CERIAS Tech Report 2002-69

Performance Evaluation of Linear Hash Structures in a Nested Transaction Environment

by S Madria, M Tubaishat, B Bhargava
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

ELSEVIER

The Journal of Systems and Software 63 (2002) 219–239

The Journal of Systems and Software

www.elsevier.com/locate/jss

Performance evaluation of linear hash structure model in a nested transaction environment

Malik Ayed Tubaishat [a], Sanjay Kumar Madria [a,*], Bharat Bhargava [b]

[a] *Department of Computer Science, University of Missouri-Rolla, Rolla, MO 65401, USA*
[b] *Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA*

Received 14 May 2000; received in revised form 11 October 2000; accepted 26 September 2001

Abstract

We design and implement a linear hash algorithm in nested transaction environment to handle large amount of data with increased concurrency. Nested transactions allow parallel execution of transactions, and handle transaction aborts, thus provides more concurrency and efficient recovery. We use object-oriented methodology in the implementation which helped in designing the programming components independently. In our model, buckets are modeled as objects and linear hash operations are modeled as methods. The papers contribution is novel in the sense that the system, to our knowledge, is the first to implement linear hashing in a nested transactions environment. We have build a system simulator to analyze the performance. A subtle benefit of the simulator is that it works as the real system with only minor changes.
© 2002 Elsevier Science Inc. All rights reserved.

1. Introduction

Nested transaction processing and concurrency control (Moss, 1985) issues play a major role in providing higher concurrency and better handling of transactions abort and hence have been an important area of research in database systems. Such a mechanism allows for the dynamic decomposition of a transaction into a hierarchy of subtransactions thereby preserving all properties of a transaction as a unit and assuring atomicity and isolated execution for every individual subtransaction. The motivation for using nested transactions is to allow transactions to exploit parallelism that might naturally occur within themselves. The benefits of parallel execution of transactions include performance improvements and better control over recovery.

Attention is being given to the designing of concurrency control algorithms which take advantage of the knowledge of particular data structures and the semantics of operations such as insert, delete, find, etc. to improve availability and expedite accesses. Data structures that have been studied with above in mind are B-trees (Sagiv, 1985; Lehman and Yao, 1981) and hashing techniques (Enbody and Du, 1988; Kumar, 1989; Larson, 1988; Ellis, 1987; Mohan, 1993).

Hashing is one of many addressing techniques used to find a record based on its unique key value. When a record is to be updated, the system takes the value of the key and performs some type of calculations to derive a target address for the record. Among various hashing techniques, Linear Hashing (Ellis, 1987; Hsu et al., 1990) is interesting because of its simplicity, ability to handle large amount of data with increased concurrency, and is known to be very fast in retrieving data. Linear Hashing allows storing records in a file without changing significantly its access time, independently of the number of record insertions or deletions that occur in it (Baeza-Yates and Soza-Pollman, 1998).

In Fu and Kameda (1989), B-tree algorithm in nested transaction environment has been presented to show its correctness but no implementation issues have been discussed. In Madria et al. (1998), linear hashing using nested transactions has been studied only with the aim of formalizing and proving the correctness of the algorithm using I/O automaton model (Lynch et al., 1994). In our work, we present a system implementation of a nested transaction version of the linear hash structure

* Corresponding author. Fax: +1-5733414501.
E-mail addresses: mma882@ume.edu (M.A. Tubaishat), madrias@umr.edu (S.K. Madria), bb@cs.purdue.edu (B. Bhargava).

0164-1212/02/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
PII: S0164-1212(02)00016-X

algorithm using object-oriented concepts and multithreading paradigm. In our model, buckets are modeled as objects and linear hash operations are modeled as methods. These methods correspond to nested transactions and are implemented using multithreading paradigm. To our knowledge, no implementation and the performance of nested transactions accessing linear hash structure has been available in literature.

Multithreading is a programming paradigm that allows the application processes to run concurrently. Multithreading can be used very efficiently to implement the behaviors of nested transactions. We have designed and implemented our system using layered system architecture in a three-tier client/server environment, which allows more flexibility in term of decomposing of application programs whose modules may be designed and implemented independently. The three-tiers of the system are split into the client, the database, and the application server (middleware) that resides between the client tier and the database tier. The implementation of our system was done in Microsoft VC++ using Microsoft Foundation Class (MFC).

The relationship between a client and a server is conducted by means of *transactions* consisting of well-defined requests and responses. More precisely, client/server technology is a paradigm or model for the interaction between concurrently executing software processes.

To test the performance and the efficiency of our system, we build two simulators; Client simulator and middleware simulator. They work as the client and the application server (middleware) applications, respectively. We randomly generate 10,000 keys accessed by randomly initiated operations (i.e., insert, delete, and find). More than one client simulator can run on each workstation whereas only one middleware simulator per workstation is needed. We test the performance of concurrent operations by reducing the sleep time between each client's request in the client simulator.

We run six simulations split into two parts. In the first part, we test the performance of the system running only insert operations. In the second part, we test the system running different operations randomly. We gradually increase the number of clients, or the number of operations in each experiment. The experiments mainly depict the performance of the system by presenting the average concurrency, elapsed time, and the number of times an operation occurs (operation occurrence).

According to the insert-operation simulation results: (1) Average concurrency increases proportionately with the increasing number of clients. (2) Average concurrency increases proportionately with the increasing number of insertions. (3) Elapsed time for the transactions in the insertions simulations does not change significantly. (4) Elapsed time does not affect the increasing number of clients, number of insertions, and the size of the database, (5) Split operation occurrence decreases when the database becomes larger.

According to our random-operation simulation results: (1) Average concurrency increases proportionately when number of clients, and consequently with number of operations increase. (2) Elapsed time for the transactions in the simulations does not change significantly. This implies that elapsed time does not affect the increasing number of clients, number of insertions, and the size of the database. (3) Merge elapsed time is lower than other operations elapsed time. (4) Average occurrence of split and merge operations is only 1% to 2% and in many cases, it is 0% which makes the system faster.

Rest of the paper is as follows. In Section 2, we present an overview of linear hash algorithm. Section 3 discusses our layered system architecture design in a three-tier client/server environment. Section 4 discusses the object-oriented implementation of the system. In Section 5, we present a simulation model to measure the systems performance and its evaluation. We conclude the paper in Section 6.

2. Background

2.1. Concurrency in linear hashing

In a linear hash structure (Ellis, 1987), there are primary buckets where each holds some constant number b of records. The function $h_0 : k \rightarrow \{0, 1, \ldots, N-1\}$ is initially used to load the file. There exists a sequence of functions $h_1, h_2, \ldots, h_i, \ldots$, such that for any key value k, either $h_i(k) = h_{i-1}(k)$ or $h_i(k) = h_{i-1}(k) + 2^{i-1}N$, where N is the initial number of buckets. The hash functions change as the hash structure grows or shrinks. The operations defined on a linear hash structure are find, insert, delete, split, and merge.

A variable *level* is used to determine the appropriate hash function for find, insert, or delete operations. A pointer called *next* is used to point to the bucket to be split or points to the bucket involved in the merge operation. *Level* and *next* are both initially set to 0, and together they referred to as *root* variables. Each bucket keeps an additional field *local-level* that specifies the hash function appropriate to that bucket. The private variable *lev* keeps the value of *level* at the time root variables are read. To access a bucket, the process checks whether *lev* value matches that buckets *local-level*, and if not, it increments *lev* value and recalculates the address $h_{\text{lev}}(\text{key})$ until a match is found. This operation is called *rehashing*. A process in its search phase behaves as follows: the *root* variables are read and their values determine the hash function to be used initially. The bucket and hash function are updated as follows:

$$
\begin{aligned}
&\text{lev} = \text{level}\\
&\text{bucket no.} \leftarrow h_{\text{lev}}(\text{key})\\
&\text{if bucket no.} < \text{next then}\\
&\quad \{\\
&\quad \text{lev} \rightarrow \text{lev} + 1\\
&\quad \text{bucket no.} \leftarrow h_{\text{lev}}(\text{key})\\
&\quad \}\\
&\text{where } h_{\text{lev}}(\text{key}) = \text{key mod}\,(2^{\text{lev}}N)
\end{aligned}
$$

Attempting to insert a key into a full primary bucket leads to collision. The problem of collision is handled by creating a chain of overflow bucket associated with that particular bucket address. If any bucket overflows, the bucket pointed by the pointer *next* will be split by moving some keys from its original bucket to a new primary bucket to be appended at the end of the hash file. The split operation is applied cyclically. Split operation increases the address space of primary buckets in order to reduce the accumulation of overflow chains. Since the bucket to split is chosen in round-robin fashion, eventually all buckets are split, thereby redistributing the data entries in overflow chains before the chains get to be more than one or two pages long. All bucket chains with addresses less than the value of *next*, and buckets with addresses greater than or equal to $2^{level}N$ have *local-level* values equal to $level + 1$. All the buckets in between have $local\text{-}level = level$, which indicates that they have not efficiently been split in the current round of restructuring.

The *next* pointer travels from 0 to $2^{j-1}N$ with hash function h_j. After the split, the *next* is updated as follows:

$$
\begin{aligned}
&\text{next} \leftarrow (\text{next} + 1)\ \text{mod}(2^{\text{level}}N)\\
&\text{if}\ \ \text{next} = 0 \text{ then level} \leftarrow \text{level} + 1
\end{aligned}
$$

When a primary bucket becomes empty, a merge operation is called. Here, the *next* bucket (that is the bucket pointed to by pointer *next*) will be merged with its *partner* (where $partner = next + 2^{level}N$). That is, keys will be moved from the *partner* bucket to the *next* bucket. After merge the following changes will be made:

$$
\begin{aligned}
&\text{if next} = 0 \text{ then level} \leftarrow \text{level} - 1\\
&\text{next} \leftarrow (\text{next} - 1)\ \text{mod}(2^{\text{level}}N)
\end{aligned}
$$

The goal in (Ellis, 1987) is to allow a high degree of concurrency among processes executing find, insert, and delete operations on a shared linear hash file. In linear hashing algorithm (Ellis, 1987), the find operation can be performed concurrently with find, insert, delete, and split operations. Insert and delete operations may operate in parallel if they are accessing different bucket chains. A split may be performed in parallel with insert and delete operations that are not accessing the particular chain being split. No concurrency is possible between a merge and processes doing find, insert, or delete operations. That is, these processes can neither access the two buckets being merged nor they can read the values of *level* and *next* while merging process is using them. At most, one restructuring operation either merge or split is executable at a time.

Locking protocol in (Ellis, 1987) uses three types of locks mode as shown in Table 1. The primary bucket, and all its overflow buckets are locked as a unit. The find algorithm uses lock-coupled *read-lock* on the *root* variables and holds the lock until a *read-lock* is placed on the bucket. Lock-coupling provides the particular flow of locking in which next component is locked before releasing the lock on the current component. The insert and delete processes hold *read-lock* on *root* variables and *selective-lock* on buckets with lock-coupling. The split operation uses *selective-lock* on the *root* variables as well as on the buckets. *Exclusive-locks* are used on the *root* variables and on both the buckets involved in merge when old overflow buckets are de-allocated. If rehashing is required, a lock is placed on the subsequent bucket before the lock is released on the current bucket. The *read-lock* on the *root* variables held by the searching process prevents a merge from decreasing the size of the address space (by updating *level* and *next*) during the initial bucket access.

