Experimental Evaluation of Dynamic Data Allocation Strategies in a Distributed Database With Changing Workloads †

Anna Brunstrom[1], Scott T. Leutenegger[2] and Rahul Simha[1]

[1]*Department of Computer Science*
College of William and Mary
Williamsburg, VA 23185
{brunstro,simha}@cs.wm.edu

[2]*Mathematics and Computer Science Department*
University of Denver
2360 S. Gaylord Street
Denver, CO 80208-0189
leut@cs.du.edu

Abstract

Traditionally, allocation of data in distributed database management systems has been determined by off-line analysis and optimization. This technique works well for static database access patterns, but is often inadequate for frequently changing workloads. This paper addresses the problem of dynamically reallocating data in a partionable distributed database with changing access patterns. Rather than complicated and expensive optimization algorithms, a simple heuristic is presented and shown, via an implementation study, to improve system throughput by 30% in a local area network based system. For a wide area network the performance gain is expected to be even larger. It is also shown that individual site load must be taken into consideration when reallocating data. A a simple policy that incorporates load in the reallocation decision is provided.

†This research was partially supported by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the second author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

1 Introduction

To achieve good performance in a distributed database it is often recommended that portions of the database be located at the sites from which they are most frequently accessed. Prior research in the area of data allocation has typically assumed different site access frequencies are known, in order to formulate a discrete optimization problem and solve it off-line to find near-optimal locations for parts of the database [1, 5, 7, 10, 15, 21, 24, 25] – see [11, 30] for a survey. Our work is motivated by a key observation not applicable to much of past research in this area: site access patterns and their consequent workloads in a database are often *not static*.

In this paper, we address the problem of how to reallocate portions of the database in a distributed system with a *changing* workload, that is, when the access frequencies to various portions of the database from a particular site vary with time. Changing workloads are found in many applications in which workloads and access patterns to different portions of the database from the sites change for a variety of reasons. These reasons range from short-term system load fluctuations to long term global changes due to gradual growth in data and changes in daily patterns of human users.

Consider the example of a distributed global stock trading database. Typically, at any given time, intensive trading (and hence database access) occurs at places that are currently within business hours. As the day progresses, places in the East start closing, resulting in diminished access, while active trading appears farther West. Rather than subject users to continually large communication costs during trading (by permanently assigning the data to fixed locations), it makes eminent sense to move the relevant parts of the database to where accesses are most active. In this way, most accesses are likely to be local, thereby improving response times and throughput. In such an environment of access patterns, no static assignment of the data across the different sites can be optimal throughout the twenty-four hour cycle. Note that temporal variations in access patterns are also found in databases located entirely within a local area network, for example, when users switch among different tasks.

To the best of our knowledge, nobody has successfully demonstrated data re-allocation algorithms in a working database with a realistically changing workload, although scores of papers propose schemes based on queueing models that work in theory (most of which make steady-state assumptions). In this paper we study re-allocation in a working distributed database, in particular, a benchmark-standard database with concurrency control, two-phase commit and recovery overheads. The re-allocations take place simultaneous with the regular operation of the database, and are achieved within the framework of a transaction, hence all data within a partition being re-allocated is locked during the move. We have implemented and tested two simple strategies in a distributed relational database (that utilizes an object server as the database engine) running a standard benchmark on a cluster of workstations. We consider only simple algorithms that maintain counts of the number of accesses made by sites to blocks of the database, and that try to place blocks where they are most frequently accessed. While our algorithms could be criticized for their simplicity, we find that a real system has a lot of "noise", especially with a changing workload, and that in practice it is quite difficult to get even simple schemes to work well. Nevertheless, we show that provided that the workload does not change too rapidly, simply-computed dynamic allocations can offer a significant improvement in transaction throughput over static allocations. Our experimental results also show that, besides striving to maximize the amount of local access to the data, load balancing is an important practical consideration when re-allocating the data. Loosely speaking, this paper is less about touting new load re-allocation algorithms than a proof-of-concept in implementing data re-allocation in a real system with a changing workload.

Our data re-allocation schemes were implemented in a distributed relational database built upon the Exodus storage manager [3, 4], which has built-in concurrency control, logging and robust communication protocols. While our system is not a highly tuned relational system comparable to large commercial databases, using the Exodus storage manager as our database engine enables us to easily include realistic overheads, thus adding credibility to the application of our results to real database management systems. The sizes of relations used in our experiments would generally classify the database as small; owing to resource limitations, larger sizes could not be tested. In addition, we limit our system to two sites in this initial study to retain focus on the central issues. Nonetheless, our goal is to show proof of concept and thus we are interested in the relative performance of our system with and without dynamic re-allocation.

Our workload model assumes that the database is partitionable [29], i.e. relations can be decomposed into groups of tuples each of which can be placed on different sites in the system. Many workloads, including the TPC benchmarks [16], have this property. The partitionability of the workload allows for a logical unit of re-allocation and load balancing.

The rest of the paper is organized as follows. In the next section we describe how our work is related to past work. In Section 3 we describe our system and the algorithms used for detecting changes to the access pattern and moving data in response to the change. In Section 4 we describe the workload we considered. Section 5 contains our results. Our conclusions and plans for future work are presented in Section 6.

