METHODS FOR DISTRIBUTING A B\(^+\) TREE IN A LOOSELY COUPLED ENVIRONMENT

by

G. Matsliach and O. Shmueli

Technical Report #594

December 1989
METHODS FOR DISTRIBUTING A B⁺ TREE IN A LOOSELY COUPLED ENVIRONMENT

Gabriel Matsliach     Oded Shmueli

TECHNION – Haifa, Israel.

abstract

We consider the problem of maintaining a data file which must be distributed among disks, each controlled by a processor and residing in a different site. The pairs, of disks and processors, are connected via a local broadcast network. A simple and practical method for maintaining such a file is presented and analyzed.

Keywords: Distributed data structures, B⁺-trees, loosely coupled environments.
1 Introduction

There has been much work on data structures for secondary memory [3], [8], [11], [16], [17]. This work was mainly done in the context of traditional systems, in which memory and processing power were limited. Modern systems possess an abundance of resources: memory and processors. The latter appear as either loosely coupled or tightly coupled.

The idea of using parallelism for achieving higher performance is not new, for example see work on parallel database machines [2], [5], [6], [15]. In the context of data structures, there has been work on parallel access to shared data structures such as B-trees and Hash tables, for example [7], [13], [14]. However, little work was carried out on distributing the data structures themselves. We consider a special case in which a large file (that does not fit into one disk) must be distributed among several disks. Each disk is controlled by a processor, and each pair, of disk and processor, resides in a different site; the sites are connected via a local network.

Many file systems use variants of B-trees, usually B+-trees. Our goal is to design a file system, based on a B+-tree, in a multiprocessor environment with P processors which are located in different sites and connected via a local network. We consider the following standard operations:

- **SEARCH**: Searching for a record given its key.
- **UPDATE**: Updating an existing record.
- **INSERT**: Insertion of a record.
- **DELETE**: Deletion of a record.

*Research supported by M. Lutin Fund for Laboratory Research.
• **SUBRANGE**: Producing a list, in key order, of all records in a subrange of keys.

The first method we consider for distributing a B+•tree is called the *Straight Forward Method* (SFM for short). SFM *equally* partitions the records among the processors, and each processor stores its records in a local B+•tree (each tree node spans a disk page). Such a distribution of keys among the processors may obey the following *distribution rule*:

If \( i > j \) then the key values of all records of processor \( i \) are greater than the key values of the records of processor \( j \).

This rule ensures that no merge operation is necessary to produce a result for a *SUBRANGE* operation. However, in order to *equally* partition the records among the processors so that the distribution rule is obeyed, statistics on the distribution of keys is needed. Such statistics may be difficult to obtain in practice, and therefore the records are usually (equally) partitioned among processors using round-robin or a hash function (and thus the distribution rule is not obeyed). For example, GAMMA, a parallel Database machine [6] consists of 20 processors which are connected via a token ring network, distributes the indices among its processors according to the straight forward method described above.

A family of methods for utilizing many small disks, RAID levels 0-5, is reported in [9]. One scheme (which is used by the *Connection Machine* supercomputer) involves 32 disks, each storing one bit of each 32 bit word. In addition, 8 more disks contain error correcting bits and one additional disk is on standby to replace a failing disk. Each access to the simulated large disk is performed by accessing in parallel all 32 small disks and reading their pages to an array of pages in cache. This is essentially a simulation of a large disk containing large pages by small disks containing small pages. A sequence of improvements on this general idea is presented in [9].

The methods of [9] can obviously be used to distribute a B+•tree among processors. However, in a loosely coupled environment, this results in a communication overhead since each time a tree node need be accessed, all disks are requested to send their parts of the node to the host computer, which operates on the node and distributes it back to the disks. We later on refer to this method as PDM (Physical Distribution Method). From the above discussion, it seems that PDM is more appropriate for a tightly coupled environment in which special hardware can handle the simultaneous accesses to the disks.