2.2. Linear hash structures in nested transactions environment

In Madria et al. (1998), they have presented a nested transaction version of the concurrency control algorithm using a linear hash structure. Their new algorithm increases concurrency and handles transaction aborts. The new algorithm in Madria et al. (1998) has been formalized using I/O automaton model and it is proved that the concurrent linear hash structure algorithm in nested transaction environment is "serially correct". Our implementation verifies the claim aroused by Madria et al. (1998). Fig. 1 presents the structure of our model. The path of the nested transaction tree starts at the user level to the key level consisting of various levels of abstraction.

In the nested transaction tree, we have a level of transaction managers (TMs). Each TM corresponds to an operation invocation. The scheduler, as shown in Fig. 1, directs the user-transaction to the appropriate TM depending on the user's operation request. User's

Table 1
The lock compatibility matrix

Lock request	Existing locks		
	Read-locks	Selective-locks	Exclusive-locks
Read-locks	Yes	Yes	No
Selective-locks	Yes	No	No
Exclusive-locks	No	No	No

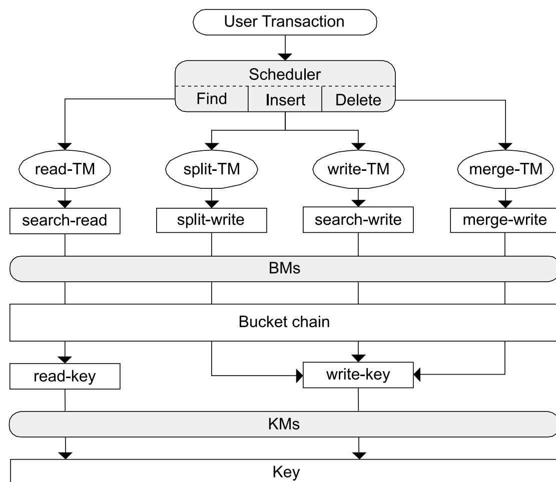

Fig. 1. Linear hash structure in a nested transaction tree.

request can be any of the operations find, insert, or delete. TMs can be thought of as separate, concurrently running processes. Each TM handles transaction-related processing requests like when transactions are created and when they are to be committed or aborted. The find operation is performed logically by an equivalent *read-TM* whereas insert and delete operations are performed by *write-TM*. The *search-read* (for find) and *search-write* (for insert/delete) are access subtransactions created by *read-TM* and *write-TM*, respectively, to accomplish the desired operation. *Search-read* accesses read the target bucket whereas *search-write* accesses physically modify or delete the target bucket. For split and merge operations, a *split-TM* and a *merge-TM* are provided respectively. During an insert operation, an overflow means a split is required whereas a delete operation may account for an underflow, which signals the need for merge. If the root transaction *T* (at the scheduler) intercepts an "overflow" message, it triggers a *split-TM* whereas if it intercepts an "underflow" message, it invokes a *merge-TM*. These TMs invoke access subtransactions of the type *split-write* and *merge-write* respectively to physically accomplish the split and merge operations. When the key is found, new subtransactions will be initiated. The *read-key* and *write-key* accesses will be created to read and write the key respectively. These accesses subtransactions are situated at the leaf level.

On the system building side, when a user requests an operation to be performed, the scheduler creates a root transaction that initiates the corresponding TM to accomplish the desired operation. The scheduler determines the order in which transactions are to be executed. TMs run independently of each other and the scheduler works as a connector and synchronizer for the TMs. In our model, the scheduler holds a lock on the shared variables *level* and *next*, which are referred to as *root* variables. The benefit of locking the *root* variables is due to ensuring the consistency of the variables *level* and *next*, and to the bucket chains. The value of the *root* variables would be returned to the scheduler when a lock is granted to a transaction. This is called lock-coupling protocols given in Ellis (1987). The scheduler receives two types of request from the TMs. A request to create a transaction (*REQUEST-CREATE*) or request to commit a transaction (*REQUEST-COMMIT*). In return, the scheduler responds by creating, committing, or aborting the transaction as shown in Fig. 1. For example, when *write-TM* reports to the scheduler the state of its current transaction, the scheduler will decide whether to commit the transaction or initiate *split-TM* to perform the split operation. The scheduler controls operations of the transactions, makes a decision about the completion of the transaction, reports to their parent (a *REPORT-COMMIT* or *REPORT-ABORT*), and informs objects the fate of the transactions.

When TM receives the transaction from the scheduler, it can then initiate various subtransactions, which are passed to the appropriate bucket containing the needed data. TM receives back the data from the subtransaction, processes the collected data, and may initiate some more subtransactions if needed.

For the transactions to work concurrently in consistency, Madria et al. (1998) have implemented two levels of locking mechanism. The first locking level is called bucket managers (BMs) whereas the other locking level is called key managers (KMs). BMs grant locks to the subtransactions, created by TMs, before accessing the buckets whereas KMs grant locks to the subtransactions before accessing the keys. This is done to ensure that changing of data at a vertex is done in an atomic step to ensure serializability.

A BM handles one bucket and all its overflow buckets (bucket chain). BM provides the status and the lock type of each transaction accessing the bucket. BM handles the transactions by determining the appropriate time to release each transaction or halting it. In addition, BM determines whether to execute one transaction or more depending on each transactions lock type.

Lock management on the *root* variables and on buckets is done using Moss's two-phase locking algorithm (Moss, 1985) and locking algorithm using lock-coupling protocols given in Ellis (1987), while the locks on the keys are managed using read/write lock algorithm (Madria et al., 1998).

3. Architecture of the layered system

We have designed our system as a layered architecture (see Fig. 2) which is compliant with the new model of the three-tier client/server architecture. Each module in the layered system is designed and implemented sep-

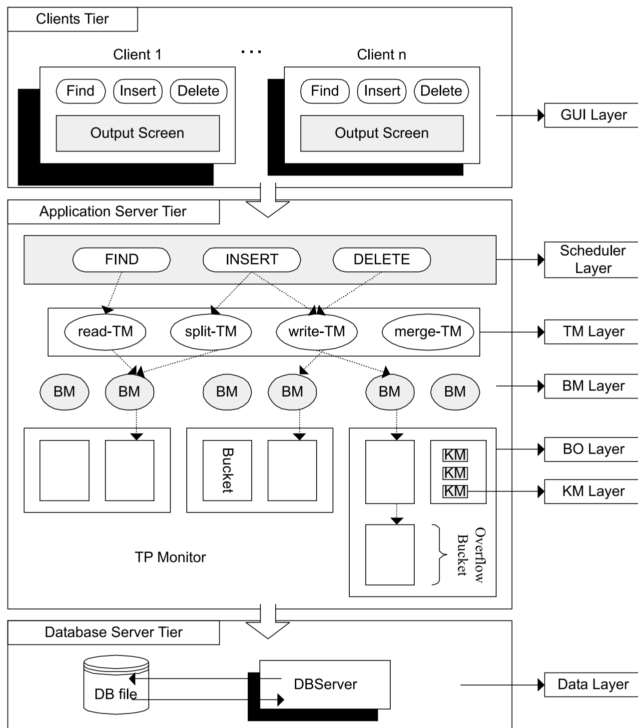

Fig. 2. Layered architecture of three-tier client/server model.

arately. The layered architecture is helpful in the implementation of the nested transactions.

In this section, we will categorize our discussion in three parts. In the first part, we study the structure of our model. In the second part, we discuss how the different components in the system communicate among themselves. Finally, we will discuss the main system components consisting of insert, delete, and find operations.

3.1. Three-tier client/server model

Splitting the processing load is a central client/server concept. Three-tier servers split processing load between (I) clients running the graphical user interface (GUI) layer, (II) the application server (or middleware), and (III) the database server (see Fig. 2).

The architecture of our model consists of seven layers distributed among our three-tier model. One layer at the client side, another one at the database server side, and the remaining layers reside at the application server. The layers are assembled at the application server which does all the heavy work. The layer at the client side is called GUI layer, and the layer at the database server is called data layer. The layers at the application server side are ordered as scheduler layer, transaction processing monitor (TP monitor) layer, TM layer, BM layer, bucket object (BO) layer, and finally KM layer respectively. All layers at the three tiers work in chronological order to achieve the transactions computation.

3.1.1. Presentation tier

Presentation tier deals directly with the end-users. This tier is designed to capture request from the users via the GUI that resides at the client's machine. This tier contains only GUI layer.

3.1.1.1. Graphical user interface layer. In the two-tier client/server, the majority of the application logic runs on the client. This architecture is called *fat client* and it can request data directly from the server-resident database. In three-tier client/server, the client is less engaged as the application logic is moved to the application server. This provides better security by not exposing the database schema to the client. The client in the three-tier client/server is called *thin client*. The *thin client* contains only the GUI. This is the only layer that has been implemented at the client side. It works as an interface between the users and the server. GUI layers task is to capture an operation request from the user and pass it to the scheduler layer at the application server tier. GUI will send a message to the application server containing the key and the type of operation (i.e., find, insert, or delete) upon user has requested.

3.1.2. Application server tier

Application server or middleware consists mainly an important component called Transaction Processing Monitor (TP Monitor), as shown in Fig. 2. TP Monitor acts as an operating system for transaction processing applications. It is also a framework for running middle-tier server applications. The job of a TP Monitor is to guarantee the ACID properties while maintaining high transaction throughput. To do that, it must manage the execution, distribution, and synchronization of transaction workloads. TP monitor ensures that all the updates associated with an aborted transaction are removed or "rolled back". When the resource managers are across networks, the TP Monitor synchronizes all the transactions updates.

To help TP Monitor performs its job efficiently, we have designed the following layers:

3.1.2.1. Scheduler layer. This layer is considered the main layer in the application server as it works as the root of the transaction. The scheduler works as the coordinator in the nested transaction tree. The scheduler layer is the only layer that communicates with the database server layer. When a client sends a request to the middleware, TP Monitor will capture this request and pass it to the scheduler. The scheduler contains three operations; these are *Find*, *Insert*, and *Delete*. The scheduler will direct the request for one of these operations to the appropriate TMs at the TM layer to

accomplish the user's request. The scheduler receives *REPORT-COMMIT* message from the TMs layer when the transaction is successfully committed. The scheduler will then send a message to the database server to update the DB file. After successfully committing the transaction and updating the DB file, the scheduler will inform the TP Monitor that the transaction had been committed. In turn, TP Monitor will send back to the client the result of the operation requested.

The *root* variables' lock, as mentioned earlier, is handled by the scheduler. Before the transaction passed to the next layer (i.e., the TMs layer), the scheduler will check the lock on the *root* variables. The scheduler will grant a lock to the *root* variables whenever a new transaction is received. If the new transaction is compatible with the *roots* lock, it will pass the transaction to the next layer otherwise the scheduler will hold the transaction until the appropriate time.

