The Use of a Modified Backpropagation Neural Network for Random Access to Data Files on Secondary Storage

Jim Etheredge

University of Louisiana at Lafayette
P.O. Box 41771
Lafayette, LA 70504-1771

Abstract

For many applications random access to data is critical to providing users with the level of efficiency necessary to make applications usable. It is also common to maintain data files in sequential order to allow batch processing of the data. This paper presents a method that uses a modified backpropagation neural network to locate records in a file randomly. The modifications necessary to the backpropagation model are presented. Correlations are drawn between the features of the backpropagation model and the access patterns common to data files. Finally, the performance of the neural network is compared to the B^+ tree indexing method commonly used to provide both sequential and random access to stored data files. The results presented show that the proposed method can provide performance comparable to the B^+ tree depending on the attributes of the file.

Introduction

The B^+ tree and its[1] variants are the most widely used indexing structures for providing sequential and random access to data files. While this indexing method has been proven to be a good choice for random access, it does have some disadvantages. The size of the index increases as the data file grows. There is a cost associated with maintaining the B^+ tree structure as records are added and deleted. The index itself must also reside on secondary storage. Finally, the B^+ tree treats all records in the file uniformly. The cost of accessing any record in the file is the same regardless of its activity. An inactive record occupies the same space in a B^+ tree as the most active record in the file.

This paper describes the use of a modified backpropagation neural network to randomly access records within a sequentially organized data file. The performance of the neural net is compared to the performance of the B^+ tree index based on the average number of disk accesses required by each method to retrieve records from the file. The following sections describe the B^+ tree index method, the modified backpropagation neural network, and the architecture of the testbed. The final sections present the results of the test runs, draw conclusions, and discuss future work.

B^+ tree Indexes

The creation and maintenance of B^+ trees has been well documented in many file structures publications such as Folk [2] and Tharp [5]. The original idea was presented by Knuth [4] in 1973 as a variant of the B-tree structure developed by Bayer (with McCreight) [1] in 1972. Figure 1 shows the general architecture of a B+ tree index and the associated data file. The data records are stored in the terminal blocks of the tree. The terminal blocks are linked together to produce a logical sequential file. This chain of linked data blocks is called the sequence set. The non-terminal blocks in the tree contain key values and pointers to child blocks in the level below. This tree structure is called the index set. Records are accessed sequentially by following the links between sequence set blocks. Records are accessed randomly by following a path from the root of the tree downward until a sequence set block is reached. The sequence set block is then read and searched sequentially for the desired record.

The B^+ tree is created by inserting records into the sequence set. When the first sequence set block fills up, a new block is allocated and the data is divided between the two, maintaining sequential order. In addition, a non-terminal block is allocated and the key value separating the two sequence set blocks is placed in it along with a pointer to each of the two sequence set blocks. New records are placed in the appropriate sequence set block. When a new record is inserted and there is no room for it, a new block is allocated and the data is split evenly between the two. The sequence set linkages are adjusted to accommodate the new block. A new key separating the two blocks is determined and inserted into the parent index set block. If