We also use parallelism by distributing the tree among the processors. Conceptually, we use a “wide” B+•tree, which is called the *conceptual tree*, and its distributed implementation which is called the *physical tree*. The method is called a *Totally Distributed B+•tree* (TDB for short). TDB is based on the following idea: the *entries* (keys and pointers) of each node in the conceptual tree (called a *conceptual node*) are partitioned among the processors. In TDB, each disk is accessed by a different processor. In other words, each processor accesses *only a part* of each large simulated page and, in order to work properly, does not need to “see” the other parts of the page. Therefore, TDB is more appropriate in a loosely coupled environment in which each processor-disk pair resides in a different site. We note that TDB is also appropriate for tightly coupled environments (in which the communication among processors is done via a shared memory).
The rest of this abstract is organized as follows. In section 2 we describe the physical environment. In sections 3 and 4 we describe TDB, and in section 5 we demonstrate that TDB is scalable. A performance comparison, with respect to a single operation and a sequence of operations, is presented in sections 6 and 7.

2 The physical environment

The physical environment consists of a host computer (host for short), P processors (numbered 0...P-1) and P disks. Each disk is controlled by exactly one of the P processors. Each pair, of processor and disk, resides in a different site, and all pairs are connected via a local, Ethernet type, network. The interaction between the host and the processors, and among the processors, is via the local network. In this network, each message sent by a processor, can be "heard" by all processors. Therefore, the cost of a message that is directed to a specific processor (by writing its identification number on the message), and the cost of a message that is sent to all processors, are essentially the same. A diagram of the physical environment is presented in figure 1.

Each processor mainly operates on its own disk. In some cases a processor might need to access another disk. This is done by requesting the desired information from the processor in charge of that disk. When an operation is requested from the host, the host sends it to all the processors, and the processors execute in parallel (with some interactions) in order to produce the results. The processors send the produced result(s) back to the host.

3 The conceptual tree

The B+-trees used by SFM are defined as follows. In a B+-tree of degree M, each internal node contains no less than M/2 pointers and at most M pointers. The number of keys in an internal node is one less than the number of pointers in it. The leaves contain an equal number of pointers and keys. Therefore, this number is no less than (M/2)−1 and
at most $M-1$, and space for one pointer is unused. (Note that the precise leaf content is usually left unspecified in defining a $B^+$-tree. We have fixed a convention to facilitate calculations). The pointers in the leaves point to the actual data records. Thus, the tree is a dense index [16]. All leaves are doubly linked from the leftmost leaf (which contains the lowest key in the tree) to the rightmost leaf (which contains the highest key in the tree). This description implies that one can operate on the internal nodes and on the leaves in, essentially, the same way.

The conceptual tree is a $B^+$-tree of degree $E$ (as defined above) with the following exception: each conceptual node contains two additional keys. In a conceptual internal node, these keys are the minimum possible key value and a key larger than the maximum possible key value in the subtree rooted at the conceptual node. In a conceptual leaf, one additional key is a key larger than the maximum possible key of any record which is pointed from this conceptual leaf, and the other additional key enables the leaf to contain an additional entry. (The two additional keys of each internal node and one of the additional keys of a leaf may be omitted in practice. They are added so that all parts of a conceptual node (which are held by processors) have identical structure). We conclude that each conceptual node contains space for up to $E+1$ keys and $E$ pointers. Thus each conceptual node may contain no less than $E/2$ pointers and up to $E$ pointers. (Recall that the leaves' pointers point to the actual data records). To simplify the presentation, we assume that $E$ and $P$ are even.

4 TDB (Totally Distributed $B^+$ Tree)

In TDB, the conceptual tree is distributed among the processors by assigning each processor an exclusive part of each node in the conceptual tree. Therefore, each conceptual node is partitioned among the processors holding the physical tree. All processors, together, implement the physical tree. Each part of a node in the tree is called a cut because it can be thought of as obtained from the conceptual node by cutting it at two points. Each processor's cut spans a disk page and therefore a conceptual node spans $P$ disk pages.