3.1.2.2. Transaction manager layer. TM layer contains four TMs; *read-TM*, *write-TM*, *split-TM*, and *merge-TM* as explained before. Each TM creates a number of subtransactions to complete the required operation. TMs work concurrently and independently of each others. The scheduler will synchronize transactions created at this layer by controlling the *root*'s variable lock. Subtransactions created by TMs can also work concurrently and only leaf level subtransactions will access the keys directly. TM will send its subtransaction to the BO layer to perform the required operation on the target key. In case of insert or delete operation, a split or merge operation may encounter. In this case, *write-TM* will send a message to the scheduler which in return will initiate *split-TM* or *merge-TM* respectively. This is because TMs do not communicate among themselves which helps in keeping the database consistent. This is due to the fact that split and merge locks are different from insert and delete locks. Hence, to ensure that the *roots* lock is changed when split or merge is encountered, *write-TM* will communicate with the scheduler to grant a suitable lock on the *root* variables according to the requested operation.

3.1.2.3. Bucket manager layer. We consider this layer as a shield layer for the buckets. Subtransactions created by TMs will be granted locks before accessing the buckets. Each BM handles a bucket and all its overflow buckets. Before a transaction accesses any bucket, the corresponding BM will determine whether to permit the transaction to access the bucket or suspend it with the help of the lock compatibility-matrix as shown in Table 1. If two transactions accessing the same bucket interfere with each other, BM will permit one of the transactions to access the bucket while suspend the other until the other transaction finishes. Thus, BMs restrict access to the buckets that lead to an inconsistent database state. BMs work independently of each other.

In our model, we have considered lock on the bucket (page) rather than on the object, as each object in our implementation contains two buckets.

3.1.2.4. Bucket object layer. In our model, each BO contains two buckets. We have considered two buckets in each object rather than one to reduce the systems overhead during split and merge operations. This is due to the fact that when the first bucket of any object say O1 needs to be split, a new object say O2 will be created with two buckets. However, when the second bucket from the object O1 splits, no new object (bucket) has to be created as the keys from this bucket can move to the second bucket of the object O2. Similarly, no object needs to be deleted when one of the buckets becomes empty after a merge operation as the other bucket may contain some data. An object will be deleted to free memory space, only if both the buckets become empty. On the other hand, having more buckets in each object will waste memory space. Thus, having two buckets in an object is a compromise between speediness and space utilization. In addition, we have implemented these buckets to be dynamic which will not waste any space before adding keys to them.

3.1.2.5. Key manager layer. To insure serializability, KM handles the keys lock within the bucket chain. The subtransactions, which run through the KMs, are called decisive subtransactions as they are the only subtransactions that access the keys directly and any interference between transactions at this layer will lead to an inconsistency. KM will insure that there is always one accessible version of the key.

When the key is updated, the data layer at the database server tier will be initiated. When the transaction is successfully committed, scheduler will send a message to the database server to update the DB file.

3.1.3. Database server tier

Our model contains a centralized database server. Our database server is called a *thin server*. This is because the database server is responsible only for storing or loading the data. The database server tier contains only one layer as shown below:

3.1.3.1. Data layer. This is the last layer in our system. Database server insures the durability of the transaction. Any updates on the buckets will be stored in the database file (i.e., DB File shown in Fig. 2) which resides on a permanent disk space. Updating of the database file will be performed after the commit of a top-level transaction (at the scheduler layer) to insure the consistency of our database file. Database server is also responsible for storing and loading the keys from/to the

database file. It can also restore the current database state in case of a system crash.

3.2. Communications via tiers

The communications among various components in our system are done via messages. In the previous section, we have seen how transactions have been used to communicate among various objects. In this section, we will see how the multithreads have been used to implement nested transactions in our system and how these multithreads pass the messages from one object to another.

In our system, multithreads are managed in such a way that they can not interfere with each other. Life cycle of a thread is hidden from clients. During a threads life cycle, many multithreads may be created. A thread may be halted, or an inconsistency may occur, however, these situations will not affect clients. For the clients, a creation and a commit of the top-level thread are important.

As clients operate concurrently, many multithreads will be created. To control these threads from interfering, a locking scheme explained earlier is implemented. The operations in our model will require multithreads to complete their jobs such as searching for the object, inserting, deleting, etc. The arrows in Fig. 3 represent the beginning and the end of the thread route. Each arrow is designated by a number to show the sequence of creation of the multithreads. Only threads at the leaf level will access the data directly.

In Fig. 3, we would like to think of the interface between the clients and the scheduler as a multithreads manager. Multithreads manager is responsible for creating a new thread for each notification of a client's request. Threads created by the multithreads manager are handled by the scheduler which in return pass the thread to the appropriate TM.

To follow a thread cycle, we will take an example of $user_A$'s request as shown in Fig. 3. Suppose $user_A$ initiates a thread to insert a key. $User_A$ will send thread TA_1 to the middleware. Multithreads manager will create a new thread to handle this insert request and pass it to the scheduler. When the scheduler receives the new created thread (i.e., TA_1), it will grant a *read-lock* to the *root* variables. However, before the scheduler grant a lock to the *root* variables, it will check the *root* variables' current lock mode. If the lock formerly granted to the *root* variables is compatible with the new lock, the scheduler will pass the TA_1 thread to the appropriate TM. Otherwise, the scheduler will suspend the TA_1 thread until the lock on the *root* variables becomes compatible with the new lock mode (i.e., *read-lock*). The scheduler then will send the insert-thread TA_2 to the appropriate TM, that is *write-TM*. *Write-TM* will create a subthread TA_3 to search for the target bucket. The search-$write_A$ access subthread will be granted a *selective-lock* before accessing the target bucket. When the target bucket is found search-$write_A$ will pass the thread TA_4 to read the key. The KM will grant a *read-lock* to the read-key_A subthread before reading the key. If the key does not exist in the bucket, search-$write_A$ will pass a new subthread TA_5 to insert the key. Meanwhile, KM will grant a *write-lock* to thread TA_5 before inserting the key. After inserting the key, *write-TM* will report back a

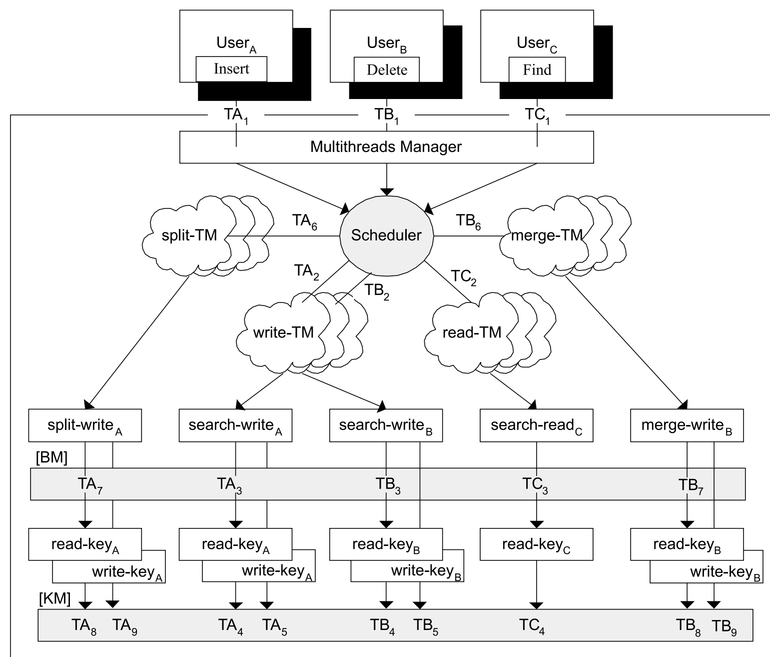

Fig. 3. Multithreading scenario.

transaction-commit to the scheduler, which will inform the TP Monitor the result of the insertion.

In case an overflow occurs, *write-TM* will send a message to the scheduler to inform about the overflow. The scheduler then will create a new subthread TA_6 and send it to the *split-TM*. *Split-TM* will create another subthread TA_5 to access the split-write$_A$ operation. Before the splitting the target bucket, BM will grant a *selective-lock* to TA_7 subthread. Split-write$_A$ will rehash all the keys in the bucket to check whether the key will be moved to the parent bucket or stay at the same bucket. Before rehashing any key, KM will grant a *read-lock* to the read-key$_A$ subthread TA_8. In addition, KM will grant a *write-lock* to the write-key$_A$ subthread TA_9. The same scenario can be followed in case of delete or find operation as shown in Fig. 3.

4. Object-oriented implementation

We have implemented our layered system architecture discussed before using object-oriented methodology since it provides the way for objects to communicate among themselves via methods. Each method invocation corresponds to the initiation of a subtransaction. Each time a method is called, a message is created which is either sent to other object or used by the same object.

A variable of the class type is called an *object*. Any object can be created from a class, and an object is said to be an instance of a class. The main program and the subprograms that oversee the interaction among objects handle the accesses. The control code directs an object to access its data by using one of its methods or operations. A sender passes a message to a receiving object and asks the object to perform a task. The receiving object may return the information to the sender or pass messages to other objects requesting the execution of additional tasks. Each object is designated by an id, called object id (*OID*), to distinguish from other objects.

Our system contains three applications: (1) the client, (2) the application server (middleware), and (3) the database server. The communications among the three applications are done via Windows Socket. The purpose of Windows Socket is to abstract away the underlying network and so the application can run on any network that supports sockets.

Following, we will study each application (i.e., the client, the application server, and the database server) in more details along with their classes. The study of each application contains an explanation of the methods that construct each class. In addition, we will explain how the classes and their methods interact among each others. While implementing, we have considered the efficiency of including a class within another class and the organizing of the classes within the application. To make the discussion clearer and easier, we will support our discussion with more figures and programming codes that will help in understanding the explanation of the classes.

Generally, each application contains four main classes (components). These classes are responsible for:

1. *Initialization*: This class initiates Windows Socket and the main components of the application.
2. *Methods Monitor*: This part is the main component and it contains number of classes that work together to accomplish the application's main task. The name *Methods Monitor* depicts that this part of the application is responsible for calling the appropriate method(s) to do a required task.
3. *Connections*: This class is responsible for the networking between two applications.
4. *Messages*: This class is responsible for passing requests and results from and to the applications.

4.1. Client classes

The four main classes (components) as shown in Fig. 4 are the initialization: *CClientApp*, controlling transactions: *CClientDlg*, Connections: *CClientSocket*, and messages: *CClientMsg*. Our client application is a dialog-window-based program that is easy to interact with the users and better handling notification messages from the other dialog boxes. Following is an explanation of each of these classes in detail:

4.1.1. Initialization

CClientApp is the start up and the base class for the client's dialog interface application. *CClientApp* starts by initiating Windows Socket, by calling *AfxSocketInit* command.

After initiating the Windows Socket, *CClientApp* instantiates an object from the *CClientDlg* class to create the user interface dialog. This dialog box interface receives requests directly from the user or receives the result of the request from the middleware via the network connection.

