M.I.T. Laboratory for Computer Science

Request for Comments No. 241
March 9, 1983

Concurrent Access to a Semi-Recoverable, Extendible Hash Table

by James W. Stamos

1. Overview

Three factors increased the complexity of implementing a particular extendible hashing scheme [5]. Concurrent access, controlled updates, and canonical buffer indices caused a number of difficulties. The application, a table of object descriptors for the Swallow repository [1] called the Object Header Table (OHT), required concurrent access. Improving the efficiency of crash recovery led to the second requirement of reliability. Finally, although canonical indices also appear in the original hash table design, they became a nuisance in a multiple-process environment for recoverable hash tables. This paper assumes familiarity with extendible hashing.

2. A Novel View of Extendible Hashing

Many expositions attempt to explain extendible hashing by presenting a conventional hash table whose length changes dynamically [4]. The table grows by dividing the entries in one or more buckets according to the first unexamined bit in their hashed values. If this bit is zero, the entry stays put. Otherwise, the expansion algorithm appropriately places it in one of the new buckets that it adds to the far end of the existing table (Figure 1). This technique widely spreads keys with similar hashed values and seems to be a difficult notion to express and understand.

Another scheme for extendible hashing doubles the directory size when any bucket referenced by only one directory index overflows and splits into two buckets [3]. Using a conceptual, 'pictorial' distance measure, relocated keys do not move as far because the two new buckets are always adjacent (Figure 2).

For schemes in which the directory attains a maximum size (or has only one size) as in [5], the following description is much clearer and thus easier to comprehend. Consider a fixed-length hash table with 2^m buckets in a horizontal row (Figure 3). The m most significant digits in the binary expansion of a hashed key reveals the bucket that would contain the key if it were inserted. This hash table changes size by growing or shrinking one bucket at a time. If a bucket has a length of 2^n, the m+1th to the m+nth bits in the binary expansion of the hashed key yield the portion of the bucket that should contain the key. A two-dimensional picture in which the height of the bucket is proportional to its size (or the log of its size) intuitively captures the redistribution of keys when the table shrinks or grows. This redistribution is local in that keys move only in the vertical direction when their containing bucket changes size. The neighbors of the key in the original bucket are still its nearest neigbors in the new bucket, except that the distance is scaled according to the bucket's expansion or contraction ratio.

3. Terminology

The extendible hash table contains a collection of key-value pairs called *entries.* A fixed-size *index buffer* acts as a directory and contains pointers to all *nodes* of the hash table. Each key-value pair maps to exactly one node and one page in that node, as explained above. The ordering of entries within a page is immaterial. Associated with each node is an unbalanced binary tree called an *overflow tree.* Overflow trees are a special form of binary tree that consist only of leaves. The shadow nodes of the implicit binary tree exist only as references from one leaf to another leaf. The "leftmost" leaf contains the implicit root. Overflow trees contain entries that mapped to a full node page when they were last inserted.

Primary memory contains a copy of the index buffer, a cache of recently accessed pages, and a buddylist for allocating and deallocating pages in the disk file. A buddy management scheme is used because every request is for a consecutive sequence of disk pages that has a size equal to some power of two. In addition to the latest version of the index buffer, the disk file contains node pages, overflow pages, and free pages. The table shrinks and grows in a controlled manner so that the buddylist is not needed for recovery.

4. Concurrency

Swallow repositories provide atomic, stable storage for objects involved in (nested) transactions. To increase throughput, the repository contains a moderate number of processes that service requests. These requests may belong to the same transaction, related transactions, or completely different transactions. Although some form of locking is needed to control concurrent access to the hash table, different operations have different requirements.

For example, the hash table changes size by doubling or halving one node at a time and appropriately processing the overflow binary tree. Here the natural unit of locking is the set of pages in a node and its overflow tree.

Inserting, deleting, fetching, and updating all access exactly one node page. In addition, these operations may traverse a single path in the corresponding overflow tree. The natural unit of locking for these operators is a single node page and the entire overflow tree.