All the processors hold their cut of a particular node at the same actual address. Each processor can access only its cut of each node. During key search, this enables a processor, holding a portion of a conceptual node $v$, to provide an address for continuing the search through a child of $v$, even though the cut, through which the search should proceed, is in a different processor. The requirement of same actual address can be replaced by a requirement of same virtual address. This may result in experiencing non-uniform seek time. However, by providing an interrupt mechanism, with which a processor that "knows" how to proceed notifies the others to stop searching their cuts; performance is dominated by the time it takes, for the "right" processor, to determine where to "go down".

Naturally, there will always be exactly one processor which can determine the address of the next node while searching in the physical tree. This implies that a search in the
FIGURE 2a: A conceptual tree (of degree 4).
(a) The root (kept in the host's main memory).

(b) First cut (processor 0).

(c) Second cut (processor 1).

FIGURE 2b: The physical tree (two processors).
physical tree is done cooperatively by all the processors. In figure 2 we present the general idea: figure 2a presents a conceptual tree of degree 4, figure 2b presents the physical tree which is held by two processors. The root of the conceptual tree is held in the host's main memory, thus, the root is not partitioned among the processors.

Reading a node in the conceptual tree means that each processor reads its cut of the node in the physical tree. We would like that all accesses to the disks will be done simultaneously as if only one large disk is accessed. The environment assures us that indeed all accesses can be done almost simultaneously because each processor reads from its dedicated disk. It is possible though, that an actual data record is not in the disk of the processor whose cut, of a leaf, contains the pointer that points to that record. This has to do with a decision not to move large data records between processors (more details are in the appendix).

The algorithms that implement TDB operate on the physical tree. Certain structural features are required; these features and algorithms are presented in the appendix. Although we use the variant of a B+-tree whose leaves contain entries, TDB can be adapted to the variant in which the leaves contain the actual data records.

5 Adding processors and disks to the system

When files become larger, their conceptual trees grow taller and thus performance deteriorates. In order to improve performance, one may add more pairs of processors and disks to the system. TDB is flexible enough to improve performance using these new pairs. Assume that X processors (numbered P ... P+X-1) are added to the existing P processors (numbered 0 ... P-1). Thus in TDB, each conceptual node becomes wider and spans P+X disk pages. The new processors' cuts of the conceptual nodes (which are the rightmost cuts of the nodes) will initially be empty. This may result in conceptual nodes which are less than half full. We dynamically solve this problem by concatenating such nodes, while accessing them due to operations on the file and during idle times.

6 Performance comparison of a single operation

SFM equally distributes the records among the processors (see section 1). Each processor keeps its records in its own "local" B+-tree. Each such B+-tree is as defined in the first paragraph of section 3, and thus each tree node spans one disk page. Recall that each conceptual node of TDB spans P pages. TDB is better than SFM in terms of maximum retrieval cost and minimum, maximum, and average capacities. These results are intuitive as the degree of the conceptual tree (of TDB) is almost P times that of the degree of each of the P trees of SFM, and due to the exponential effect of the degree (of the tree) on capacity and height. Therefore, our analysis, regarding these claims, is omitted.
The performance of TDB and PDM\(^1\), in terms of retrieval time and capacities, are essentially identical as they both implement a conceptual tree. However, we show that TDB is better than PDM as the communication overhead of TDB, in terms of total size (in bits) of messages sent per operation, is much smaller than the respective cost for PDM. Recall that the cost of a message being sent to all processors is essentially the same as the cost of a message sent to a specific processor, as all processors are connected via a broadcast local network. Also recall that in PDM, an access to a conceptual node is done by requesting all processors to send their parts of the node to the host, and if the conceptual node is changed, then all the parts that were changed are sent back to the processors. We examine the communication cost when searching, inserting, and deleting a record. Throughout the calculations, we omit the constant cost of putting the messages on the network, and we denote by \(H\) the height of the conceptual tree. The following tables summarize the costs of TDB and PDM:

### TDB

<table>
<thead>
<tr>
<th>op</th>
<th>The cost of TDB in number of bits sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>(H \cdot (\text{pointer.size}))</td>
</tr>
<tr>
<td>Insert</td>
<td>(H \cdot (\text{pointer.size}) + X \cdot P \cdot (\text{counter.size} + \text{block.size}) + P \cdot (\text{counter.size} + \text{entry.size}))</td>
</tr>
<tr>
<td>Delete</td>
<td>(H \cdot (\text{pointer.size}) + X \cdot P \cdot (2 \cdot \text{counter.size} + \text{block.size}) + P \cdot (2 \cdot \text{counter.size}) + (P/2) \cdot \text{entry.size})</td>
</tr>
</tbody>
</table>

### PDM

<table>
<thead>
<tr>
<th>op</th>
<th>The cost of PDM in number of bits sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>(H \cdot P \cdot (\text{block.size}))</td>
</tr>
<tr>
<td>Insert</td>
<td>(H \cdot P \cdot \text{block.size} + 2 \cdot X \cdot P \cdot \text{block.size} + 2 \cdot P \cdot \text{block.size})</td>
</tr>
<tr>
<td>Delete</td>
<td>(H \cdot P \cdot \text{block.size} + 2 \cdot X \cdot P \cdot \text{block.size} + 2 \cdot P \cdot \text{block.size})</td>
</tr>
</tbody>
</table>

In the tables above:

- \(X\) is the number of splits (resp., concatenations) in the insert (resp., delete) operation. Thus \(X \leq H\) and \(X\) is usually much smaller than \(H\).

- The cost of the final insertion (deletion) of PDM (which is the third term) is smaller than \(2 \cdot P \cdot \text{block.size}\), in the case that some processors' parts of the conceptual node are not changed. This cost is always larger than or equal to \((P + 1) \cdot \text{block.size}\).

We conclude that the communication costs of TDB are much smaller than those of PDM.

#### 7 Performance comparison of a sequence of operations

In this section we compare the average total cost of an operation, for a sequence of operations produced by a Poisson source. The average total cost of an operation is the sum of the average waiting cost of an operation (i.e. the average waiting time of an operation in the queue, before it starts executing), and the average execution cost of an operation.

\(^1\)PDM — Physical Distribution Method (section 1)
Here, all costs are measured in terms of number of disk accesses. For simplicity, we assume the following:

1. As nodes' splits and concatenations are rare, we assume that neither splits nor concatenations are performed. Therefore, there are two basic execution costs: the cost of **SEARCH**, **INSERT**, and **DELETE** operations (which is equal to the height of the tree), and the cost of **SUBRANGE** operations.

2. While executing a **SUBRANGE** operation, each processor in both TDB and SFM, accesses exactly $Y$ disk pages, in addition to the nodes (i.e. disk pages) along the search path.

3. In SFM, the records are equally partitioned among processors, and all trees have the same height.

Under these assumptions, the analysis that we present for TDB is exact, while the analysis for SFM gives a lower bound, which is not tight, to the average total cost of an operation in SFM. We will use the following notation: the height of the conceptual tree of TDB is denoted by $X$, the difference between the height of this conceptual tree and the heights of SFM's trees is denoted by $\Delta$ (In [12] we show that for all reasonable cases $\Delta > 0$), the number of processors is denoted by $P$, the operations arrival rate (from the Poisson source) is denoted by $\lambda$, and the probability of a **SUBRANGE** operation is denoted by $Q$.