4.1.2. Methods monitor

This class is the main class at the client site. *CClientDlg* creates Windows Sockets, initiating connection to the middleware, and controlling sending/receiving of the messages to and from the client and the middleware. In general, it manages all the components in the client application. As shown in Fig. 4, *CClientDlg* is responsible for creating the GUI. *CClientDlg* contains four methods that handle the operations of the Windows Socket and the sending or receiving of messages along with the user interface operations:

The *Create&ConnectSocket* method task is to handle the Windows Socket, that is, creating sockets and connecting to the application server. MFC supports programming with the Windows Socket by providing two

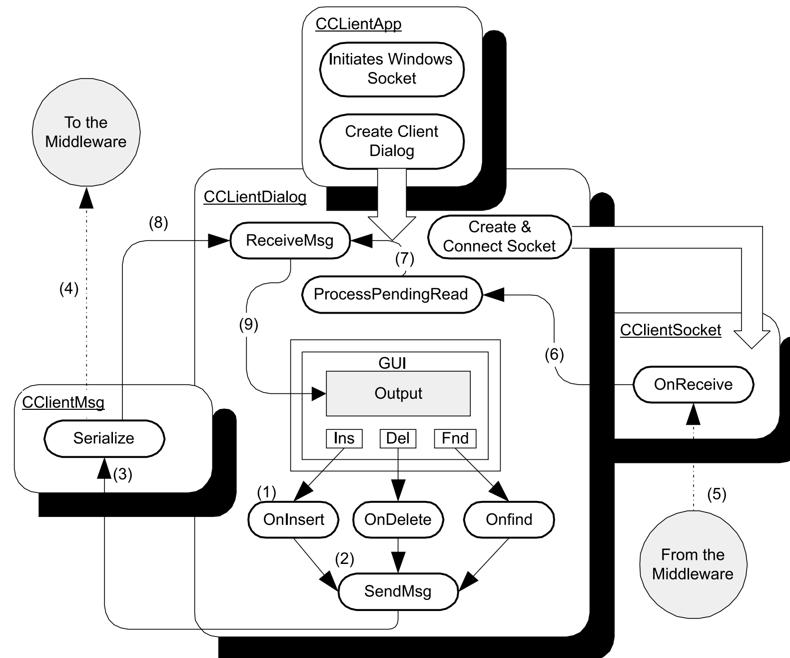

Fig. 4. Client's structure.

classes. One of these classes, *CSocket*, provides a high level of abstraction to simplify the network communications programming. The other class *CAsyncSocket* object represents a Windows Socket—an endpoint of network communication. *CAsyncSocket* encapsulates the Windows Sockets API, providing an object-oriented abstraction for programmers who want to use Windows Sockets in conjunction with MFC.

CSocket works with classes *CSocketFile* and *CArchive* to manage the sending and receiving of data. *CSocket* derives from *CAsyncSocket* and inherits its encapsulation of the Windows Sockets API. A socket is a communication endpoint an object through which a Windows Sockets application sends or receives packets of data across a network. *CSocket* uses a version of the MFC serialization protocol to pass data to and from a socket object via an MFC *CArchive* object. The *CArchive* class allows saving a complex network of objects in a permanent binary form (usually disk storage) that persists after those objects are deleted. Later these objects can be loaded from persistent storage, reconstituting them in memory. This process of making data persistent is called *serialization*. Like an input/output stream, an archive is associated with a file (i.e., the *CSocketFile* class) and permits the buffered writing and reading of data to and from storage. An input/output stream processes sequences of ASCII characters, but an archive processes binary object data in an efficient, nonredundant format. Server socket will create a socket file for sending and receiving data across the network. In the *Create&ConnectSocket* method, there are two *CArchive* variables, one for loading data (*m_ pArchiveOut*) and the other for storing the data (*m_ pArchiveIn*).

SendMsg method sends a message, containing the specified key and the type of operation (i.e., find, insert, or delete) requested by the user, to the middleware via *CClientMsg* class. *SendMsg* initiates Serialize method, from the *CClientMsg*. The function *Flush* ensures that all data is transferred from the memory to the sending/receiving file, that is, the socket file class*CSocketFile*.

ProcessPendingRead is initiated by the virtual *OnReceive* method in the *CClientSocket* class whenever a new message is sent from the application server. The message sent from the application server to the client is the response message of the client's request.

When a new message is received, *ProcessPendingRead* calls *ReceiveMsg* method to read this message.

4.1.3. Connections

CClientSocket works as an interface between the client and the middleware. When the middleware sends a message (response) to the client, *OnReceive* method in the *CClientSocket* class will be initiated. This means that a message from the middleware is sent and it needs to be read. *OnReceive*, then, will call *ProcessPendingRead* to handle this incoming message.

4.1.4. Messages

This class is responsible for the serialization. The *Serialize* method will be called twice in every single user request. First time, it is called in the *SendMsg* at the *CClientDlg* (i.e., *msg.Serialize* (*m_pArchiveOut*)) to

send the data to the middleware. Next it is called in the *ReceiveMsg* at the *CClientDlg* (i.e., *msg.Serialize* (*m_pArchiveIn)) to read the data that came from the middleware.

4.2. Application server (middleware) classes

The middleware resides between the clients and the database server. Thus, the middleware works as a server (i.e., when listen the clients) and as a client (i.e., when connected to the database server). TP Monitor is the component that is responsible for *listening* the clients whereas the scheduler is responsible for connecting the middleware to the database server.

The main component in this tier is the TP Monitor. TP Monitor contains number of components (classes) that help in achieving a transaction's task when spawn into subtransactions. This characteristic increases the concurrency by allowing each component to perform a small part of the transaction. Hence, the component spends less time on each task which allows more transactions to be performed.

Next, we will split the discussion of the middleware to two sections. In the first section, we will study the middleware as a server and in the second section, then we will study the middleware as a client.

4.2.1. Middleware as a server

As we mentioned earlier, one part of the middlewares function is to accept the clients' connections. This is done by *CTPMonitor* class. In this section, we will discuss the components that are responsible for the communication between the clients and the middleware.

4.2.1.1. Methods monitor. *CTPMonitor* is the main component in this tier. It starts by creating the Windows Socket and then initiating *Listen* function so the middleware will be ready to accept the clients connections. In this section, we will study the methods responsible for the connection between the middleware and the clients:

ProcessPendingAccept creates a new client socket and then checks whether the client connection is successful or not. If the client connection to the middleware is accepted, *ProcessPendingAccept* will call *CCLientSocket* to create a new socket file for message passing between the client and the middleware. Eventually, *ProcessPendingAccept* adds the new client to the connection list to keep track of the successful connected clients.

ProcessPendingRead method is initiated when a client sends a new request to the middleware. Consequently, *ProcessPendingRead* calls *ReadMsg* to read this new incoming message.

ReadMsg is responsible for reading the request sent by the client and then directing this request to the appropriate operation that will accomplish the request. As shown in Fig. 5, *ReadMsg* calls *ReceiveMsg* method to fetch the message through serialization. As we mentioned earlier, the message contains the specified key and the type of the operation requested. Depending on the operation type, *ReadMsg* sends the message to one of the three operations in thread manager component to create a thread (transaction) to accomplish the user's request. The three threads initiators are: (1) *CreateInsertThread*, (2) *CreateFindThread*, or (3) *CreateFindThread*.

CTPMonitor is responsible for creating the threads and then directing these threads to the scheduler. *AfxBeginThread* is an MFC function that creates a new thread. For example, creating an insert thread initiates *StartInsertThread* which completes a user's request for inserting a specified key. While *StartInsertThread* is processing the request, *CreateInsertThread* will be free to receive more requests for creating another insert threads. *StartInsertThread* calls *FindAndLock* to check if the key exists or not. In case the key already exits, it sends a message to the client to inform the user that the key is already exists. Otherwise, it calls *InsertToObject* to insert the key. We will explain *FindAndLock* and *InsertToObject* methods in more detail later in this chapter.

UpdateClients notifies the clients when a change in the DB file has occurred. The client is notified after accomplishing the user's request. The client is also notified when the key requested does not exist such as in case of delete or find operation. In addition, if the key is already exist as in case of insert operation, a suitable message will be sent to the client.

4.2.1.2. Connections. *CClientSocket* helps *CTPMonitor* transferring messages between the middleware and the clients. *OnReceive* method notifies the middleware when a new message is received whereas *ReceiveMsg* reads this message. After processing the request, *SendMsg* passes the result of the request to the client via serialization.

4.2.2. Middleware as a Client

Since we have explained in detail about the communication in the client tier, we are not going to discuss the communication of the middleware as a client to the database server. Rather, we will dedicate this section to discuss the implementation of the components (classes) in the middleware that are responsible for managing and executing client's requests.

We will begin our discussion by studying the physical building structure of the buckets in our linear hash file. In our system, the linear hash file starts with two empty buckets. This is because we have implemented two buckets as one object. When a new bucket is needed after a split operation, instead of creating a new primary bucket, a new object will be created which contains two buckets. As we mentioned earlier, this technique is to reduce the

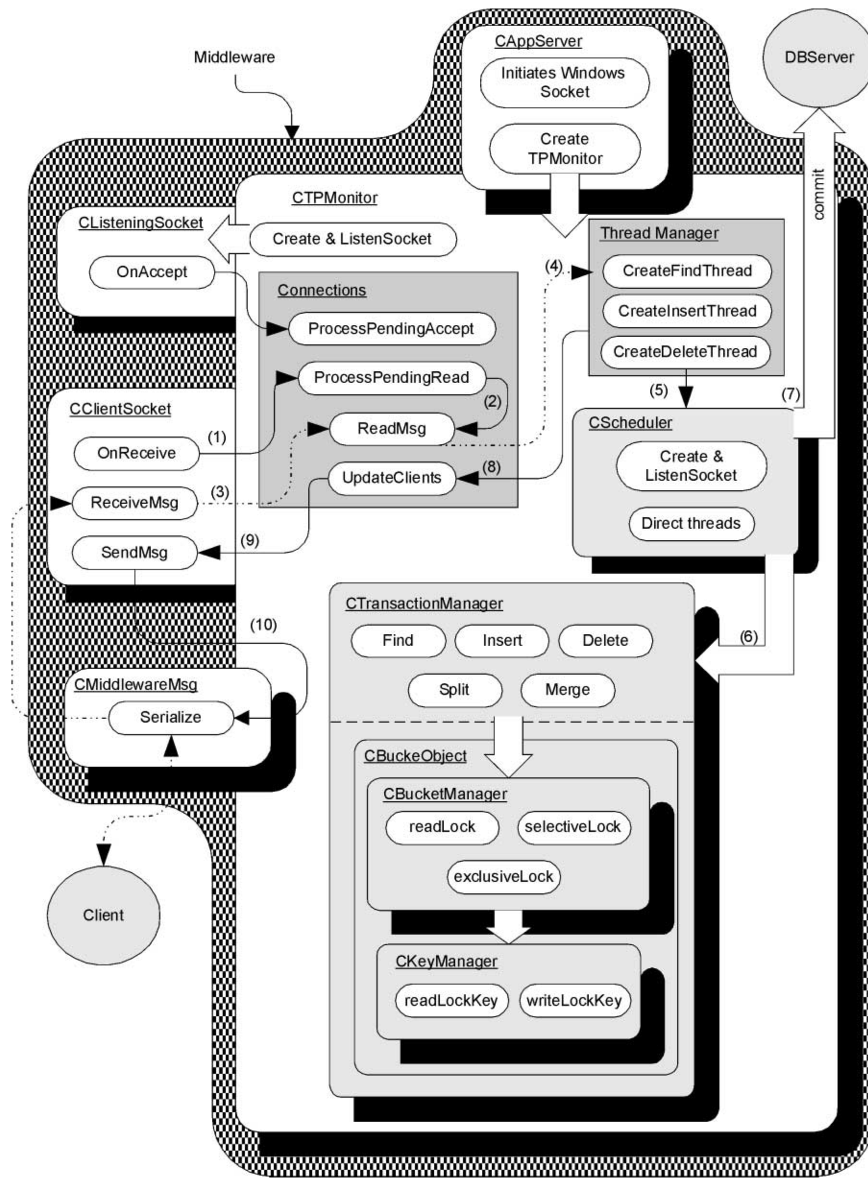

Fig. 5. Application server's structure.

systems overhead during split and merge operations. In addition, there is no need to delete the bucket whenever it becomes empty. Rather, the whole object will be deleted when the two buckets in this object become empty. Hence, the system will have more free resources for the user's request instead of spending this time in managing the linear hash file.