2 Related Work

The problem of where to locate relations or portions thereof among the sites in a distributed database, when given a static access pattern, is often treated under the File Allocation Problem [1, 5, 7, 10, 9, 11, 15, 24, 25, 30]. The general problem, which has been shown NP-complete [14], is usually formulated as an integer-programming problem with separate query and update frequencies specified for each site. In other cases, a database is assumed to be arbitrarily fragmentable and queueing-theoretic considerations are used for a static optimum [21]. In this paper we are interested in dynamically changing access frequencies for which re-solving combinatorial problems in a centralized manner for each system change is prohibitively expensive. Instead, we focus on non-optimal but small and fast heuristic re-adjustments to data allocations in response to changing conditions.

Theoretically, if future changes in workload could be accurately predicted, a Markov-decision problem [32] could be formulated and solved in advance to optimize average response times. However, in practice such predictions may not be possible, even statistically. In addition, several other system variables such as the communication time using transport protocols over an Ethernet are likely to be difficult to characterize. While data migration policies have been discussed in the context of dynamically changing environments [12, 27], to the best of our knowledge, practical implementations of these ideas have not been demonstrated in the literature. Some experimentation with load balancing of tasks (to balance CPU demands) has been reported in [22, 31, 33], but the results do not apply to database partitions. Theoretical work on load balancing can be found in [23, 26, 6, 13]. We note that the problem of scheduling data redistributions so as to minimize re-allocation time has been examined in earlier work [28]. We do not address how to minimize transfer cost since we surmise this is secondary to the importance of identifying and transferring the partitions to be re-allocated.

File replication has also been used for providing fast local access (and availability) to data shared throughout a distributed database. Issues that have been addressed include mechanisms for keeping replications consistent, tradeoffs between the availability acquired from numerous copies versus update cost, and the effect different concurrency control algorithms have on these tradeoffs [8, 17, 18, 19, 20]. In this study we do not consider replication, instead we assume only a single copy of the data exists. We make this choice to allow us to understand the issues in this simpler case first before investigating more complicated systems. We believe our results can be extended to re-allocation of copies of the data if replication is used. Further differences between our work and prior research will be more apparent following a description of our system and the algorithms used, a matter to which we next turn.

3 System and Algorithm Description

This section contains a description of the distributed database used for our experiments. The first subsection explains how our relational database is built on top of the Exodus storage manager. Our distributed process system structure is introduced next, and finally the last subsection describes the heuristic algorithms used for data re-allocation.

3.1 Relational System Structure

Our relational database system is built on top of Exodus [3, 4], a multi-threaded distributed object-oriented database system. All interaction with Exodus servers is provided through Exodus client library calls. We essentially use Exodus as an object server and to provide features such as concurrency control, indexing and recovery. Building our system on top of the Exodus client library has allowed us to expedite the process of system development while at the same time incorporating the overhead imposed by concurrency control and recovery. Although our usage of the object-oriented system for transaction processing is inefficient and results in low throughput, the system is appropriate for the purpose of studying the tradeoff between static and dynamic data partitioning in a distributed database system. We are interested in the relative performance of the system under various data allocation schemes; the actual measured response times and throughputs are not themselves significant.

The implementation of our relational database consists of approximately 3,000 lines of C++ code. A transaction in our relational system is encapsulated within a transaction in Exodus. Since all our data manipulation in the relational system is performed within Exodus transactions we rely on Exodus for concurrency control and recovery. As discussed next, we also rely on Exodus to provide transparent access to data stored at remote sites. Each tuple in our relational database is associated with an object in the underlying Exodus database. An object in Exodus is identified by an object identifier. Whether the object is stored on a local or remote data volume is transparent to a user of the Exodus library. The Exodus client library retrieves the data from a local or remote server as appropriate. When multiple servers are involved in a transaction Exodus employs a distributed two-phase commit protocol to ensure consistency. Exodus provides functions for direct retrieval of an object given its object id as well as functions for scanning all objects in a file. A file in Exodus is identified by a file identifier returned when the file is created. Again, whether a file (relation) is stored at a local or remote server is transparent to a user accessing the file. To get a handle on the data on a data volume Exodus provides what is called "root entries". A root entry allows the user to store a small amount of data and to associate a name with this data.

Each relation in our relational database corresponds to a file of objects in Exodus. The file identifier for the file (relation) is stored in a root entry under the name of the relation. Thus to access a relation we need to know the name of the relation as well as the data volume on which it is stored. Accessing a remote root entry is again transparent to the user. Given a data volume, the Exodus client library determines whether the data volume is maintained by a local or remote server and retrieves the appropriate root entry. In our system the data is partitioned among the sites and a "Branch", "Teller", "Account", and "History" file (relation) exist on each data volume. A scan of a relation in our relational model is implemented on top of Exodus file-scanning functions. Functions for inserting, deleting and updating a tuple are built on top of Exodus' object manipulation functions.

Our relational model supports primary B^+ tree indexing built on top of the B^+ tree indexing provided by Exodus. For an indexed relation the object ids of the tuples (objects) in the relation are stored in an index provided by Exodus. The key used for the Exodus index is the same as the key used for the relational model. Retrieving a tuple from an indexed relation then involves two steps. Firstly, the key is used to retrieve the object id of the tuple from the Exodus index. Secondly, the object id can be used for direct retrieval of the desired tuple (object). Similarly, inserting a tuple in an indexed relation involves two steps; first the tuple is inserted in the relation (file) and then the returned object id is inserted in the Exodus index. Indexed scans are implemented on top of Exodus function for scanning an index. We built indexes on the Account-ID, Teller-ID, and Branch-ID attributes of the Account, Teller, and Branch relations.