In SFM, in order to equally distribute the records among the processors so that the distribution rule is obeyed, statistics on the distribution of keys are needed. Such statistics are difficult to obtain in practice, and thus the keys are assumed distributed among processors using round-robin, or a hash function (without obeying the distribution rule). Therefore, TDB will be compared to both SFM that distributes the keys using round-robin (called round-robin-SFM) and SFM that distributes the keys using a hash function (called hash-SFM). It is obvious that hash-SFM is better than round-robin-SFM. We show below that TDB is also better than round-robin-SFM except for **INSERT** operations. The relationship between the performance of TDB and hash-SFM is more complex.

### 7.1 Comparing TDB and round-robin-SFM

In order to execute **SEARCH**, **DELETE**, and **SUBRANGE** operations in round-robin-SFM, all processors search in parallel (each processor searches its local tree), because one can not tell, prior to the execution, which processor stores the desired information in its local tree. Thus, the execution cost of **SEARCH** and **DELETE** (under the above assumptions) is $X+\Delta$ for each processor, while the respective cost for TDB is $X$. Similarly, the execution cost of a **SUBRANGE** operation in round-robin-SFM is $X+\Delta+Y$ for each processor, while the respective cost for TDB is $X+Y$. An **INSERT** operation in round-robin-SFM is executed only by the processor that inserts the record to its tree, at a cost of $X+\Delta$, while the same operation is executed in TDB by all processor at a cost of $X$. Therefore, it is clear that TDB is better than round-robin-SFM, except for **INSERT** operations.
7.2 Comparing TDB and hash-SFM

In order to calculate the average waiting cost of an operation in TDB and hash-SFM, we use two known results in queuing theory [10].

The average total cost of an operation in TDB:

TDB can be thought of as a server that executes a sequence of operations (whose arrival rate is denoted by \( \lambda \)); with probability \( 1-Q \) an operation's execution cost is \( X \), and with probability \( Q \) an operation's execution cost is \( X+Y \). In [12] we show that the average total cost of an operation in TDB is:

\[
Q * (X+Y) + (1-Q) * X + \frac{\lambda * ((Q*(X+Y)^2) + (1-Q) * X^2)}{2 * (1 - \frac{\lambda}{\lambda+Q} * (X+Y) - \frac{\lambda}{\lambda+Q} * X)}
\]  (1)

The average total cost of an operation in hash-SFM:

The advantage of hash-SFM (in comparison to round-robin-SFM) is that before executing \textit{search} and \textit{delete} operations, one can tell (using the hash function) the unique processor that may store the desired record in its local tree. Thus, only one processor executes the operation while the others are able to execute other operations. However, in order to execute a \textit{subrange} operation, all processors search their trees, and the execution is over once all processors finish their search. Thus SFM is a server to which operations arrive at rate \( \lambda \), and with probability \( Q \), an operation is executed by all processors (i.e. it is a \textit{subrange} operation), and with probability \( 1-Q \), an operation is executed only by one processor (with probability \( 1/P \) of being executed by a particular processor).

We calculate the average execution cost and the average waiting cost for a specific processor (these costs are identical for all processors). The resulting average total cost is only a lower bound, which is not tight, to the average total cost of hash-SFM, because the analysis assumes that the execution of a \textit{subrange} operation finishes when the execution of the specific processor (on which we focus) is over, but in fact, a \textit{subrange} operation is over only when the execution of all processors is over. Moreover, the host spends time in merging the result records for a \textit{subrange} operation, which are sent by the processors.