Unlike the previous examples, some operations require access to more than one node. One such operation is *stabilize,* which ensures that the disk file is consistent with the perceived state of the hash table. Although it is possible to achieve this updating without ever locking more than one node at a time. two problems arise. First, this form of update does not represent a consistent *snapshot* of the hash table. While the Swallow OHT does not require consistency at this level, other applications may. Second, changing the size of a hash table with a relatively small number of entries involves two nodes. In order to function properly, the hash table must split and join nodes atomically.

Iterating over a set of key-value pairs also requires the ability to lock a set of nodes. If the desired entries are those pairs whose keys hash into a given numeric range, the resulting set of nodes is contiguous. Such an iterator is extremely useful during debugging. For the more general case of an arbitrary predicate, however, the iterator must examine all entries and thus lock all nodes.

There are conceptually three lock sizes: a single node, a contiguous set of nodes, and all nodes. The first and third types are degenerate forms of the second, and thus all locks represent an interval. Since each operation locks exactly one interval of nodes, there is no possibility of deadlock. In the more general case of arbitrary lock sets, acquiring node locks according to their order in the index buffer prevents deadlock and guarantees progress.

One potential bottleneck arising from a single lock size is the inability to lock a portion of a node. Operations that touch exactly one page within a node must still lock the entire node as well as the associated overflow tree. Under the following supposition, however, this simplistic locking strategy causes only a negligible reduction in concurrency. The key assumption is the high degree of locality in time and the lack of locality in space. References in the near future are likely to concentrate on those key-value pairs recently referenced. However, given a hash function that is not correlated with reference tendencies, nonreferenced entries that happen to be in recently referenced nodes are unlikely to be referenced in the near future.

Since the node locking strategy prevents deadlock, the only other *internal* source of deadlock arises from a fixed-size cache of disk pages. Operations typically lock at most two disk pages at any one time. Exceptions to this bound occur when a node changes size, since the algorithms may lock at most five pages at once. Recursive iterators currently lock as any pages as there are in the maximum depth of any overflow tree involved in the iteration. Careful programming can easily avoid this requirement, and thus we may set a loose upper bound of five pages per process. To prevent deadlock, the cache could limit the number of processes that obtain access to a disk page at any one time. A second scheme is to make the cache size large enough to handle the maximum number of processes executing under the application. Another alternative is to take advantage of the virtual memory and permit the cache size to exceed its defined size for short periods of time. I chose the third alternative primarily for programming convenience, but only actual use will determine the extent of cache overflow.

5. Recovery

The entire hash table is essentially a tree; the lack of cycles drastically simplifies crash recovery. First level nodes of the hash table tend to be large and always contain a number of pages equal to some power of two. All other levels in the tree are overflow subtrees and contain single-page nodes. A page in the disk file representing the extendible hash table is *accessible* if, and only if, it is a page in a node referenced by the index buffer or it is an overflow page referenced by an accessible page. A key-value pair is *accessible* if, and only if, it lies on an accessible disk page.

One weak constraint from Swallow is the prohibition of any disk files that contain two accessible key-value pairs with the same key. Unless the hash table implements a timestamp scheme or some other ordering technique, the process attempting to recover from a crash is unable to distinguish two accessible key-value pairs with the same key. Swallow can circumvent this problem because OHT entries are only *hints*. That is, the Swallow repository contains enough stable information to reconstruct all object headers. One solution to contradictory information is thus to discard duplicate entries during recovery. In the general case, however, the extendible hash table may not be a hint. Those situations require a more disciplined algorithm that prevents duplicate entries from ever appearing in any recovery attempt. The current implementation meets this requirement by modifying the structure of the table in an atomic manner. Moreover, the underlying algorithm is easily adapted to one that supports atomic updates of hash table entries.