3.2 Distributed System Process Structure

The database at each site consists of a copy of the Exodus server and several client processes built on top of our relational database. All client processes are also implemented in C++ (about 2,500 lines of code). The data is non-replicated and partitioned among the sites. The data for a site is stored in a local data volume and is maintained by the Exodus server local to the site. The "home-site" of an account tuple refers to the original storage site of the tuple. Thus the "home-site" of a tuple never changes even though the tuple may be re-located to a different site in the system. All tuples in a block have the same "home-site". A brief description of the processes at each site and their function is given below:

Exodus server: This process maintains the data stored on the local data volume. It services requests made through local and remote Exodus client libraries.

Generator process: The generator process generates and executes transactions in a serial fashion. Transactions may need to access both local and remote data. Thus the Exodus client library compiled into the generator process interacts with both its local and remote Exodus servers. Several generator processes and a move process (defined below) may interact with a server concurrently. Transactions are generated consistent with the workload described in the next section. Information about each transaction is sent to the stats process (defined below) located at the home-site of the account tuple accessed by the transaction.

Stats process: The stats process does not have an Exodus client library compiled into it. It gathers the statistics for our experiments with the database based on messages received from the generator processes. It accumulates statistics such as throughput, average response time for a transaction, and the fraction of transactions requiring access to remotely stored data. The stats process at a particular site is responsible for the statistics of transactions which access account tuples with that site as their home-site. As described above, the accessed tuples may or may not be physically stored at their home site. When the database uses a dynamic data allocation scheme the stats process is also responsible for monitoring the access pattern for the tuples of its home-site and for deciding when a block of tuples need to be relocated. When the stats process decides that a move is needed it sends a message to a move process to carry out the action. The re-allocation algorithms are described in detail in the next subsection.

Move process: The move process is active only when the database uses a dynamic data allocation algorithm. It receives requests for moving a partition of some relation, the Account relation in our application, from one site in the system to another. Similar to the stats process, the move process at a site is responsible for moving tuples with that site as their home site. The Exodus client library in the move process interacts with two Exodus servers to move the tuples. The move operation is performed as a sequence of two Exodus transactions. In the first transaction the tuples to be moved are copied over to the new site. The data is unavailable during the copying. A message is then sent to all relevant processes in the system notifying them about the change in location. The tuples are then deleted from the old site in a second transaction. Hence, during the move the data is unavailable and the throughput is decreased for transactions accessing the partition being moved.

In addition to the processes described above, which exist at each site of the database, we use one *driver process* to control our experiments. The driver process is responsible for broadcasting workload changes and broadcasts a quit request at the end of an experiment. The structure of our system is illustrated in Figure 1. In the figure solid lines indicate communication provided by Exodus whereas dashed lines indicate communication external to Exodus. All communication external to Exodus is done through datagrams. In our initial experiments we assume our database is distributed over two sites. Our algorithms generalize to multiple sites but we started with two sites to retain focus on the central issues.

3.3 Re-Allocation Algorithms

This subsection gives a description of the algorithms used for dynamic re-allocation of the data. As mentioned earlier, we are interested in simple heuristic algorithms that can be easily implemented and that will allow us to show that dynamic re-allocation is worthwhile in a distributed database with a changing workload. We assume relations can be partitioned in some natural manner into several fixed-size blocks or groups of tuples. In our case, the Account relation in the TPC-B benchmark [16] (see the next section for more details) is horizontally partitioned into fixed size partitions, where partition i consists of the account tuples belonging to branch i. This is a natural partition since accesses probabilities for the tuples within a branch are uniformly distributed in our workload.

To determine when a re-allocation is needed our algorithms maintain weighted counters of the

