning Design in Object-Oriented databases. In *Lecture Notes in Computer Science volume 1308 DEXA*, pages 58–67, Toulouse-France, September 1997.

[2] S. Chakravarthy, J. Muthuraj, R. Varadarajan, and S. B. Navathe. An Objective Function for Vertically Partitioning Relations in Distributed Databases and its Analysis. *Distributed and Parallel Databases*, 2(1):183–207, 1993.

[3] C. I. Ezeife and Ken Barker. Distributed Object Based Design: Vertical Fragmentation of Classes. *Journal of Distributed and Parallel Database Systems*, 6(4):327 – 360, 1998. Kluwer academic Publishers.

[4] C.I. Ezeife and Ken Barker. A Comprehensive Approach to Horizontal Class Fragmentation in a Distributed Object Based System. *International Journal of Distributed and Parallel Databases*, 1:247–273, 1995. Kluwer Academic Publishers.

[5] C.I. Ezeife and Jian Zheng. Measuring the Performance of Database Object Horizontal Fragmentation Schemes. submitted to the 3rd international database engineering and Applications Symposium (IDEAS99), 1999.

[6] M.L. Kersten, S. Plomp, and C.A Van Den Berg. Object Storage Management in Goblin. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[7] Barbara Liskov, A. Adya, M. Day, M. Castro, S. Ghemawat, R. Gruber, L. Shrira, A.C. Myers, and U. Masheshwari. Safe and efficient sharing of persistent objects in THOR. In *proceedings of ACM SIGMOD International Conference on Management of Data*. Association of Computing Machinery, 1996.

[8] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[9] M. Savonnet, M. Terrasse, and K. Yetongnon. Fragtique: A Methodology for Distributing Object Oriented Databases. In *Proceedings of the 9th International Conference of Computing and Information ICCI'98 Winnipeg*, pages 148–159, 1998.