All key-value pairs have a fixed size and typically do not move from a given disk page. The only exceptions to this claim of immobility occur when a node doubles or halves or when an overflow page itself overflows. When a node changes size, each key-value pair in the node will appear on two disk pages, since it appears in the current node as well as the new node. Pages comprising the new node are different from those comprising the current node. To ensure that exactly one copy of each entry is accessible, the implementation first stabilizes all pages in the new node and all pages in the new overflow tree. There is a great deal of flexibility in the order in which the pages are stabilized. To ensure the new pages are stabilized at least once, the current implementation stabilizes new node pages immediately after generating them from the corresponding portion of the old node(s) and overflow tree(s). Once the implementation finishes the doubling or halving, it stabilizes the overflow tree and then changes the appropriate index buffer entry. This technique represents the usual practice of creating a new structure, changing a single reference, and then deallocating the former structure.

When an overflow page overflows, the insert routine creates a new overflow page, divides the entries on the original page according to the most significant ignored bit of their hashed values, stabilizes the new page, and then retries the insert. Note that the immediate updating of the original overflow page is not necessary. The new overflow page on the disk contains appropriate values and becomes accessible only when the first overflow page is purged from the disk page cache.

Atomic updates of entire nodes and atomic extensions to overflow trees simplify the recovery of the extendible hash table. The recovery algorithm reads the buffer of node addresses into primary memory and then looks at node sizes. A binary search quickly determines which node is the next to double. In order to restore the buddylist that manages free pages in the disk file, the recovery algorithm must traverse all overflow trees. Guaranteeing that all accessible node and overflow pages are undamaged and readable may be done if required by the application. Comparing key-value pairs for appropriateness is useful during debugging and can also serve as a periodic check on the correctness of the normal algorithms and even the recovery techniques.

The extendible hash table would be fully *recoverable* if all table operations were atomic, the representation were stably stored, and except for benevolent side effects, unlocked disk pages in the cache were clean. The current implementation does not meet the latter two requirements. While all operations are atomic, their effects do not immediately propagate to the disk file. Since there is only one copy of the disk file, the storage is not stable. Both requirements can be met by the obvious modifications.

In the current implementation, all structural changes to the table are atomic with respect to the disk file. The size of the disk page cache approximately limits the number of changed pages that would not migrate to the disk file in the event of a crash. Assuming a cache that is small in relation to the entire table, only a minute fraction of the table in the disk file will not be up to date. Since there is only one copy of the disk file, a corrupted or otherwise damaged disk page can also lose data. Because a recovery process can effectively regenerate almost the entire hash table, say the table is *semi-recoverable.*

6. Canonical Indices

As the total number of entries declines, the hash table adjusts by contracting. If nodes contain more than one page, the table halves the size of one node each time it contracts. Let this process be known as *halving.* When the table becomes so small that each node has only one disk page, it shrinks by combining two adjacent nodes to form a new one-page node. The index buffer entries that referred to the old nodes must now refer to the new node. Call this technique of merging two separate nodes a *join.* Since the table joins only adjacent nodes, index buffer entries that refer to the same node are always contiguous. Since the number of index buffer entries that refer to a given node doubles with each join, any maximal set of equivalent buffer entries has a cardinality that is some power of two. When the total number of entries increases, the hash table adjusts by performing the inverse operations of *doubling* and *splitting.* These operations also preserve the contiguity and cardinality aspects of maximal sets of equivalent buffer indices.

In order to lock a node referenced by more than one index buffer entry, a process must effectively lock all relevant index buffer entries. One approach is to lock an interval of index buffer entries. A second scheme is to choose a *canonical index*, for example the smallest, and lock that one. Since computing the interval and determining the canonical index are of comparable complexity, we will assume the latter approach.

Canonical indices caused many minor problems by requiring special-case tests and additional code. For example, the recovery algorithm can not assume that each index buffer entry refers to a distinct node. For tables containing relatively few key-value pairs, the recovery algorithm must compute canonical indices and check that the appropriate number of index buffer entries share a given node.

When a node doubles, each node page maps into two consecutive pages in the new node. When a node halves, two consecutive pages in the old node map into one page in the new node. On the other hand, during splitting/joining, the new/old nodes are not two consecutive disk pages allocated together as part of one node, but two one-page nodes arbitrarily located in the disk file. Since their lifetimes may be unrelated, they are separately allocated.