In [12] we show that the resulting average total cost of an operation, for a specific processor of hash-SFM is:

\[
(1-Q)*(X+\Delta) + Q*(X+\Delta+Y) + \frac{\lambda*(1-Q) * (X+\Delta)^2 + \lambda*Q*(X+\Delta+Y)^2}{2 * ((1 - \frac{\lambda}{\lambda+Q} * (X+\Delta) - \frac{\lambda}{\lambda+Q} * (X+\Delta+Y))}
\]  (2)

Table 1 below summarizes the comparison between the average total cost (of an operation) of TDB and hash-SFM. We used equation (2) for calculating the cost of hash-SFM, and thus, in practice, the performance of hash-SFM is worse. On the other hand, the calculations regarding TDB are exact. We use the following notations for the table's entries;
'S' means that the average total cost of hash-SFM is smaller than the respective cost for TDB, 'T' means that the average total cost of TDB is smaller than the respective cost for hash-SFM, '*' means that TDB can not handle a sequence of operations arriving in such a rate (i.e. 1/λ is greater than the average execution cost of an operation), and '=' means that both TDB and hash-SFM can not handle a sequence of operations arriving at such a rate.

These results imply that the most significant parameter is the operations arrival rate λ; as the arrival rate decreases, TDB becomes relatively (in comparison to hash-SFM) better. We conclude that according to the system's parameters, and especially according to the operations arrival rate, one can choose between TDB and hash-SFM. In the cases in which the arrival rate of TDB is not too close to its limit, TDB is better. The possibility of composing SFM and TDB (i.e. implementing a forest of TDB trees) is currently being investigated.

References


Table 1.

<table>
<thead>
<tr>
<th>Y</th>
<th>X</th>
<th>Q</th>
<th>0.1</th>
<th>0.25</th>
<th>0.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>S S S T T</td>
<td>S S S T T</td>
<td>T S S T T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>T S S T T</td>
<td>T T S T T</td>
<td>T T T T T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>T S S T T</td>
<td>T S S T T</td>
<td>T S S T T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>T T T T T</td>
<td>T T T T T</td>
<td>T T T T T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>T S S T T</td>
<td>T S S T T</td>
<td>T S S T T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>T T T T T</td>
<td>T T T T T</td>
<td>T T T T T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>T S S T T</td>
<td>T S S T T</td>
<td>T S S T T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td>S S S S S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8 Appendix

8.1 The basic features of TDB

The algorithms which implement TDB (detailed below) operate on the physical tree. Certain structural features are needed. The following are the important properties of the physical tree:

1. The root node of the tree is kept in the host's main memory.
2. Each processor's cut of a conceptual internal node contains the minimum possible key and a key larger than the maximum possible key in its subtree. Therefore, the number of keys in a cut of a node is one more than the number of pointers in that cut. A key and the pointer to its left (in a cut of a node) are called an entry. (This implies that the minimum key in a cut of a node does not belong to any entry.) Each processor's cut of a conceptual leaf contains the minimum key and a key larger than the maximum possible key of a record which is pointed to from the cut.
3. Let p be a pointer in a cut of an internal node held by a processor. Let n be the key to its left and m the key to its right. The possible key values in the subtree pointed by p are such that: $n \leq \text{value} < m$
4. Let p be a pointer in a cut of a leaf held by a processor, and let n be the key to its left. The key of the record pointed by p is n. The rightmost key value in a cut of a leaf is larger than the maximum possible key of a record which is pointed to from the cut.
5. Invariant: The maximum key in a cut of a node held by processor $i$ is equal to the minimum key in the cut of processor $i+1$ of this node. $0 \leq i \leq P-2$.
6. The left and the right links of a conceptual leaf are kept in the cut of processor $P-1$. These are used to implement the doubly connected list of conceptual leaves.
7. The cut addresses of the leaves are marked in order to distinguish between an address of an internal node and an address of a leaf. This is done by adding a special indicator to a leaf address. (Another way to identify leaves is by keeping track of the height of the tree).
8. In a physical tree that implements a conceptual tree of degree $E$, each processor holds in its cut of a node up to $E/P$ entries.
9. Each processor has a counter in each cut, containing the number of entries in the cut.
10. A pointer in a leaf's entry (pointing to a record) also contains the identification of the processor that holds the record in its disk, and, of course, the record's address in that disk.