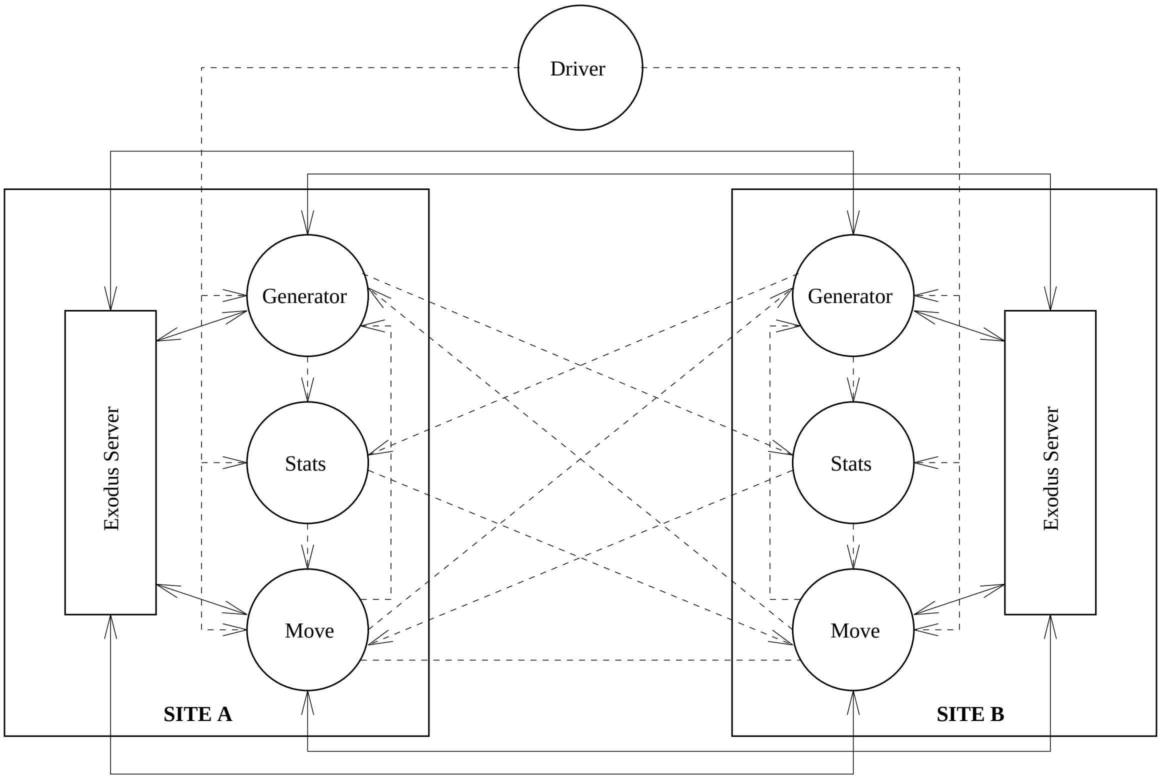

Figure 1: System Structure.

number of accesses from each site to each block. The counters for a particular block is maintained at *only one* of the sites in the system. For example, assume site 1 is the home site for partition A. Site 1 maintains a counter for the use of A by site 1 and another counter for the use of A by site 2. In a system with N sites, site 1 would maintain N counters for partition A. When a transaction accessing an account tuple from block i completes, a message containing information about the transaction is sent to the stats process at the site responsible for block i. The counters for a block are updated each time a tuple within the block has been accessed by a transaction.

As is typical in estimating a moving average, we want to discount prior samples to allow the most recent samples to properly influence the current estimates. We use a simple exponential aging scheme: the counters for a block are updated by multiplying the current values by an *aging factor* and then adding one (the latest sample) to the counter of the site where the access originated. For effective estimation, the aging factor must be small enough to allow the counters to adjust to the dynamic workload but large enough to prohibit moves due to an "unlucky" streak of requests. For our system and workload we have found an aging factor of 0.9 to work well.

We now describe two simple algorithms for dynamically re-allocating the blocks of data using the counters described above. The first simply ranks the sites according to counter values and picks the best site. The second takes into account load conditions; after all, if too many blocks were placed on a single machine then potential inter-transaction parallelism could be lost and overall throughput might decrease. Our simple counter algorithm is shown below:

Simple Counter Algorithm:

1. *The* stats *process examines the counters for each block at regular intervals.*

2. *The tuples for a block are moved if the site with the highest counter value is a site other than the current storage site.*

3. *After checking the counters for a block the stats process will wait for t_check number of transactions to be completed for the block before checking the counters again.*

Note that the value of *t_check* should be small enough to allow the system to respond quickly to workload changes but large enough to prevent premature signaling of a change in access frequencies and having the data bounce back from a move soon after. The influence of the value of *t_check* on performance is evaluated in the section on experimental results.

Determining whether a data block needs to move is a *fully local* decision, since all counters for a block are stored at the same site. This is a very desirable property of the algorithm that greatly improves scalability. The only part of the algorithm affected by the system size is the "new location" message broadcast when a data block is moved. Assuming that data relocation does not occur too frequently this broadcast should not impair performance. The communication overhead induced by monitoring the access pattern in the system is bounded by one message per transaction. Considering the communication needed to ensure consistency (distributed two-phase locking, two-phase commit) and to access the data in a distributed database this is a very low overhead.

The simple counter algorithm works well as long as the load in the system remains low or relatively balanced. However, in some cases, tuples for all blocks can end up at the same site if most requests for all blocks originate at the same site. Although the fraction of local access is optimal, this allocation may cause poor performance due to disk or CPU overloading at the site. The load sensitive counter algorithm addresses this problem:

Load Sensitive Counter Algorithm:

1. *Monitor the load (data balance) in the system as well as the access frequencies.*

2. *The need for a move is evaluated as in the simple counter algorithm. However, the moves are only carried out as long as they do not cause the portion of data stored at a site to exceed a specified threshold value. The maximum percentage of the total storage requirement for the database that is allowed storage at a site, the load threshold value, is a parameter of the algorithm.*

As the algorithm indicates, our load balancing is performed based on the disk load in the system. By disk load we mean the portion of the total database stored at a given site. Note that other metrics such as cpu or disk utilization would have a similar result since our application is such that the CPU load is roughly proportional to the disk load at each server. The threshold value does not refer to the utilization of the disk, rather it refers to the portion of the data in the database allowed at a site. Hence our threshold is an indicator of the data imbalance tolerated. It is assumed that each site has enough storage capacity to meet the imposed storage requirements. The selected threshold value is a measure of the imbalance allowed in the system. When a threshold value of 100% is used the load sensitive counter algorithm reduces to the simple counter algorithm. The data is allowed to move without constraints and all data in the database could be stored at a single site. On the other hand, using a threshold value of 50% in a database with two sites enforces even-sized data partitions and results in static allocations.