Splitting and joining have a minor impact on atomic changes to the table structure. Assume node N_1 splits into nodes N_2 and N_3 (Figure 4). Let the canonical indices be I_1, I_2, and I_3. If N_2 is before N_3 ($N_2 < N_3$), then $I_1 = I_2 < I_3$. If I_2 and I_3 are on the same disk page, the node is split atomically by updating I_1 and I_3 simultaneously. The hash table moves directly from the state in Figure 4a to that in Figure 4c. However, if I_1 and I_3 are on different disk pages, atomicity requires that the expansion process update I_3 *before* it updates I_1 (Figure 4b).

One key fact that simplifies recovery is that any page in a node has sufficient information to determine the actual size of that node. Detecting incomplete split operations is thus straightforward. Assume the machine supporting the extendible hash table crashes after writing I_3 but before writing I_1 in the above example (Figure 4b). When the recovery process finds an incomplete split, as in the case of N_1 and N_3, it simply aborts the split by discarding N_3 and returns the table to the state shown in Figure 4a.

Consider the inverse opertion of joining N_2 to N_3 to form N_1. Again assume $I_1 = I_2 < I_3$. If I_1 and I_3 are on the same disk page, the hash table moves directly from the state shown in Figure 4c to that in Figure 4a. When I_1 and I_3 are on different disk pages, however, atomicity requires that the contraction process update I_3 *after* it updates I_1. Assume a crash occurs after writing I_1 but before writing I_3 (Figure 4b). When the recovery process finds an incomplete join, as in the case of N_1 and N_3, it finishes the join operation by discarding N_3. Because the recovery algorithm can not distinguish incomplete splits and joins, it treats them similarly.

These two update orderings always inserts duplicate information into the extendible hash table and then remove the redundancy. For example, during a node splitting, the sequence of information snapshots is N_1, $N_1 \cup N_3$, and $N_2 \cup N_3$ (Figures 4a, 4b, and 4c). Since $N_1 = N_2 \cup N_3$, the sequence is equivalent to $N_2 \cup N_3$, $N_2 \cup N_3 \cup N_3$, and $N_2 \cup N_3$. During a join operation the sequence is $N_2 \cup N_3$, $N_1 \cup N_3$, and N_1 (Figures 4c, 4b, and 4a), which is equivalent to $N_2 \cup N_3$, $N_2 \cup N_3 \cup N_3$, and $N_2 \cup N_3$. For node doubling and halving, N_3 does not exist. We have N_1, N_2 and N_2, N_1, respectively. Since $N_1 = N_2$, no information is lost. Restricting duplicate entries to incomplete splits and joins and ensuring that splits and joins never occur simultaneously ensures a straightforward recovery.

7. Comments on CLU

Except for minor problems involving bit-level access, CLU was a suitable language in which to implement this hashing scheme. Mint, the existing package for treating integers as an ordered collection of bits, provided adequate facilities. However, there was no simple way of determining the efficiency of various operations. For example, creating the integer 2^n for some integer $n > 0$ may be done by an appropriate left shift of the number one. Setting the nth bit in zero to be true achieves the same result, but there is no clear indication that either technique is more efficient. A more important problem was the appearance of tag bits in integers. Run-time support for CLU dictates that the two most significant bits of integers be identical. Attempts to set one withoug changing the other fail. To avoid special-casing the algorithms, I neglected to use the two most significant bits and considered positive integers with at most 30 bits. This decision had no other effects beyond a slight reduction in the potential resolution of hashed values. However, if an application requires a number of bits equal to a power of two, this shortcoming will become intolerable.