4.2.2.1. Scheduler class. *CScheduler* works as an interface between the middleware and the database server. As *CScheduler* is responsible for sending and receiving data from and to the database server, we like to name the *CScheduler* as a client to the database server. We are not going to explain how the scheduler connects to the database server as we have discussed this earlier, when we studied the connection of the clients to the application server. *CScheduler* considered the top-level (the root) of the transaction (thread). When *CTPMonitor* creates a thread to perform a request by the user, *CScheduler* is the first component to start processing this task. As the transaction's characteristic is that it is performed either in entirety or not at all, *CScheduler* will only send a message to the database server to update the DB file when the transaction is successfully committed. This will ensure the consistency of DB file.

The three methods in the scheduler: (1) *FindAndLock*, (2) *InsertToObject*, and (3) *DeleteFromObject*, will only send the result of the operation to the client after committing the whole transaction. Hence, the DB file is only updated after the transaction (at the scheduler) commits and all its subtransactions (at the lower level) commit, too.

When a user request for an operation, the client will send a message containing the key with the type of the operation required. *CTPMonitor* will direct this request to the *CScheduler* which in turn will search for the key in which object and then in which bucket resides. The searching for the key is done by *FindAndLock* method. Another two methods that exist in the *CScheduler* are *InsertToObject* and *DeleteFromObject*.

To gain access to a specific object, we simply specify the object id (*OID*). For the BM objects, the *OID* is the bucket number as every BM handles one primary bucket and its chain buckets. Whereas, for the TM objects the *OID* can be obtained by dividing current bucket number by number of buckets in one object (i.e, $objectNo = bucketNo/No_Buckets$, where $No_Buckets = 2$). This is due to that each TM object contains one BM object and each BM object contains two buckets.

Following we will study each one of these methods in detail:

FindAndLock method is called for every request of an operation. *FindAndLock* starts by holding a *read-lock* on the *root* variables. After locking the *root* variables, it searches for the appropriate BM object that is associated to the target bucket to pass it the appropriate lock. The lock granted to the bucket depends on the operation type requested. As we mentioned earlier, locking the bucket is done in the scheduler to accelerate the unlocking of the *root* variables.

To search for the key, *FindAndlock* calls the appropriate TM object that contains the target BO. BM locks the bucket before accessing it. If the key is not found in this specified bucket, TM increases the *lev* variable and calculate the value of the *bucketNo* again (rehashing is needed, at this stage, as the target key may be moved to another bucket due to a former split operation). The new bucket is locked before releasing the lock on the current bucket. This process will continue until the key is found or it is the end of search.

InsertToObject method calls *insert-TM* to perform the insert operation. After successfully inserting the key, it sends a message to the database server to save the new inserted key into the DB file. In case an overflow occurred, *InsertToObject* calls *split-TM* to perform the split operation.

DeleteFromObject method calls *delete-TM* to perform the delete operation. After successfully deleting the key, it sends a message to the database server to update the DB file. In case an underflow occurred, *DeleteFromObject* calls *merge-TM* to perform the merge operation.

The garbage collection part of our implementation resides at this method. The technique is that whenever a merge operation occurs, a method called *CheckEmptyObject* checks whether the last object is empty or not. If the last object is empty, then this object will be deleted along with the two BMs handling the buckets. *CScheduler* checks for garbage collection after a merge operation and only after the last object becomes empty, it deletes this empty object. Allowing garbage collection to occur when the last object becomes empty instead of after the last bucket becomes empty, increases the concurrency and expedites operations. This is because more resources will be free, and hence, more transactions can run concurrently.

4.2.2.2. Transaction manager class. Every TM object contains one BO which in turn contains two bucket lists. Each list is a collection of buckets and each buckets capacity is three keys. Every object instantiated from the *CBucket* class is a three keys bucket size. To find a specific bucket we traverse the target bucket chain.

If the target buckets *level* does not equal to *lev* variable, rehashing is needed. This means that the target key was moved to another bucket due to a former split operation, as we mentioned earlier. The *previousBucket* variable is used to keep the previous bucket locked before locking the new bucket (lock-coupling protocol) when rehashing to the new bucket.

To insert or delete a specific key, the same technique will be done as in find operation. The target bucket chain will be traversed and then updating the target bucket. In case of a split operation, the whole bucket chain will be updated and every key in the target bucket within the bucket chain list (i.e., the *next* bucket chain) will be rehashed. If the key, after rehashing, has larger bucket number value than the current one, then the key will be moved to the new bucket chain. Otherwise, the key will remain in its current bucket.

In case a merge operation occurs, the keys in the bucket chain will be moved to the *partner* bucket.

4.2.2.3. Bucket manager class. *CBucketManager* is responsible for managing the locks granted to the transactions that access the buckets before reading or updating them. BMs are implemented as objects. Hence, whenever a new bucket is created, a new BM object will be instantiated from *CBucketManager*, and associated to that bucket. Buckets are locked using the compatibility-matrix shown in Table 1.

The *LockBucket* method checks the current lock mode of the bucket before granting a new lock to the target bucket. In case the current lock mode is not compatible with the new lock mode, BM will wait until the current lock mode becomes compatible with the new lock mode.

4.2.2.4. Key manager class. Every bucket in our implementation can store up to three keys. A key in a bucket is handled by a KM. KM will determine when a key is accessible or not. When updating a key, KM holds a *write-lock* on the key whereas when reading a key, KM holds a *read-lock* on the key.

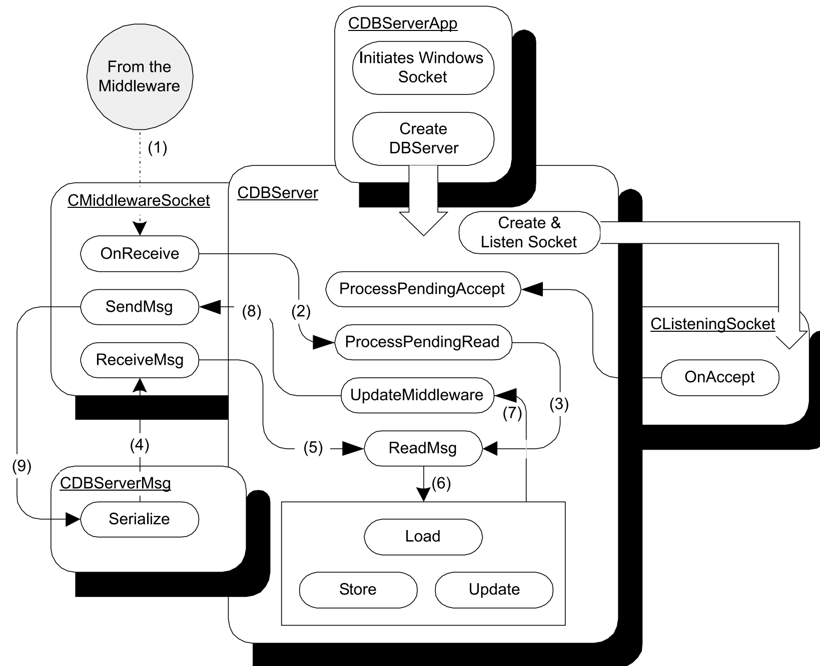

Fig. 6. DBServer's structure.

4.3. Database server classes

Database server tier is responsible for reading and writing the data from and to the DB file. The structure of this tier is shown in Fig. 6. Following are the main classes that handle a transactions operation.

4.3.1. Initialization

The first application that should starts in our system is the database server as it contains the DB file. When the database server starts, the first thing it does is to prepare the networking for the up coming client (i.e., the middleware). Next, *CDBServerApp* prepares the database server for work. As soon as the middleware connects to the database server, it loads the DB file so the objects can be fetched from the memory for faster retrieval.

4.3.2. Methods monitor

CDBSever starts by creating the Windows Socket and then initiating *Listen* function to make the *CDBSever* class ready to receive the client connection (i.e., the application server).

In addition, *CDBSever* contains two types of methods. The first type is the one that is responsible for the connection between the database server and the middleware. The other type is the one that is responsible for managing the DB file. As we explained how the server works while studying the application server, we will concentrate here on studying the methods that are responsible for managing the DB file.

When the middleware connects to the database server, database server loads the DB file and sends it to the middleware. We have used *CFile* class to handle the DB file. *CFile* is the base class for MFC. It directly provides unbuffered, binary disk input/output services, and it indirectly supports text files and memory files through its derived classes. *CFile* works in conjunction with the *CArchive* class to support serialization of MFC objects.

ReadMsg method reads the message sent by the middleware and directs the message to the appropriate operation depending on the type of operation requested:

The two methods that are responsible for reading and writing data are:

1. *Load* method loads a key from the *CArchive* and then sends the key to the middleware,
2. *Store* method stores a key in the DB file.

5. Simulation

In this section, we will study the performance of our system. We have built two system simulators to analyze the performance. These two simulators are the *client simulator* and the *middleware simulator*. A subtle benefit of both the simulators is that they work as the real applications with only minor changes. For the middleware simulator, we have added a *thread performance class* to track the transactions starting and finishing time. In the client simulator, we have added a new method called *Simulator* that will access random keys under random operations. No changes have been made in the other components.

5.1. Simulation preparation

The performance evaluation factors for the simulation are based on the following assumptions:

5.1.1. Assumptions

1. The keys are randomly generated among 10,000 keys.
2. Insert, delete, and find operations are randomly chosen with equal probability.
3. The bucket capacity is fixed to three keys in terms of the number of keys it can hold.
4. Each workstation can run more than one client simulator.
5. A client simulator can run random operations or run only insert, delete, or find operations separately.
6. To test the robustness of the concurrency, we have considered only 500 ms sleep time between each client's request. Hence, more transactions can run concurrently.
7. Only one middleware simulator is needed.