Point 10 above needs an explanation. In some cases, entries need to be transferred from one processor to another. For example, when trying to insert an entry into a conceptual node which is not full, we may have a problem if the processor that should insert the entry to its cut of the node, has a full cut. In this case we have to "make room" for this new entry by transferring at least one entry out of this processor's cut. This is neither complicated nor expensive when dealing only with a transfer of entries, but it may be an expensive operation when dealing with records since a record may be massive. Therefore, we prefer to avoid moving records pointed by transferred entries. This implies that processor $i$ will request processor $j$ to retrieve a record if the entry of the record is in processor $i$'s cut and the record was originally inserted into processor $j$'s cut or moved, for some reason, to processor $j$'s cut.

8.2 The algorithms that implement TDB

We first define three basic procedures (We assume familiarity with $B^+$-tree operations [8], [16]):

1. Given a key, finding the path to the record containing the key.
2. Inserting an entry into a tree node.
3. Deleting an entry from a tree node.
We note that we describe procedures 2 and 3 above, only with respect to internal nodes. The few changes needed to adapt these procedures to operate on the leaves, are described in [12].

Searching

The host finds the appropriate entry in the root and sends the actual address contained in this entry to the processors. From now until finding the leaf, in each level, each processor searches its cut of the node for the appropriate entry (i.e. the leftmost entry in the conceptual node whose key is greater than the searched key). This entry is the only entry in the conceptual node, whose key is greater than the searched key and the key to its left is smaller than or equal to the searched key. (This is why properties 3 and 5 in the previous section are needed). Only one processor finds the entry and sends the address of the node pointed by the entry to the other P-1 processors. Then, all processors “go down” to their cut of the pointed (conceptual) node at the next level. Finally, a processor finds that the entry is pointing to a leaf and informs the others that a leaf has been reached. At the end of this procedure, the processor that holds in its cut the entry which points to the desired record can determine the disk in which the record is kept and its address in that disk.

Inserting an entry into a node

Each processor checks if the new entry should be inserted to its cut of the node. Only one processor will find the cut to which the new entry should be inserted. Call this processor center. There are three possibilities:

1. There is enough free space in the page holding center's cut of the node. In this case center inserts the new entry to its cut of the node, increments the cut's entry counter and sends the others a stop message informing them that this insertion operation has ended.

2. The page holding center is full but there is free space in the conceptual node. (That is, the conceptual node is not full, as the number of entries held by all processors is less than the degree of the conceptual tree. This is checked for (by center) by sending a count message instructing each processor to send the content of its cut's counter to all other processors). In this case, by the result of the count message, center determines a nearest processor whose page (containing a cut of this node) is not full. Call this processor np. Space for the new entry is freed in the cut of center (of this node) as follows. If np>center (respectively, np<center) then entries are shifted left (respectively, right) from processor i to processor i+1 (respectively i-1), center≤ i <np (respectively, np<i ≤center). Figure 3 exhibits this idea.

3. The conceptual node is full. Therefore, it is split into two conceptual nodes. In the physical tree, this means that each processor should have a cut of each of these two conceptual nodes. Thus, each processor sends its cut's entries to the appropriate processors. (Based on its identification number and the counters of all the processors, each processor can determine to which processors it should send which entries). Hence, each processor's cut of the old conceptual node is now changed. As a result of this conceptual node-split, recursively, a new entry is inserted into the parent of the split conceptual node.