Since CLU does not yet explicitly support concurrency, I used the MESA model of processes, monitors, and condition variables. Interval locks governed access to nodes, provided mutual exclusion, and prevented deadlocks (Figure 5a). Except for changes to the size of the hash table, this strategy would also suffice for synchronizing access to the index buffer. However, updating one index buffer entry on the disk requires either two disk page transfers (read the page, modify the one index buffer entry, and then write the page) or one page transfer and the stability of all index buffer entries on that page (copy a portion of the main memory version to disk). Implementing the second, more efficient scheme required additional restrictions on index buffer access. A process operating under the monitor lock is free to read any index buffer entry. A process that has locked a node does not need the monitor lock to read the index buffer entries corresponding to that node. However, such a process cannot update the corresponding index buffer entries without first securing the monitor lock. In summary, reading an index buffer entry requires either the corresponding node lock or the monitor lock. Writing an index buffer entry requires both the node lock and the monitor lock (Figure 5b). This convention permits a process that extends or shinks the table to write the one or two modified pages of the index buffer to the disk file without locking all nodes whose index buffer entries fall on those pages. Since the table changes size infrequently, the overhead of securing the monitor lock in addition to the node lock is negligible. This scheme permits reading of index buffer entries without incurring any overhead beyond that required for node access synchronization. Since index buffer entries are read much more than written, the solution appears to be favorable.

8. Limitations

The current scheme suffers from two important drawbacks. First, it offers no synchronization capabilities beyond mutual exclusion of concurrent operations. For example, two processes can use the *fetch* operator to obtain the same key-value pair and then use the *update* operator to change the value component of the entry. Although the implementation serializes the invocation of these operators, it makes no guarantees as to their execution order.

The second drawback is the lack of full recoverability. One method of achieving this goal is to have each operation stabilize all the pages it modifies. However, the overhead associated with this scheme will make it grossly inefficient. An alternative is to use a *stable* portion of primary memory as an over-written, write-ahead log for frequent operations. While the details and utility of such a scheme have not been thoroughly investigated, it appears promising.

9. Interactions with Memory Management

CLU uses a conventional page-swapping virtual memory. Implementing the extendible hash table essentially outside the CLU environment had two drawbacks but provided one benefit. One disadvantage is the duplication of code and effort from 'reinventing the wheel,' since the hash table is extremely similar to a virtual memory. The cache fetched pages on demand, purged pages when necessary, and maintained clean/dirty bits for pages. A double-indexing scheme mapped a hashed key into its unique node page. Viewing this hashed key as a virtual address, the index buffer as a page table, and node pages as words within a page, the correspondence between the hash table and a standard virtual memory is striking. Moreover, since the pages were actual CLU objects, it is possible for the operating system to purge the virtual pages containing these page objects and thus cause double page faults when only one was anticipated.

Another disadvantage of working outside the CLU environment was the loss of type checking and the ability to reference objects. Each page object consisted of an ordered collection of bytes. Keys and values were not stored directly on a page. Instead, their bit-level representations were stored and directly manipulated. Operating on bit-level representations of keys and values required two new pseudotypes related to the original key and value types. For these kinds of types, there is little checking that may be done at compile time.

Since the hash table was outside the CLU world, any object references it contained would not be in the domain of the garbage collector. The unreliability of such references prevented keys and values from containing any pointers to CLU objects. There was thus no general method for keys and values to reference CLU objects. CLU objects could also not directly reference key-value pairs, because these pairs were not ordinary objects. However, by holding a copy of the immutable key, CLU objects could indirectly reference key-value pairs.

The one important benefit of operating outside the CLU environment is the fine-grained control over the location of node and overflow pages as well as the order in which a set of pages is forced to the disk or stable storage. All CLU objects are *transient* because their lifetimes do not extend between user sessions or across crash boundaries. On the other hand, the hash table and its key-value pairs are *persistent* objects, since they can exist for any number of user sessions and (generally) survive crashes. This fundamental difference in outlook prohibits ordinary references from persistent objects to transient objects but allows references in the opposite direction. Current schemes that automatically manage memory also prevent the implementor from establishing constraints on the timing and manner by which persistent objects and structures are stabilized.