5.1.2. Performance notations and factors

Table 2 lists performance parameters in our simulation:

Table 2
Performance parameters

DB	Number of keys in the database file	F_{ET}	Find elapsed time (ms)
N_C	Number of clients connected to the middleware	NI_{ET}	Not insert elapsed time (ms)
N_{OP}	Number of operations assigned to each client simulator	ND_{ET}	Not delete elapsed time (ms)
N_T	Number of transactions, $N_T = N_C N_{OP}$	NF_{ET}	Not find elapsed time (ms)
N_I	Number of insert transactions	Conc	Number of times concurrency occurred
N_S	Number of split transactions	A_C	Average concurrency, $A_C = \text{Conc}/N_T$
N_D	Number of delete transactions	A_I	Average insertions, $A_I = N_I/N_T$
N_N	Number of merge transactions	A_D	Average deletions, $A_D = N_D/N_T$
N_F	Number of find transactions	A_F	Average finds, $A_F = N_F/N_T$
N_{NI}	Number of not insert transactions, unsuccessful insert	A_S	Average split, $A_S = N_S/N_T$
N_{ND}	Number of not delete transactions, unsuccessful delete	A_M	Average merge, $A_M = N_M/N_T$
N_{NF}	Number of not find transactions, unsuccessful find	N_{BM}	Total number of BMs
ET	Elapsed time in milliseconds (ms)	N_O	Total number of objects
I_{ET}	Insert elapsed time (ms)	N_{PB}	Total number of primary buckets
S_{ET}	Split elapsed time (ms)	Task	Type of operation
D_{ET}	Delete elapsed time (ms)	Start	The time the operation thread start
M_{ET}	Merge elapsed time (ms)	End	The time the operation thread finish

5.2. Simulation results

The simulation was split into two parts. First group of the experiments was to measure the insert operations performance only. The second group of experiments are to measure the performance of the three operations (i.e., insert, delete, and find) running concurrently at random

5.2.1. Simulation of insert operations

Here, we have performed three simulations. Every simulation consists of group of experiments. In each experiment, we have gradually increased the number of insertions. In addition, we have also increased the number of clients from one simulation to another (see Figs. 7–9).

In each simulation, we measure the following four characteristics:

1. *Average concurrency*: We measure average concurrency by counting number of times concurrency occurred in a simulation experiment (i.e., two more insertions occurred concurrently). Then, we divide this number by the number of insertions requested in the simulation experiment. For example, if in a simulation experiment 90 insertions were processed and the concurrency occurred 30 times then average concurrency will equal to 30%.
2. *Elapsed time*: In the simulation tables below, elapsed time refer to the average elapsed time in each simulation experiment. Elapsed time of an operation is the period between the time of requesting an operation to the time of receiving the result of this request. Elapsed time is measured in milliseconds (ms).
3. *Insert vs. split* (*elapsed time*): We have measured the elapsed time of the insert operations and the split operations separately. The objective of this part of the experiment is to prove the claim that our model reduces the overhead system by reducing the number of times a new object is created whenever a split operation is initiated. Although a split operation occurs more frequently than an insert operation, we will see later that the elapsed time of the insert and split operations are almost equal.
4. *Insert vs. split* (*number of occurrence*): Here, we track the average number of times split occurred in correlation with number of insertions and with the size of the DB file.

From the simulation charts below, we conclude the following:

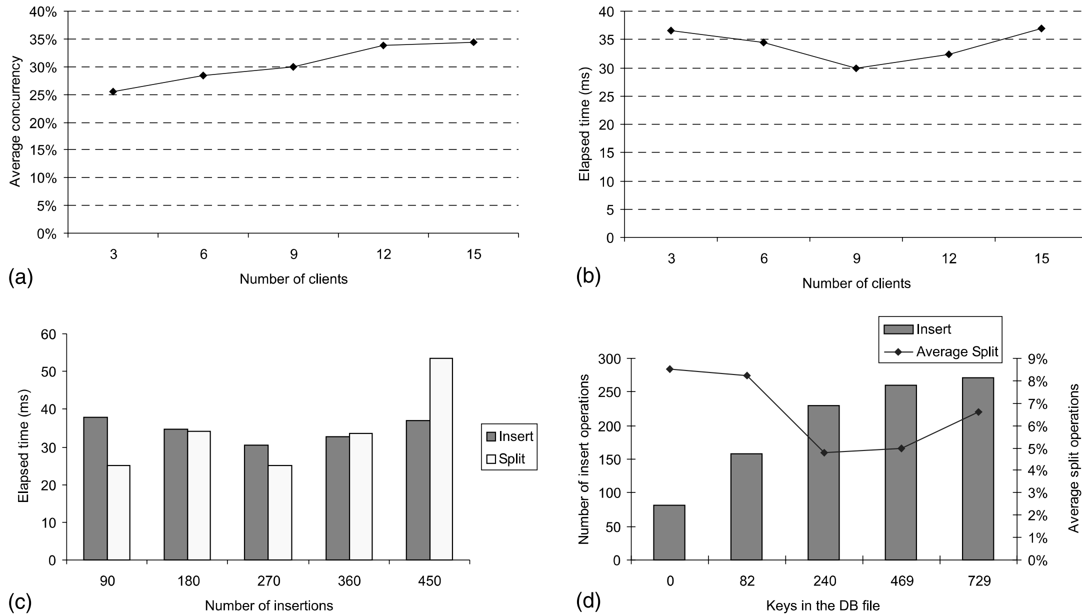

Fig. 7. Simulation 1: (a) average concurrency, (b) elapsed time, (c) insert vs. split elapsed time and (d) insert vs. split occurrence number.

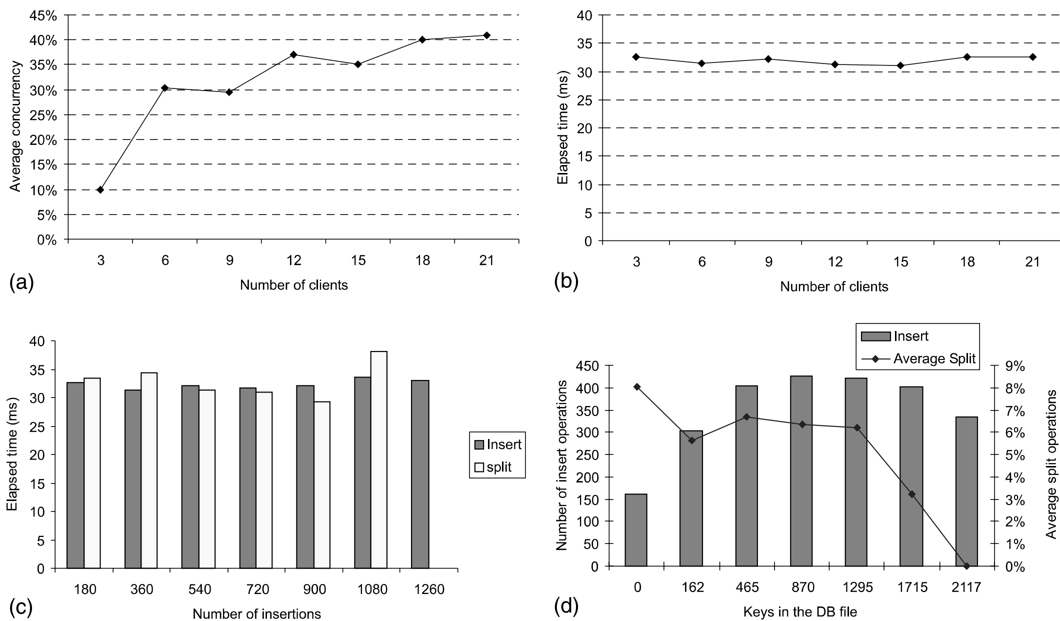

Fig. 8. Simulation 2: (a) average concurrency, (b) elapsed time, (c) insert vs. split elapsed time and (d) insert vs. split occurrence number.

1. From Figs. 7a, 8a and 9a, we have proved that the concurrency increases proportionately with the increasing number of clients. In addition, the increasing number of clients implies an increasing number of insertions. Hence, we can affirm that concurrency also increases proportionately with the increasing number of insertions.
2. From Figs. 7b, 8b and 9b, we find that the elapsed time for the transactions in the insertions simulations does not change significantly. The elapsed time in the three insertion simulations keeps an average of 32 ms. This implies that elapsed time does not affect the increasing number of clients, number of insertions, and the size of the DB file.
3. From Figs. 7c, 8c and 9c, we find that the split operations elapsed time tend to be equal. When the DB size becomes large we noticed that split elapsed time becomes zero, as in Figs. 8c, and 9c. In Fig. 7c, the

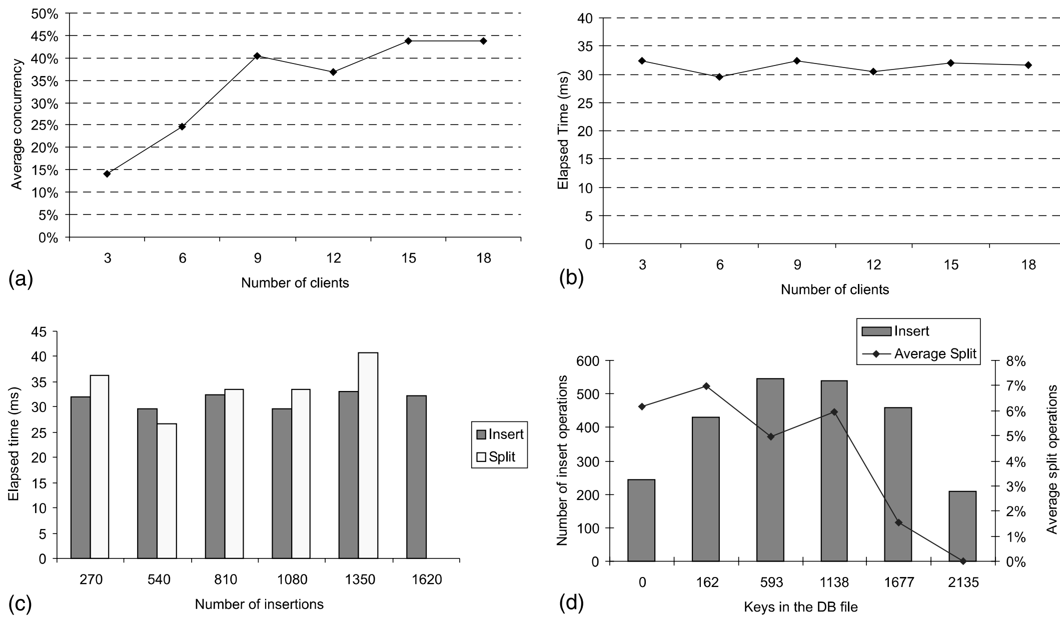

Fig. 9. Simulation 3: (a) average concurrency, (b) elapsed time, (c) insert vs. split elapsed time and (d) insert vs. split occurrence number.