4 Workload Description

The workload in our system is based on the TPC-B benchmark [16]. We are using a modified version of the benchmark, where the access rules are slightly changed to create access distributions that vary with time. The TPC-B benchmark provides an effective basis for our workload since it is well understood in the database community and has been used extensively for performance evaluation of commercial as well as experimental systems. The TPC-B benchmark emphasizes update-intensive database services in the context of a hypothetical bank. The database contains four relations: a *Branch* relation, a *Teller* relation, an *Account* relation and a *History* relation. A single transaction which models a deposit or withdrawal from the bank is repeatedly performed. The transaction updates a tuple in the Branch, Teller and Account relation to reflect the new balance for the respective entity and inserts a tuple in the History relation recording the transaction. The branch and teller accessed by a transaction are local to the site at which the transaction originated whereas the account may be either local or remote. The probability of accessing an account belonging to

Sites	Branches							
	b0	b1	b2	b3	b4	b5	b6	b7
s1	0.2	0.2	0.2	0.2	0.05	0.05	0.05	0.05
s2	0.05	0.05	0.05	0.05	0.2	0.2	0.2	0.2

Figure 2: Sample access matrix

the branch being accessed is 85% for the TPC-B benchmark. Our actual workload differs in that we assume a slightly larger percentage of calls to remote branches; 80% of the account tuples are located on the local site and 20% are located on the remote site. In addition, we assume each branch tuple has 10 teller tuples but only 2,500 account tuples associated with it. Due to resource limitations, the number of account tuples associated with a branch is smaller in our workload relative to the number specified in the TPC-B benchmark. Our buffer sizes were chosen such that the Branch and Teller relations were memory resident, but I/O was required to access account tuples.

While the TPC-B benchmark is static in terms of workload statistics, our changing workload is derived from the benchmark by changing the access probabilities for the account tuples. The access probabilities are given by an access matrix which specifies the probability of accessing an account tuple from a given branch when the transaction originates at a given site. The probability of accessing a given tuple within a branch is uniformly distributed. A changing workload is created by periodically changing the access matrix. Thus the workload is completely described by the sequence of matrices used and the times at which they change. In our experiments the driver process is controlling what matrix is used for generating transactions. A sample access matrix for our system, which contains 8 branches distributed over two sites, is displayed in Figure 2. If the same matrix values were used continually (static usage), the workload would not change; to create a changing workload, site access probabilities for the branches in the above matrix are made to vary with time (see next section). Static usage of the matrix in figure 2 with branches 0 through 3 allocated to site one and branches 4 through 7 allocated to site 2 would create a workload very similar to the TPC-B benchmark. Rather than using the matrix statically, in our workload the matrix is used as the initial matrix in a sequence of different matrices.

5 Results

In this section we present our experimental results. The performance of the system under our dynamic re-allocation schemes is compared to the performance of the system under a static data partitioning. As mentioned earlier, our database system is admittedly somewhat slow and therefore the actual numbers presented in this section should not be seen as representative of a highly tuned relational database running on dedicated machines. Rather, we ask the reader to focus on the relative performance of the dynamic and static systems and on the performance trends exhibited by our dynamic algorithms as various parameters are varied. The machines used for our experiments were all Sun4 Workstations running SunOS 4.1.3 and connected through a local Ethernet. The total storage requirement of our database was approximately 3M. We allocated (via the server buffer group) a buffer pool of 800K at each site. We chose a small buffer size to reflect the fact that most real databases are not memory resident. The data was stored on raw disk partitions to prevent secondary buffering by the operating system. The Branch, Teller and Account relations were all implemented as indexed relations. The Branch-ID, Teller-ID, and Account-ID were used as the key for their respective indexes.

5.1 The Simple Counter Algorithm

The first set of experiments evaluates the influence of various parameters on our simple counter algorithm and compares its performance to the static system. The results are intended to answer the question: when is it useful to use dynamic re-allocation? We find that dynamic re-allocation is desirable over a broad range of parameters; however, it is not effective when the system workload is changing very rapidly.

The changing workload used for the experiments is defined by a sequence of matrices, each of which is used in one interval in the sequence of intervals that comprise the progress of time. By successively numbering these intervals and identifying a matrix with each one, we will have specified the changing workload. The matrix shown in Figure 2 was used for interval zero. The matrix used for interval $i + 1$ was constructed by swapping the access frequencies for site one and two for branches $i \bmod 8$ and $(i + 4) \bmod 8$. Thus the workload is periodic with 8 matrices in a period.

The access matrix was changed every *t_change* seconds. Except for in the second experiment *t_change* was set at a constant value of 800 seconds. Averaged over all matrices in a period the probability of an access coming from a particular site is 0.5 for every branch. Thus any static allocation which positions half of the branches at each site is optimal (among static allocations). The static data partitioning we used placed branches 0 through 3 at site one and branches 4 through 7 at the second site. These were also the initial branch allocations used for the dynamic system. Thus the system was started in the same initial state irrespective of which allocation mechanism – dynamic or static – was used. Furthermore, the system was started up with the data in an optimal static partition.