10. Lessons

Given the opportunity to implement another extendible hash table, I would avoid unnecessary complexity by preventing the table from ever reaching the splitting and joining stage. This decision drastically simplifies coding and debugging but reduces the amount of code only slightly. The only shortcoming is the amount of disk space that it wastes.

The novel aspects of this project arose from its being an exercise in retrofitting concurrency and semi-recoverability into an existing data structure that was not designed (or at least advertised) to accommodate these features. In terms of the number of lines of code, the basic extendible hash table required as much code as the lower-level abstractions: the disk page cache, the buddylist, a module to manage page operations, and a parameterized set. In the extendible hash table with concurrency and semi-recoverability, explicit recovery code represented 20% of the top-level code, or only 11% counting the lower-level abstractions. Estimates for the concurrency code are 5% and 3%, respectively. Introducing concurrency and semi-recoverability had only minor effects in terms of lines of code.

The run-time costs of these features are also modest. Semi-recoverability required the prepurging of additional pages whenever a node changes size or an overflow page overflows. Since these events occur infrequently, the overhead is not substantial. Support for concurrent access introduces a small amount of overhead by requiring a process to first gain the monitor lock in order to lock the appropriate node. Except for insertions and deletions that change the size of the table, the only additional time penalties are the subsequent release of the monitor lock and the node lock.

The relative ease in extending the existing algorithms to support concurrency and semi-recoverability is due primarily to the tree structure of the hash table and the natural partitioning of the structure into unrelated nodes. Since the table is always acyclic, atomic updates to the structure are straightforward. More importantly, the overflow paths form a directed acyclic graph. Each page in a node coresponds to a possibly improper subtree of the overflow tree associated with the entire node. Although the overflow reference of a node page originally points to the 'root' of the overflow tree, accessing the overflow tree through a node page has a benevolent side effect. After the search, the overflow reference on the node page refers to the appropriate subtree and thus hastens future overflow searches through that page. Atomic updates to the structure of the entire hash table preserve the following invariant. Every key maps to exactly one accessible node page and exactly one accessible overflow page. In addition, there is exactly one path from the node page to the overflow page. There are never any merged or cyclic overflow paths and dangling references do not exist.

11. Related Research

Although providing concurrent access to B-trees has been the focus of numerous researchers, little work has concentrated on concurrent access to extendible hashing. In one study, Ellis [2] implemented Fagin's hashing scheme [3] and used the concurrency design as a basis for distributing the extendible hash table and the algorithms that manipulate it. That implementation, however, did not address crash tolerance. On the other hand, I considered recoverability but neglected the aspects of a distributed implementation.

Many of the differences between the two designs arise from the specific details of the two hash tables. In Fagin's scheme, each node consists of exactly one page and there are never any overflow trees. Ellis locks a single page and/or the entire directory and uses three types of locks to provide additional concurrency by permitting multiple readers. The costs of this concurrency are intricate locking strategies, subtle interactions between concurrent operations, and the complex deallocation scheme required to avoid dangling references. Ellis links the buckets in ascending hashed-key order to permit an access to correctly execute, even if the desired key-value pair moves because of intervening node splits and joins. In that implementation, node splitting and joining are not symmetric operations. Insertion and deletion are also not similar. While insertion/deletion may cause a local split/join in Fagin's scheme, in Lomet's scheme [5] the split/join or doubling/halving is typically unrelated to the particular insert/delete request. This separation permits a decoupling of the standard operators from the extension mechanisms and leads to a simpler, more modular decomposition.

There are a number of similarities between the two schemes. First, neither provides any general synchronization capabilities beyond a straightforward serialization of requests. Second, both employ recursive insert routines, since splitting a node page (or an overflow tree leaf) may not generate enough space to perform the insertion. Third, both use two locking granularities. Ellis defines locks to cover the entire directory as well as locks for a one-page node. My scheme uses the monitor lock to synchronize access to the directory but uses interval locks that typically lock only a single node. Implementing the additional concurrency found in the Ellis design for Lomet's hash table requires a combination of the approaches for concurrency in hash tables as well as that for tree-like structures, since every node references an overflow tree. Finally, both schemes utilize the total ordering on nodes defined by hashed-key values to avoid deadlock when an operation requires more than one node lock.