DB size is still small and hence split operations occurs more frequently to compensate due to the lack of the primary buckets.

4. Although in some cases splits elapsed time is a slightly higher than inserts elapsed time, this does not degrade the performance of our system. This is because number of split operations decrease when the size of the DB file becomes large.
5. From Figs. 8d and 9d, we find that number of split decreases proportionately with the increase in the size of the DB file. In Fig. 7a, this is not true. The reason is that split decreases only when there is enough buckets in the memory. As simulation 1 has a small DB file size due to fewer insertions, not enough buckets exist to compromise the number of split operations.

Simulation 1: 30 insert transactions per client

In simulation 1, we have done 5 experiments. The simulation starts with 3 clients in the first experiment and we gradually increase number of clients up to 15 in the last experiment. As we assigned 30 insertions for each client, number of insertions increase with the increasing number of clients. In addition, as we can see from Table 3, the size of the DB file increases from one experiment to the next due to insertions only. So, in simulation 1 DB file starts with zero keys and end with a size of 729 keys.

In Table 3, we have categorized the table to three parts. The first part gives information about the experiment, such as number of clients and number of operations in each experiment. In addition, the first part of the table shows the average elapsed time of the experiment and the average concurrency in this experiment. Second part of the table shows number of occurrence of each operation. That is, we count number of times a specific operation occurred in one experiment. Finally, the last part shows the elapsed time of each operation. Other simulation tables have the same specifications.

Table 3
Simulation 1: 30 insertions per client

Experiment data						Operation occurrence				Elapsed time		
Experiment	N_C	N_T	DB	ET	A_C (%)	N_I	N_S	A_S (%)	N_{NI}	I_{ET}	S_{ET}	NI_{ET}
1	3	90	0	37	26	82	7	9	1	38	25	25
2	6	180	82	34	28	158	13	8	9	35	34	31
3	9	270	240	30	30	229	11	5	30	31	25	27
4	12	360	469	32	34	260	13	5	87	33	33	31
5	15	450	729	37	34	272	18	7	160	37	53	37
Average			–	34	30	–	–	7	–	35	34	30

Simulation 2: 60 insert transactions per client

Table 4
Simulation 2: 60 insertions per client

Experiment data						Operation occurrence				Elapsed time		
Experiment	N_C	N_T	DB	ET	A_C (%)	N_I	N_S	A_S (%)	N_{NI}	I_{ET}	S_{ET}	NI_{ET}
1	3	180	0	33	10	162	13	8	5	33	33	25
2	6	360	162	31	30	303	17	6	40	31	34	31
3	9	540	465	32	29	405	27	7	108	32	31	33
4	12	720	870	31	37	425	27	6	268	32	31	31
5	15	900	1295	31	35	420	26	6	454	32	29	30
6	15	1080	1725	33	40	102	13	3	665	34	38	32
7	21	1260	2117	33	41	334	0	0	926	33	0	32
Average			–	32	32	–	–	6	–	32	33	31

Table 5
Simulation 3: 90 insertions per client

Experiment data						Operation occurrence				Elapsed time		
Experiment	N_C	N_T	DB	ET	A_C (%)	N_I	N_S	A_S (%)	N_{NI}	I_{ET}	S_{ET}	NI_{ET}
1	3	270	0	32	14	244	15	6	11	32	36	34
2	6	540	162	29	25	431	30	7	79	30	27	29
3	9	810	593	32	40	545	27	5	238	32	34	33
4	12	1080	1138	30	37	539	32	6	509	30	33	31
5	15	1350	1677	32	44	458	7	2	885	33	41	31
6	18	1620	2135	32	44	207	0	0	1413	32	0	31
Average			–	31	34	–	–	5	–	32	34	32

Simulation 3: 90 insert transactions per client

As shown in the Tables 4 and 5, we have increased the number of insertions assigned to each client simulator. This is to test our system with more running insertion operations and with larger DB size. Studying the simulation results in Tables 3–5, we conclude that split operations decrease with the increasing size of the DB. As we mentioned earlier, this is because number of primary buckets increase with the increasing size of the DB. This result is very valuable as it supports our claim of building a system that increases concurrency with the increasing size of the DB. This is because decreasing the occurrence of split operations will consequently increase the concurrent operations as no *selective-lock* will be granted to the *root* variables, which is more restrictive than the *read-lock* in case of insert operation.

5.2.2. Simulation of random operations

In this part, we have done three simulations. Each simulation consists of a group of experiments. In each experiment, we have gradually increased the number of transactions assigned to each client simulator. In addition, we have also increased the number of clients from one simulation to another.

In each simulation, we measure the following four characteristics:

1. *Average concurrency*: We calculate average concurrency by counting number of cases where two or more operations run concurrently. Then, we divide this number by the number of all operations assigned to the simulation experiment.
2. *Elapsed time*: It is the elapsed time of one experiment including all its operations.
3. *Insert*, *delete*, *find*, *split*, *and merge operations* (*elapsed time*): In this part, we separate the elapsed time of each operation. This will allow us to compare the elapsed time among different operations.
4. *Insert*, *delete*, *find*, *split*, *and merge operations* (*average number of occurrence*): Here, we count number of times each operation occurred individually and then divide the result by the total number of operations in this experiment.

Simulation 4: 30 random operations per client, DB file starts with 2000 keys

From the simulation tables and charts, we conclude the following:

1. From Figs. 10a, 11a and 12a, we show that average concurrency increases proportionately when number of clients, and consequently with increased number of operations.

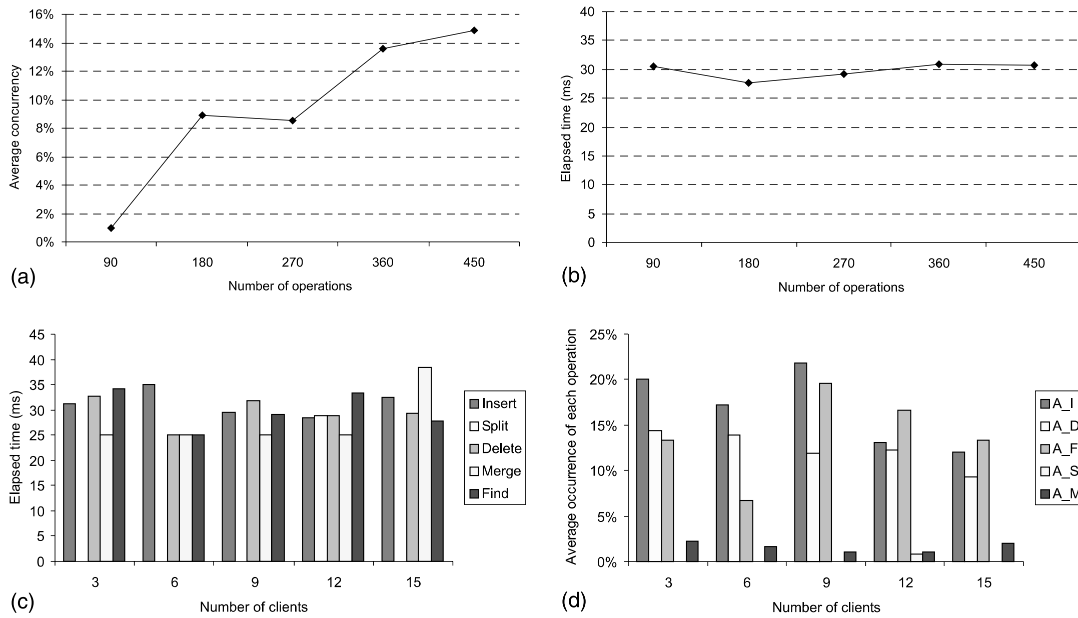

Fig. 10. Simulation 4: (a) average concurrency, (b) elapsed time, (c) comparison between the operations' elapsed time and (d) number of operations occurrence.

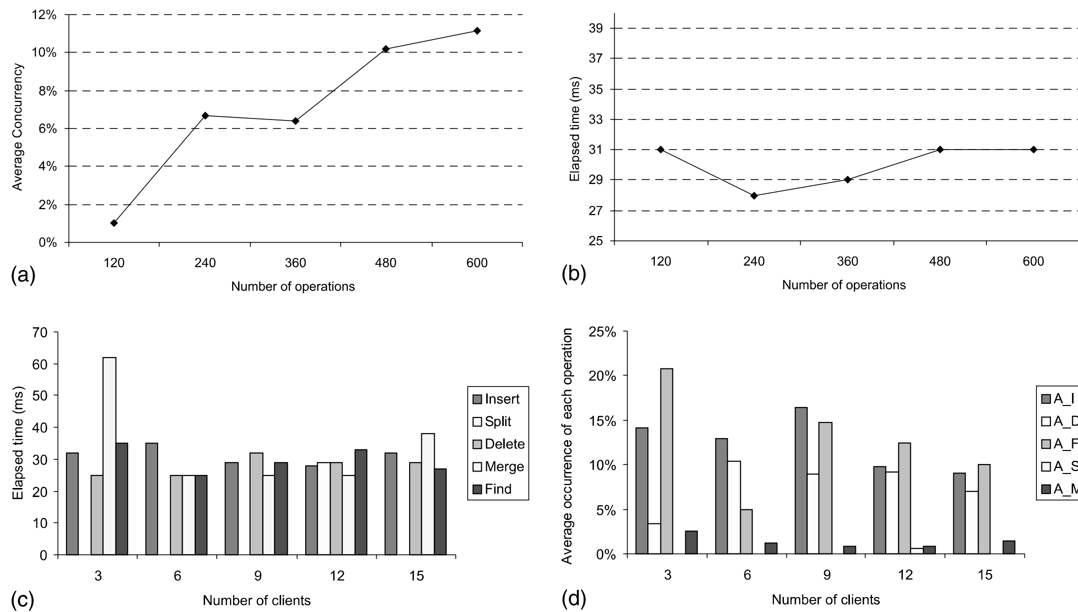

Fig. 11. Simulation 5: (a) average concurrency, (b) elapsed time, (c) comparison between the operations' elapsed time and (d) number of operations occurrence.