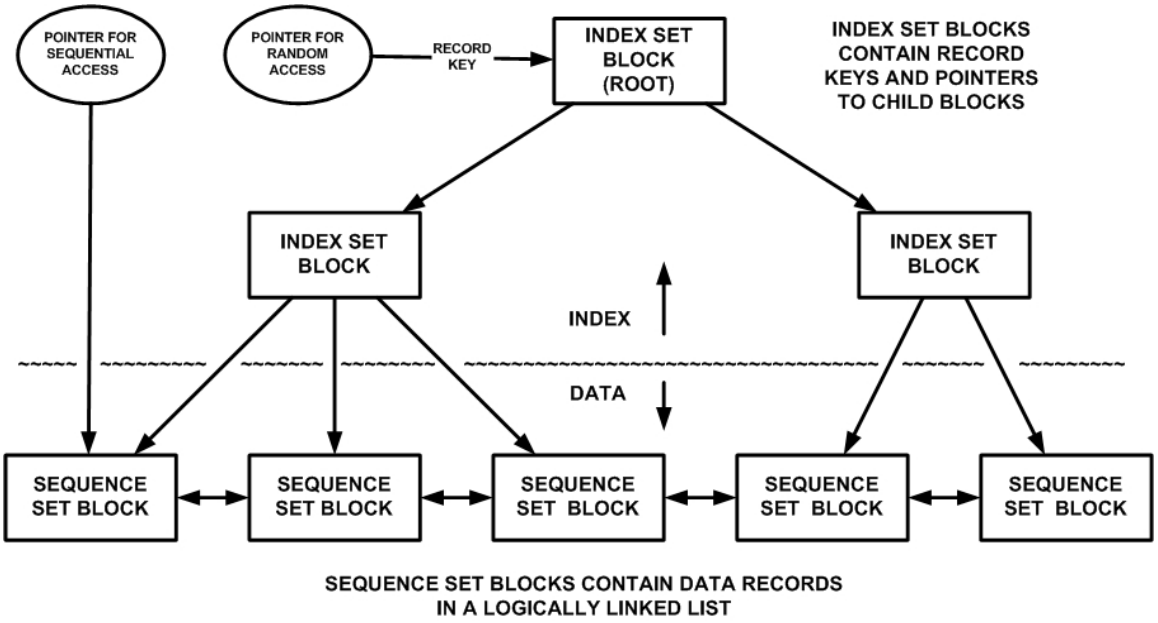

FIGURE 1: B^+ tree Index Architecture.

there is no more room in that block, another index block is allocated and the key/pointer pairs are divided equally between the two. The middle key is promoted to the parent block in the index set. Thus, insertion of a new record into the file can potentially cause a split in the sequence set block and an associated split in the index set at every level of the tree back up to the root. When the root splits, a new root is created and the tree increases in height by one. The doubly-linked list implementation of the sequence set blocks shown in Figure 1 is necessary to allow backward as well as forward searches when locating records using the neural network.

Every block in the B^+ tree (except possibly the root) is always at least half full. The height of the tree is a function of the number of records in the file, the size of the record keys, and the number of key/pointer pairs that can be stored in an index block (called the fan-out factor). Deletion of records from the file has the reverse effect on both the sequence and index sets.

The Modified Neural Network

It is assumed that the reader is already familiar with the backpropagation neural network model. The discussion presented here focuses on the modifications required to use it as a random access method.

The neural network has 30 nodes in the input layer, and 15 nodes in the output layer. There are two hidden layers with 29 and 25 nodes respectively. The neural network is fully connected with a learning rate of 0.9 and a momentum rate of 0.7. Record keys and block numbers are encoded as shifted binary (0 => 0.1, 1 => 0.9)

The neural network developed for this application deviates from the standard model in three important ways. First, there is no separate training phase prior to using the neural network. Every time a data record is inserted into or retrieved from the data file, its key and the sequence set block number where the record resides are presented to the neural network. The difference between the actual output of the network and the target output is used to train the network. The training and utilization phases are combined into a single continuous process. Second, overfitting is no longer considered an undesirable condition. In fact, memorization of record keys and their location in the sequence set is the goal of the neural network. Finally, each time a record is accessed, it is presented to the neural network multiple times to speed the learning process.

The B^+ tree/Neural Network Testbed Architecture

The testbed is a combination of a B^+ tree and a modified neural network. The B^+ tree code was obtained from Jannik [3] and modified for use with the neural network. Both the B^+ tree and the modified neural network are written in C++. Figure 2 shows the architecture. Records are created using a random number generator to produce nine digit numbers. These numbers represent data record keys in the file. The keys are inserted into the file using the B^+ tree. This is necessary since the B^+ tree index is created as a byproduct of the insertion process. Once the file is created, the same set of random keys is used to simulate record access in the file. Any record that is either inserted into or retrieved from the file is also presented to

the neural network as a pair of values: the record key and the sequence set block number where the record is actually stored. The input key is propagated through the neural network and the error is propagated back through the neural network. The performance of both methods is measured in terms of average accesses to retrieve records over time. The number of accesses required to find a record using the neural network is the distance, in blocks, between the actual block and the output of the neural network. The number of accesses required to locate a record using the B^+ tree is equal to the height of the tree.

The neural network configuration and current weights are saved to a file when the test run ends. This allows multiple test runs to have a cumulative effect on the ability of the neural network to locate records in the file

The Test Plan

The primary goal of the project is to evaluate the viability of using a neural network as the primary method for providing random access to records in a data file. In large data files it is likely that a large percentage of the records see little, if any, activity. Since the neural network is trained on every record accessed, it gives preference to (remembers) the more active records in the file and gives less consideration to (forgets) records with less activity.