The first experiment assesses the influence of the value of *t_check* on our simple counter algorithm. Recall that *t_check* is the number of accesses needed for a data block before relocation of the block is considered. The throughput for various values of *t_check* (measured in seconds) as well as the throughput for the static system is displayed in Figure 3. We can see that the dynamic

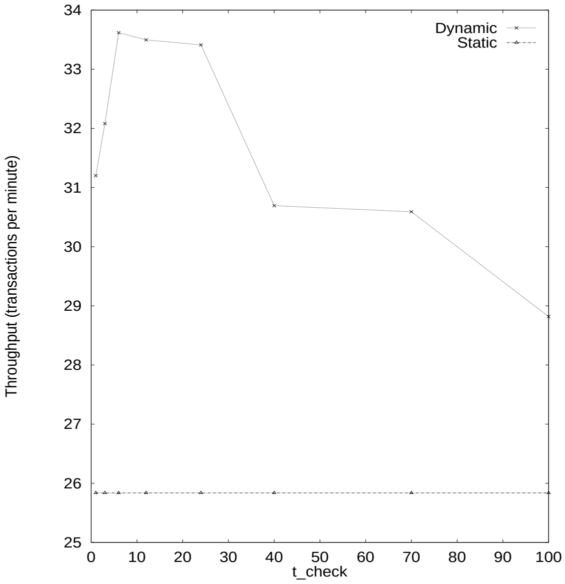

Figure 3: Influence of *t_check*

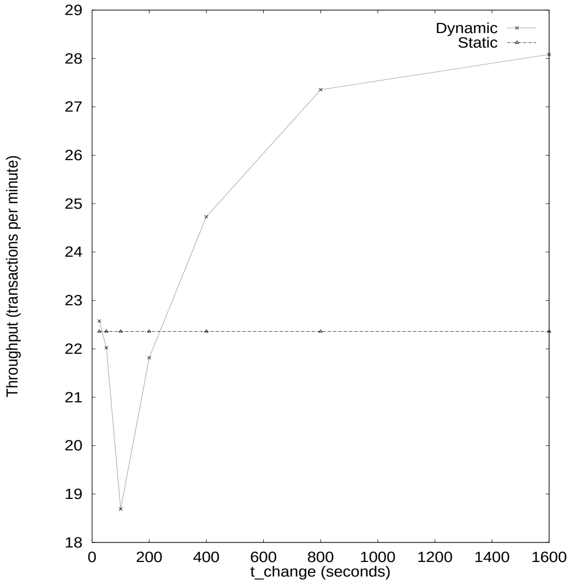

Figure 4: Influence of *t_change*

re-allocation scheme clearly outperforms the static allocations for our workload. The influence of *t_check* on the performance is what we would intuitively expect. When the value of *t_check* is too small, performance degrades since the partitions may bounce back and forth between the two sites during workload changes. Since our threshold on the fraction of requests coming from a site for re-allocation to occur is 50% an unfortunate streak of requests will cause a bouncing effect. Note, this could be mitigated by increasing the threshold. On the other hand, when the value of *t_check* is too large the algorithm responds too slowly to workload changes and performance plummets. We can see in Figure 3 that a value of $t\_check = 6$ works well for our system. This was the value used in the remainder of the experiments. In practise, the value for *t_check* could be adjusted adaptively.

Our next experiment was designed to evaluate how the effectiveness of our dynamic re-allocation scheme is influenced by the rate at which the workload changes. The total run time of the experiment was held fixed but the rate at which the workload changed was doubled between each successive run. As a result the overall time spent using a certain access matrix remained constant for all runs and thus the performance of the static system was unaffected. The performance of the counter algorithm as a function of *t_change*, the time between workload changes, is displayed in Figure 4. The performance of the static system is also displayed in the figure.

As we might expect, the dynamic re-allocation scheme performs best when the rate of change is low. In this situation, our dynamic re-allocation scheme can maintain optimal allocations most of the time. As the rate of change increases the dynamic re-allocation algorithm cannot keep up with the changing workload and performance drops. Performance is at its worst when the algorithm is attempting to keep up with the workload but is unable to do so. At this point lots of re-allocation is performed but by the time a partition is re-allocated, the workload is about to change or has already changed again. Hence, the data ends up being in the wrong place most of the time and the system performs worse than when static allocations are used. When the workload changes very fast the dynamic re-allocation algorithm no longer has time to detect the changes in workload and thus will not even try to re-allocate the data. Performance improves over the previous case and becomes similar to the performance of the static algorithm.

We can see from figures 3 and 4 that as long as the workload doesn't change too fast and we use appropriate parameter settings our dynamic re-allocation algorithm offers an approximate 30% performance gain over static allocations. These results are for a Local Area Network. When the database is distributed over a WAN we would expect a dynamic re-allocation algorithm to offer even greater performance gains. Over a WAN, remote access to the data will be more costly and thus the allocation of the data even more crucial.

5.2 Load Balancing Considerations

For the workload used in the experiments described in the previous subsection the load in the system remains fairly balanced. However, as mentioned earlier, if the majority of the account requests come from the same site for all blocks, then all the data will end up at the same site under our simple counter algorithm. Although this produces an optimal fraction of local requests the load in the system is highly imbalanced and performance may not improve. Our load sensitive counter algorithm and our final experiment was designed to evaluate the tradeoff between maximizing the fraction of local requests and maintaining a balanced load in the system.