Deleting an entry from a node

Each processor checks if the entry should be deleted from its cut. Only one finds the entry. Call it center; center checks if the conceptual node contains more than E/2 entries. (This is checked for (by center) by sending a count message. Recall that E is the degree of the conceptual tree). There are three possibilities:

1. The conceptual node contains more than E/2 entries. In this case center deletes the entry from its cut.
2. The conceptual node contains exactly \( E/2 \) entries but the sibling to its left (or its right) contains more than \( E/2 \) entries. (This is also checked using a count message). In this case processor \( P-1 \) (0) transfers the rightmost (leftmost) entry of its cut of the conceptual sibling, to processor 0 (\( P-1 \)) that inserts the entry, as the leftmost (rightmost) entry, in its cut of the conceptual node. (Therefore one entry is transferred from the conceptual sibling to the conceptual node. It is possible that a shift of entries among cuts of this conceptual node is necessary to make room for the transferred entry). In the conceptual parent, the key, between the pointer that points to the conceptual node and the pointer that points to the conceptual sibling, is updated accordingly. Now the conceptual node contains \( (E/2)+1 \) entries and therefore center can delete the entry from its cut of the conceptual node.

3. The conceptual node contains exactly \( E/2 \) entries and so are the sibling to its left and the one to its right. In this case the conceptual node and its sibling to its left (to its right) are concatenated and become one conceptual node that contains, after the deletion, \( E-1 \) entries. Therefore, in order to perform the concatenation in the physical tree, each processor sends the entries of its cut of the conceptual node and of the conceptual sibling, to the appropriate processors. (Using its identification number and the counters of all the processors, each processor can determine to which processors it should send which entries). Following this concatenation, the algorithm deletes an entry, recursively, from the parent of the concatenated node.

We note that the count messages, in cases one and two above, can be omitted if each processor keeps, in each of its cuts, a counter that counts the number of entries in the whole conceptual node.

To complete the description of the algorithms we outline how to implement the required operations using these three basic procedures:

- **SEARCH**: By using the procedure in 5.1 the processor that contains the entry that points to the searched record, in its cut, requests the proper processor (the number of the disk is part of the pointer in the entry) to retrieve the record, and to send it to the host.

- **UPDATE**: Same as SEARCH but once retrieving the record, the processor updates it and writes it back. (We assume that no modification to a record’s key is made by the update operation).

- **INSERT**: Using the procedure in 5.1 the proper leaf is located. Using the procedure in 5.2, a new entry, that points to the new record, is inserted to that leaf. We omit details concerning the initial placement of the record.

- **DELETE**: Using the procedure in 5.1 the proper leaf is found. Using the procedure in 5.3, the entry that points to the record is deleted from that leaf and the record is discarded. We omit garbage collection details.

- **SUBRANGE\((k_1, k_2)\)**: Using the procedure in 5.1 (searching for \( k_1 \)), the leftmost leaf that points to a record whose key is greater or equal to \( k_1 \), is found. Each processor sends to the host all the records that are pointed by its cut’s entries and whose keys are in the subrange (recall that some of the records pointed by a processor’s cut, may not be stored in its disk). Then, all the processors, simultaneously, read their cut of the next leaf (to the right of the current leaf), and this process is redone. (Recall that processor \( P-1 \) contains, in each leaf’s cut, the links to the right and the left sibling). When processor \( P-1 \) finds, in its cut of a leaf, an entry that points to a record whose key is greater than \( k_2 \), it sends the others a message that this subrange operation ends in this leaf. The host collects the messages from the processors. Each group of messages contains the records in the subrange that are pointed by a conceptual leaf. The host concatenates the messages (of such a group of messages) in processor number order. In that way the records are produced in key order.
Assume we have a $B^*$ tree of degree 10. (A) presents a typical node (with two free entries). The same node in the conceptual tree is shown in (B). In the physical tree, $P=5$, each processor can hold in its cut up to two entries; (C) presents a possible distribution of entries to cuts. The entry $(41, L_{new})$ should be entered into the conceptual node. $P_1$ is full. Using a count message, $P_1$ determines that there is still room for a new entry and that the closest processor that can accommodate an entry in its cut is $P_3$. Therefore, a shift of the maximum entry from $P_1$ to $P_2$ and from $P_2$ to $P_3$ is performed resulting in (D). Now $(41, L_{new})$ is entered into $P_1$, as shown in (E).