The test plan involves comparison of the performance of the neural network to that of the B^+ tree in the random access of records in a data file. The B^+ tree index serves both as a mechanism for creating the data file and a baseline for the performance of the neural network. Various combinations of file sizes and access patterns were tested. Due to space limitations, only two of the test runs are shown. Both of these runs used a randomly generated file of 1000 records. In one run, each time a record was accessed it was presented to the neural network 100 times. In the other run, each time a record was accessed it was presented to the neural network 200 times.

Results

Figures 3 and 4 show the results of the two test runs. In each case, the top X% of the records receive 100 - X % of the activity. For example, the top 20% of the records in the file receive 80% of the activity. This simulates the assumptions that a small percentage of the records have a high percentage of the activity. The last two digits of the record key are used to assign the record to an activity group. Records with 00 are the most active and records with 99 are the least active. For each test run, average file accesses are shown for the most active 20% of the records, the least active 20%, and the overall performance. The difference between the runs shown is the number of times each key/block pair is presented to the neural network. The values used were 100 and 200 times respectively. The graph in Figure 4 also shows the average accesses for the

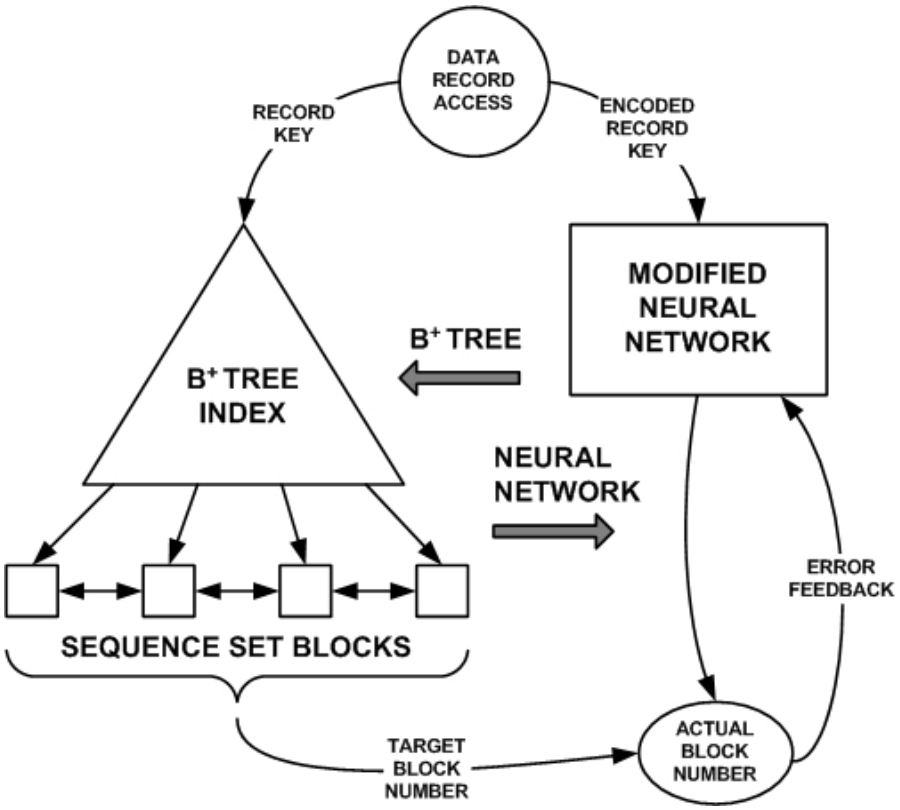

FIGURE 2: Testbed Architecture.

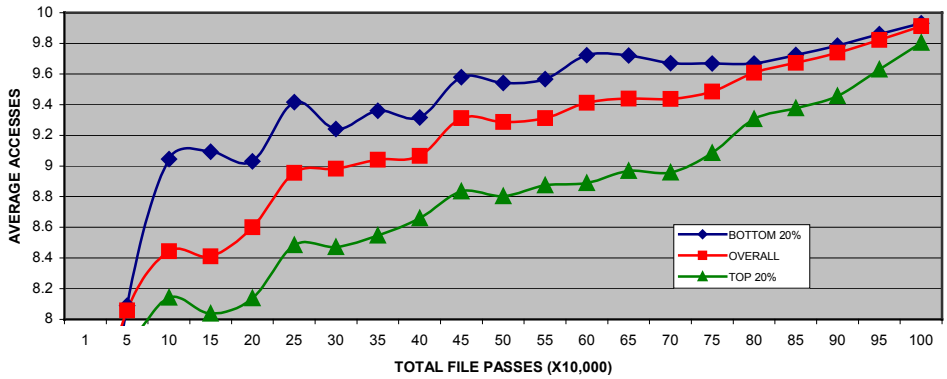

FIGURE 3: 100 Presentations Per Access

B^+ tree. The x axis in the graphs indicates the number of times the file is processed during the test run (times 10,000).