It is not hard to envision a situation where all requests for a relation temporarily originate at the same site. Consider a database distributed over a cluster of workstations. Only a subset of the workstations may be able to physically store portions of the database. (In our system only two machines run a database server.) If only one of the machines running a server has users logged in at the moment, then all requests relevant to the allocation of the data originate at one site. Additional users may be using the database from sites which do not have a database server, and thus cannot be considered when relocating the data, thereby creating a high system load at sites which have servers. Focusing strictly on maximizing the fraction of local requests in this situation may then cause poor performance. Note that requests from users at sites which cannot store portions of the database will always have to be serviced remotely. For these requests load balancing strongly influences performance.

The workload for our final experiment was modified to represent a situation similar to the one described above. Only one of our two sites containing the database generated TPC-B transactions. To increase the load in the system additional transactions were generated at three new sites in our network of workstations. Since there was no local database at these sites, modifications to the local

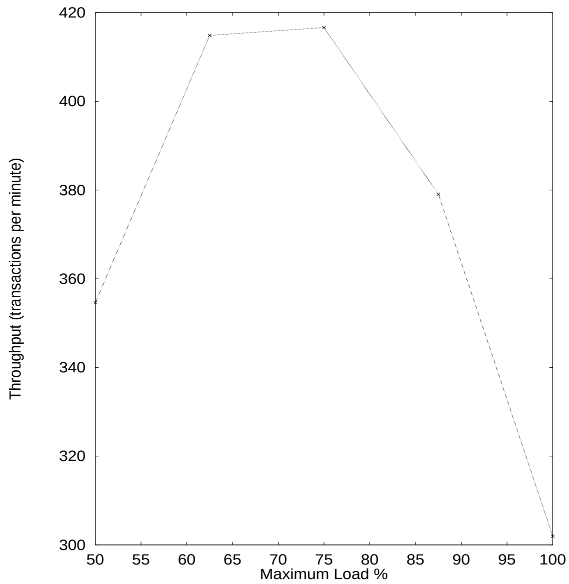

Figure 5: Sensitivity to load

Branch, Teller and History relations were omitted. The transactions generated at the new sites only modified a tuple in the Account relation. Requests were uniformly distributed within entire relations for all transactions. The workload should be seen as representative of one configuration of a changing workload where load balancing might have to be considered. As before, the branches were initially divided evenly between the two sites of the database. Performance of the system under our load sensitive counter algorithm was measured while varying the load threshold value for the algorithm.

Figure 5 displays the system throughput as a function of the load threshold used to restrict data movement. The tradeoff between load balancing and maximizing the fraction of local transactions is clearly displayed by the graph. For our system and the workload used, a load threshold value of 75% gives optimal observed performance. When the load threshold is set at 50% the system is fully load balanced but the system is static and the fraction of local transactions is far from maximized. As the load threshold is increased, performance initially improves since the fraction of local transactions increases. When the load threshold goes past 75%, performance starts to drop due to contention at the site handling all the data. Even though the fraction of local transactions increases, performance declines when the site handling the data gets overloaded. For a load threshold of 100%, contention causes performance to drop below the value obtained for the static system. Thus we can see that for an extreme workload, like the one considered in this experiment, our simple counter algorithm would not perform well. (Recall that when a threshold of 100% is used the load sensitive counter algorithm reduces to the simple counter algorithm.) The balance of load in the system is therefore also an important consideration.

6 Conclusions

Performance in distributed database systems is heavily dependent on the allocation of data among the sites of the database. The allocation of data is traditionally static and determined off-line, using estimates of access frequencies. However, in many situations the access frequencies from varies sites in the database are not known a priori or fluctuate with time thereby creating a changing workload. This paper showed that for a database with a changing workload dynamic re-allocation of the data can significantly improve performance.

A simple counter algorithm was presented which monitors the access frequencies in the system and moves the data so as to maximize the fraction of local accesses in the system. For our workload the algorithm offered up to 30% performance gain over static allocations in a Local Area Network. We would expect even higher performance gains in a Wide Area Network as the cost of going across the network increases. Our experiments also showed that for certain workloads load balance must be considered when re-allocating the data. We presented a load sensitive counter algorithm which was shown to outperform the simple counter algorithm as well as static allocations for a class of workloads.

While our experiments with a small-scale database make a practical case for dynamically re-allocating data in a changing environment, more work is needed to study various possible algorithms for dynamic data allocation and to test these algorithms on large-scale distributed databases. Several structural issues seem worthy of investigation such as appropriate block sizes for partitioning and the granularity at which statistics should be gathered for blocks in different relations. Our future plans also include lending theoretical support to our experimental results through analytical models.

Acknowledgements
We would like to thank the Exodus project for making Exodus available to the database community. We would also like to acknowledge the work of Brian Dewey in initially implementing the TPC-B workload running on top of the Exodus storage manager.

References

[1] P.M.G.Apers. Distributed Allocation in Distributed Database Systems. *ACM Trans. Database Sys.*, Vol. 13, No. 3, September 1988.

[2] B.Awerbuch, Y.Bartal and A.Fiat. Competitive Distributed File Allocation. *Proc. ACM STOC 1993*, 1993, pp. 164-173.