2. From Figs. 10b, 11b and 12b, we find that the elapsed time for the transactions in the simulations do not change significantly. The elapsed time in the three simulations keep an average of 30 ms. This implies that elapsed time does not affect the increasing number of clients, number of insertions, and the size of the DB file. Except for only few cases, as shown in Tables 6–8, merge elapsed time is lower than other operations elapsed time. This is because, merge operation suspend other operations from accessing the DB file. Hence, no concurrency occurs while merge operation is in progress.
3. From Figs. 10d, 11d and 12d, we find that the average occurrence of split and merge operations is only 1% to 2% and in many cases it is 0% which proves that our algorithm is ideal for such system. This is be-

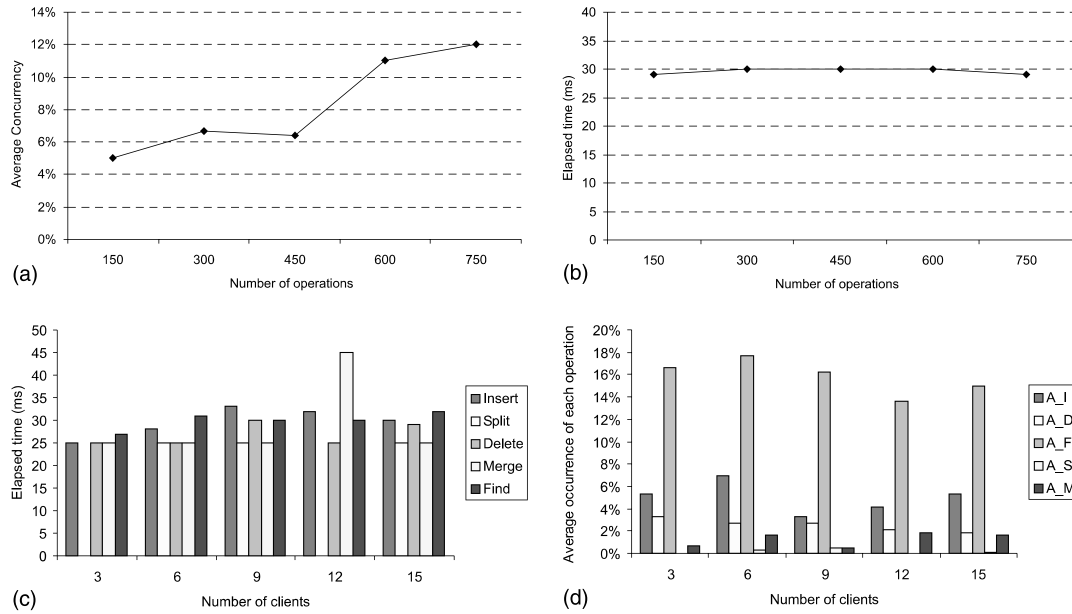

Fig. 12. Simulation 6: (a) average concurrency, (b) elapsed time, (c) comparison between the operations' elapsed time and (d) number of operations occurrence.

Table 6
Simulation 4: 30 random operations—average concurrency (panel A) and elapsed time (panel B)

Experiment	N_C	N_T	A_C (%)	A_I (%)	A_D (%)	A_F (%)	A_S (%)	A_M (%)		
Panel A										
1	3	90	1	20	14	13	0	2		
2	6	180	9	17	14	7	0	2		
3	9	270	9	22	12	20	0	1		
4	12	360	14	13	12	17	1	1		
5	15	450	15	12	9	13	0	2		
Average			9	17	12	14	1	2		
		ET	I_{ET}	S_{ET}	D_{ET}	M_{ET}	F_{ET}	NI_{ET}	ND_{ET}	NF_{ET}
Panel B										
1	3	30	31	0	33	25	34	29	25	35
2	6	28	35	0	25	25	25	28	25	25
3	9	29	29	0	32	25	29	30	28	25
4	12	31	28	29	29	25	33	30	33	30
5	15	31	32	0	29	38	28	30	36	28
Average		30	31	29	30	28	30	29	29	29

cause it reduces the time of creating new objects whenever a split occurs, and deleting garbage objects whenever a merge operation occurs.

4. Elapsed time of the operations decreases from simulation 4 to simulation 6. This is because when the size of the DB is small, fewer objects are created in the memory. Hence, more time is needed to create new objects.

6. Conclusions and future research

In this paper, we have presented a linear hash structure algorithm (Ellis, 1987) in nested transaction environment (Moss, 1985) to expedite concurrency within large amount of data and to handle transaction aborts. We discussed the object-oriented client/server model for the implementation of nested transactions that access linear hash structures. We exploited nested transactions in concurrent linear hash structure to robust the concurrency and handle transaction aborts. We have implemented locking protocol at vertex level using (Moss, 1985) two-phase locking algorithm and locking protocols of linear hash structure algorithm with lock-coupling technique (Ellis, 1987). In addition, we have implemented locking at the key level (Madria et al., 1998) to ensure serializability.

Simulation 5: 40 random operations per client, DB file starts with 2000 keys

Table 7
Simulation 5: 40 random operations—average concurrency (panel A) and elapsed time (panel B)

Experiment	N_C	N_T	A_C (%)	A_I (%)	A_D (%)	A_F (%)	A_S (%)	A_M (%)		
Panel A										
1	3	120	1	14	3	21	0	3		
2	6	240	7	13	10	5	0	1		
3	9	360	6	16	9	15	0	1		
4	12	480	10	10	9	13	1	1		
5	15	600	11	9	7	10	0	2		
Average			7	12	8	13	1	1		
		ET	I_{ET}	S_{ET}	D_{ET}	M_{ET}	F_{ET}	NI_{ET}	ND_{ET}	NF_{ET}
Panel B										
1	3	31	32	0	25	62	35	28	33	25
2	6	28	35	0	25	25	25	28	25	25
3	9	29	29	0	32	25	29	30	28	25
4	12	31	28	29	29	25	33	30	33	30
5	15	31	32	0	29	38	27	30	36	28
Average		30	31	29	28	35	30	29	31	27

Simulation 6: 50 random operations per client, DB file starts with 2000 keys

Table 8
Simulation 6: 50 random operations—average concurrency (panel A) and elapsed time (panel B)

Experiment	N_C	N_T	A_C (%)	A_I (%)	A_D (%)	A_F (%)	A_S (%)	A_M (%)		
Panel A										
1	3	150	5	5	3	17	0	1		
2	6	300	7	7	3	18	0	2		
3	9	450	6	3	3	16	0	0		
4	12	600	11	4	2	14	0	2		
5	15	750	12	5	2	15	0	2		
Average			8	5	3	16	0	1		
		ET	I_{ET}	S_{ET}	D_{ET}	M_{ET}	F_{ET}	NI_{ET}	ND_{ET}	NF_{ET}
Panel B										
1	3	29	25	0	25	25	27	30	28	34
2	6	30	28	25	25	25	31	31	29	32
3	9	30	33	25	30	25	30	29	31	30
4	12	30	32	0	25	45	30	30	30	28
5	15	29	30	25	29	25	32	29	29	29
Average		30	30	25	27	29	30	30	29	31

We used object-oriented technology, which is suitable for client/server systems as it expedites accessing data at the server site, and its property to encapsulate communications from the user. In our model, buckets are modeled as objects whereas linear hash operations find, insert, delete, split, and merge are considered as methods. These methods correspond to nested transactions in their behavior and are implemented as threads and multithreads. Analyzing the performance of the system, we proved that the concurrency increases proportionately with the increasing number of clients. Moreover, the elapsed time does not affect significantly with the number of clients, number of transactions, or size of the DB file. Merging linear hash structure with nested transactions capabilities and employing the object-oriented paradigm under three-tier client/server architecture robust the system's behavior and performance in a centralized database systems.

Another direction of research is to study the model in a distributed environment (Litwin et al., 1996) where a linear hash structure can be spread over number of machines. This involves the studying the benefits of

distributed middlewares with either centralized or distributed databases. Distributed middlewares can handle more clients' requests with less access time.

References

Baeza-Yates, R.A., Soza-Pollman, H., 1998. Analysis of linear hashing revisited. Nordic Journal of Computing 5 (1), 1.

Ellis, C.S., 1987. Concurrency in linear hashing. ACM Transactions on Database Systems 12 (2), 195–217.

Enbody, R.J., Du, H.C., 1988. Dynamic hashing schemes. ACM Computing Surveys 20 (2), 85–113.

Fu, A., Kameda, T., 1989. Concurrency control of Nested Transactions Accessing B-Trees. In: Proceedings of the 8th ACM Symposium on Principles of Database Systems, pp. 270–285.

Hsu, M., Tung, S-S., Yang, W-P., 1990. Concurrent operations in linear hashing. Information Sciences 51 (2), 193–211.

Kumar, V., 1989. Concurrency control on extendible hashing. Information Processing Letters 31 (1), 35–41.

Larson, P.-A., 1988. Linear hashing with separator: a dynamic hashing scheme achieving one-access retrieval. ACM Transaction on Database Systems 13 (3), 366–388.

Lehman, P.L., Yao, S.B., 1981. Efficient locking for concurrent operations on B-trees. ACM Transaction on Database Systems 6 (4), 650–670.

Litwin, W., Neimat, M-A., Schneider, D.A., 1996. LH*—a scalable, distributed data structure. ACM Transactions on Database Systems 21 (4), 480–525.

Lynch, N., Merrit, M., Weihl, W., Fekete, A., 1994. Atomic Transactions. Morgan Kaufman Publishers, San Mateo, California.

Madria, S.K., Maheshwari, S.N., Chandra, B., 1998. Formalization of Linear Hash Structures Using Nested Transactions and I/O Automation Model. International Workshop on Issues and Applications of Database Technology (IADT'98), Berlin, Germany.

Mohan, C., 1993. Aries/LHS: A concurrency control and recovery method using write-ahead logging for hashing with separators. In: Proceedings of the 9th IEEE International Conference on Data Engineering, Vienna, Austria.

Moss, J.E., 1985. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, Cambridge, Massachusetts.

Sagiv, Y., 1985. Concurrent Operations on B*-trees with Overtaking. In: Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Ontario, Canada, pp. 28–37.

Malik Tubaishat has obtained his Master degree in Computer Science from Universiti Sains Malaysia, Penang, Malasyia. He has worked as a Software Developer at Zeine Technological Applications, Amman, Jordan. Currently, he is a lecturer at Yarmouk University, Irbid, Jordan, where he has obtained his Bachelor degree in Computer Science. His research interests are in Nested Transactions, Multilevel Transactions, Concurrency control and TP Monitor.

Sanjay Kumar Madria received his Ph.D. in Computer Science from Indian Institute of Technology, Delhi, India in 1995. He is an Assistant Professor, Department of Computer Science, at University of Missouri-Rolla, USA. Earlier he was Visiting Assistant Professor in the Department of Computer Science, Purdue University, West Lafayette, USA. He has also held appointments at Nanyang Technological University in Singapore and University Sains Malaysia in Malaysia. His research interests are in the area of nested transaction processing, mobile transaction processing and management of web data. He Guest-edited WWW Journal and Data and Knowledge Engineering special issues on Web Data Management. He has served as PC Chair and PC members in many international database conferences and reviewers of many database journals. He has published over 50 Journal and conference papers in the area of web data management, nested transactions and mobile data management.

Bharat Bhargava is a full professor in the Department of Computer Science, Purdue University, West Lafayette, USA. His research involves both theoretical and experimental studies in distributed systems. Professor Bhargava is on the editorial board of three international journals. In the 1988 IEEE Data Engineering conference, he and John Riedl received the best paper award for their work on A Model for Adaptable Systems for Transaction Processing. Professor Bhargava is a fellow of Institute of Electrical and Electronics Engineers and Institute of Electronics and Telecommunication Engineers. He has been awarded the Gold Core Member distinction by IEEE Computer Society for his distinguished service. He has been awarded IEEE CS technological Achievement Award for Adaptability and fault-tolerance in communication network in 1999.