Conclusions

The results presented in the previous section show that over time the neural network becomes better able to predict the location of a data record within the file. The results show that the performance of the neural network improves as the number of record accesses increases and the active data record percentage decreases. When the key/block pairs are presented to the neural network multiple times, the performance improves dramatically. At 100 presentations per access, the average accesses range between 8 and 10. At 200 presentations per access, the neural network requires fewer average accesses than the B^+ tree to find the more active records in the file. In both figures, the records with the most activity (top 20%) require fewer accesses than the overall, and the least active records (bottom 20%) require the most accesses. Figure 4 also includes the average accesses for the B^+ tree. Since the testbed architecture does not provide for the deletion of records from the file, this value is a constant, in this case 5, and is equal to the height of the B^+ tree.

The graphs also show that the average number of accesses increases with the number of file passes. This is because the low activity records are presented to the neural network more often as the number of file passes increases. The more records the network has to remember, the less accurate it becomes. However, at the 200 presentations per access level, the performance of the neural network is still better than the B^+ tree.

In general, it appears that for certain types of data files, the use of a neural network provides performance comparable to that of B^+ tree indexing without the

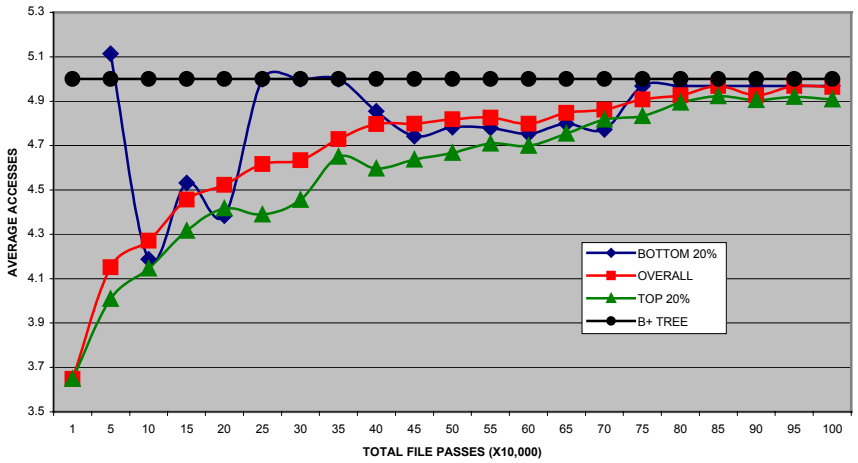

FIGURE 4: 200 Presentations Per Access

additional overhead incurred for storage and maintenance of the index.

Future Work

Future work on the project will focus on extending the capability of the neural network to provide efficient random file access.

It is anticipated that larger file sizes will saturate the current configuration. Other network configurations will be tested to accommodate larger file sizes. It is also possible that manipulation of the learning rate could be used in combination with multiple presentations to speed the memorization of record locations.

The testbed will be modified to include the deletion of records from the file. This will allow testing of the neural network's performance under more realistic file update conditions. This will also allow for more accurate assessment of the actual cost associated with the maintenance of the B^+ tree index.

Acknowledgments

The author would like to acknowledge the invaluable help of Mr. Matt Price. The implementation of the modified neural network was undertaken by Mr. Price as a student assistantship under my direction during the summer 2003 term.

As with any implementation of this scale, the success of the project depends to a large extent on the skill and motivation of the programmer. Mr. Price demonstrated an exceptionally high level of skill and dedication.

References

[1] Bayer, R. and McCreight, C., "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, Vol. 1, No. 3 (1972), pp. 173-189.

[2] Folk, M.J. and Zoellick, B., *File Structures (2nd Ed)*, Addison-Wesley Publishing Company, Inc., 1992.

[3] Jannink, J., B+ tree code obtained from URL: http://www-db.stanford.edu/~jan/jan.html, posting date 1995.

[4] Knuth, D.E., *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[5] Tharp, A.L., *File Organization and Processing*, John Wiley & Sons Inc., 1988.