[3] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, Readings in Object-Oriented Databases, pages 474-499, *Morgan-Kaufman*, 1990.

[4] M. Carey, D. DeWitt, D. Frank, G. Graefe, J. Richardson, E. Shekita, and M. Muralikrishna. The Architecture of the EXODUS Extensible DBMS. *Proc. 1st Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[5] S.Ceri, G.Pelagatti and G.Martella. Optimal File Allocation in a Computer Network: A Solution Based on the Knapsack Problem. *Computer Networks*, Vol. 6, 1982, pp. 345-357.

[6] Y-C.Chow and W.H.Kohler. Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System. *IEEE Trans. Computers*, Vol. C-28, No. 5, 1979, 354-361.

[7] S-K.Chang and A-C.Liu. File Allocation in a Distributed Database. *Int. J. Comp. Info. Sci.*, Vol. 11, No. 5, 1982, pp. 325-340.

[8] B.Ciciani, D.M.Dias, and P.S.Yu. Analysis of Replication in Distribute Database Systems. *IEEE TKDE*, Vol. 2, No. 2, 1990, pp. 247-261.

[9] D.W.Cornell, and P.S.Yu. On Optimal Site Assignment for Relations in the Distributed Database Environment. *IEEE TOSE*, Vol. 15, No. 8, 1989, pp. 1004-1009.

[10] W.W.Chu. Optimal File Allocation in a Multiple Computer System. *IEEE Trans. Comp.*, Vol. C-18, 1969, pp. 885-889.

[11] L.Dowdy and D.Foster. Comparative Models of the File Assignment Problem. *ACM Comput. Surveys*, Vol. 14, June 1982, pp. 287-314.

[12] H.Du and Maryanski. Data Allocation in a Dynamically Reconfigurable Environment. *Proc. 4th Int. Conf. Data Engg.*, 1988.

[13] D.L.Eager, E.D.Lazowska, and J.Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, vol. 12, No. 5, 1986, pp. 662-675.

[14] K.Eswaran. Placement of Records of a File and File Allocation in a Computer Network. *IFIP Conf.*, 1974, pp. 304-307.

[15] M.L.Fisher and D.S.Hochbaum. Database Location in Computer Networks. *J. ACM*, Vol. 27, 1980, pp. 718-735.

[16] J. Gray (Editor). The Benchmark Handbook for Database and Transaction Processing Systems, *Morgan-Kaufmann*, 1991.

[17] A.Hac. A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration. *IEEE TOSE*, vol. 15, No. 11, 1989, pp. 1459-1470.

[18] A.Hac, X.Jin, and J-H.Soo. Algorithms for File Replication in a Distributed File System. *Journal of Systems Software* Vol. 14, 1991, pp. 173-181.

[19] H.Inazumi, M.Kochiya, and S.Hirawawa. On the Trade-Offs Between the File Redundancy and the Communication Costs in Distributed Database Systems. *IEEE Tran. Systems, Man, and Cybernetics*, vol. 19, No. 1, 1989, pp. 108-112.

[20] A.Kumar, and A.Segev. Cost and Availability Tradeoffs in Replicated Data Concurrency Control. *ACM TODS*, Vol. 18, No. 1, 1993, pp. 102-131.

[21] J.Kurose and R.Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Trans. Comp.*, Vol. 38, No. 5, May 1989, pp. 705-717.

[22] W.E.Leland and T.J.Ott. Load Balancing Heuristics and Process Behavior. *ACM SIGMETRICS*, 1986, pp. 54-69.

[23] M. Livny and M. Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. *Proc. Modeling Perform. Eval. Comput. Syst., ACM SIGMETRICS*, April 1982, pp. 47-55.

[24] S.Mahmoud and J.S.Riordan. Optimal Allocation of Resources in Distributed Networks. *ACM Trans. Database Sys.*, Vol. 1, No.1, March 1976, pp. 67-78.

[25] H.L.Morgan and K.D.Levin. Optimal Program and Data Locations in Computer Networks. *Comm. ACM*, Vol. 20, No. 5, May 1977, pp. 315-322.

[26] L.M.Ni and K.Hwang. Optimal Load Balancing in a Multiple Processor System with Many Job Classes. *IEEE Trans. Soft. Engg.*, Vol. SE-11, No. 5, 1985, pp. 491-496.

[27] T.P.Ng. Optimal Data Migration Policies in Distributed Databases. *Proc. 15th Int. Computer Software and Applications Conf.*, 1991.

[28] P.I.Rivera-Vega, R.Varadarjan, S.B.Navathe. Scheduling Data Redistribution in Distributed Databases. *Proc. 6th Annual Data Engineering Conf.*, 1990.

[29] T.Ozsu and P.Valduriez. Principles of Distributed Database Systems. *Prentice-Hall*, New Jersey, 1991.

[30] B.Wah. File Placement in Distributed Computer Systems. *IEEE Computer Magazine*, Vol. 17, January 1984, pp. 23-33.

[31] Y-T.Wang and R.J.T.Morris. Load Sharing in Distributed Systems. *IEEE Trans. Computers*, Vol. C-34, No. 3, 1985, pp. 204-215.

[32] D.J.White. Markov Decision Processes. Wiley, 1993.

[33] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. Soft. Engg.*, Vol. SE-14, No. 9, 1988, pp. 1327-1341.