References

[1] Arens, Gail C., "Recovery of the Swallow Repository," M.I.T. Laboratory for Computer Science, MIT/LCS/TR-252, January, 1981.

[2] Ellis, C. S., "Extendible Hashing for Concurrent Operations and Distributed Data," Rochester University TR110, October, 1982.

[3] Fagin, R., "Extendible Hashing -- A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems* 4, 3 (September, 1979), pp. 315-344.

[4] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," Proceedings of the Sixth International Conference on Very Large Databases, Montreal (October, 1980), pp. 212-223.

[5] Lomet, David B., "Bounded Index Exponential Hashing," IBM T. J. Watson Research Center, RC 9192, January, 1982.

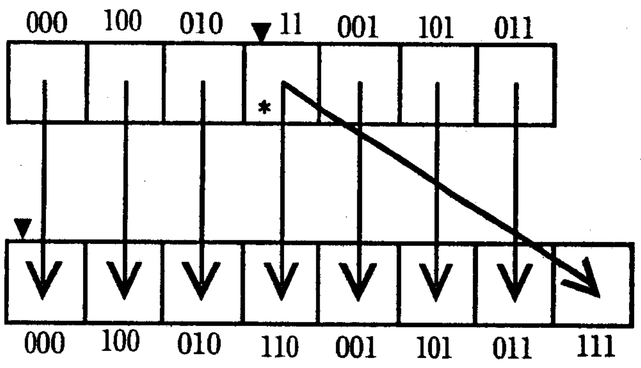

Figure 1. Extendible Hashing with Linear Table Growth

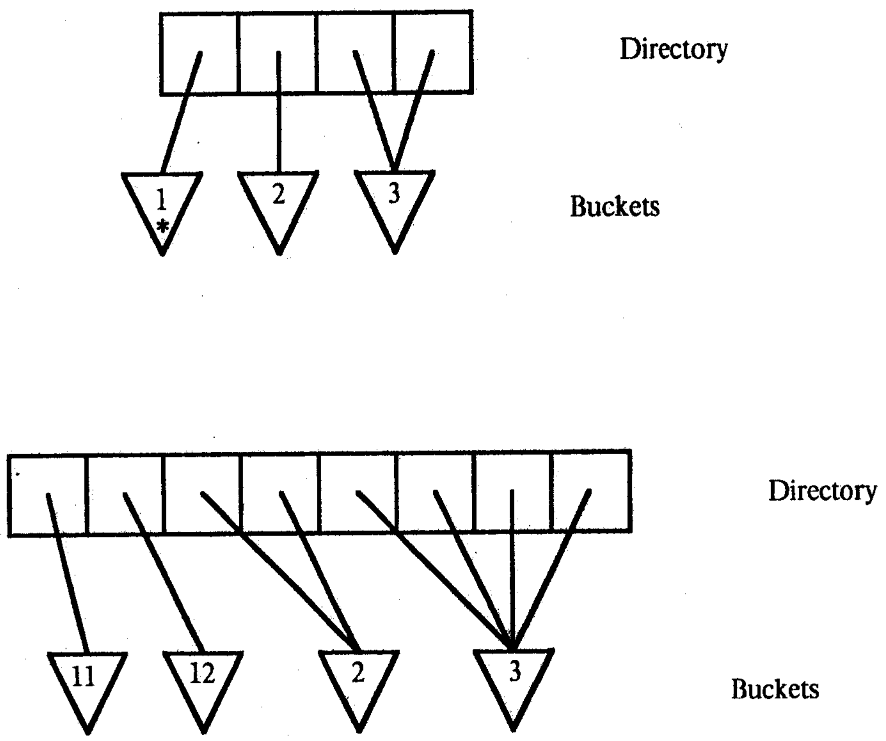

Figure 2. Extendible Hashing by Doubling the Directory

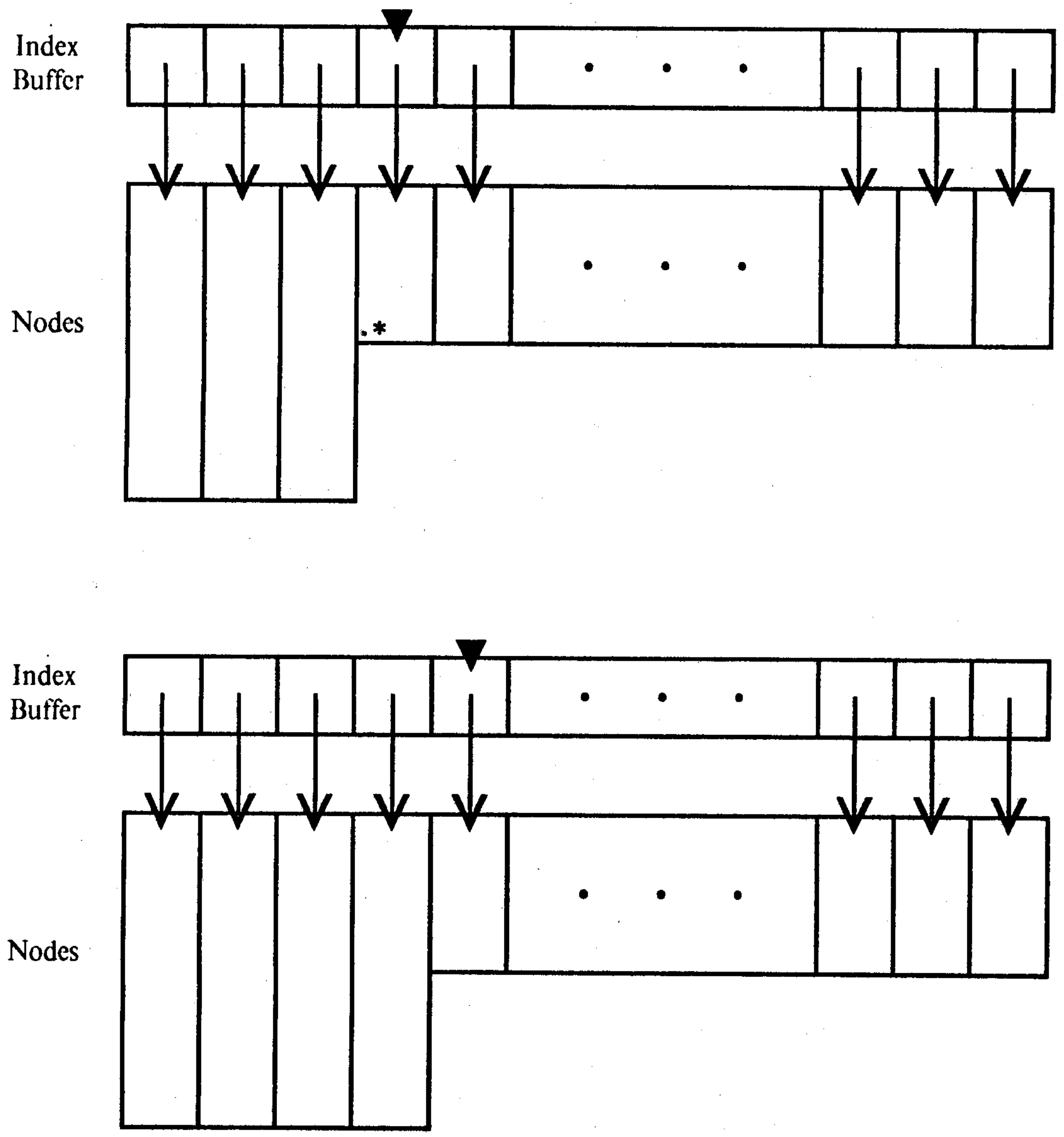

Figure 3. Extendible Hashing with a Fixed-Size Directory

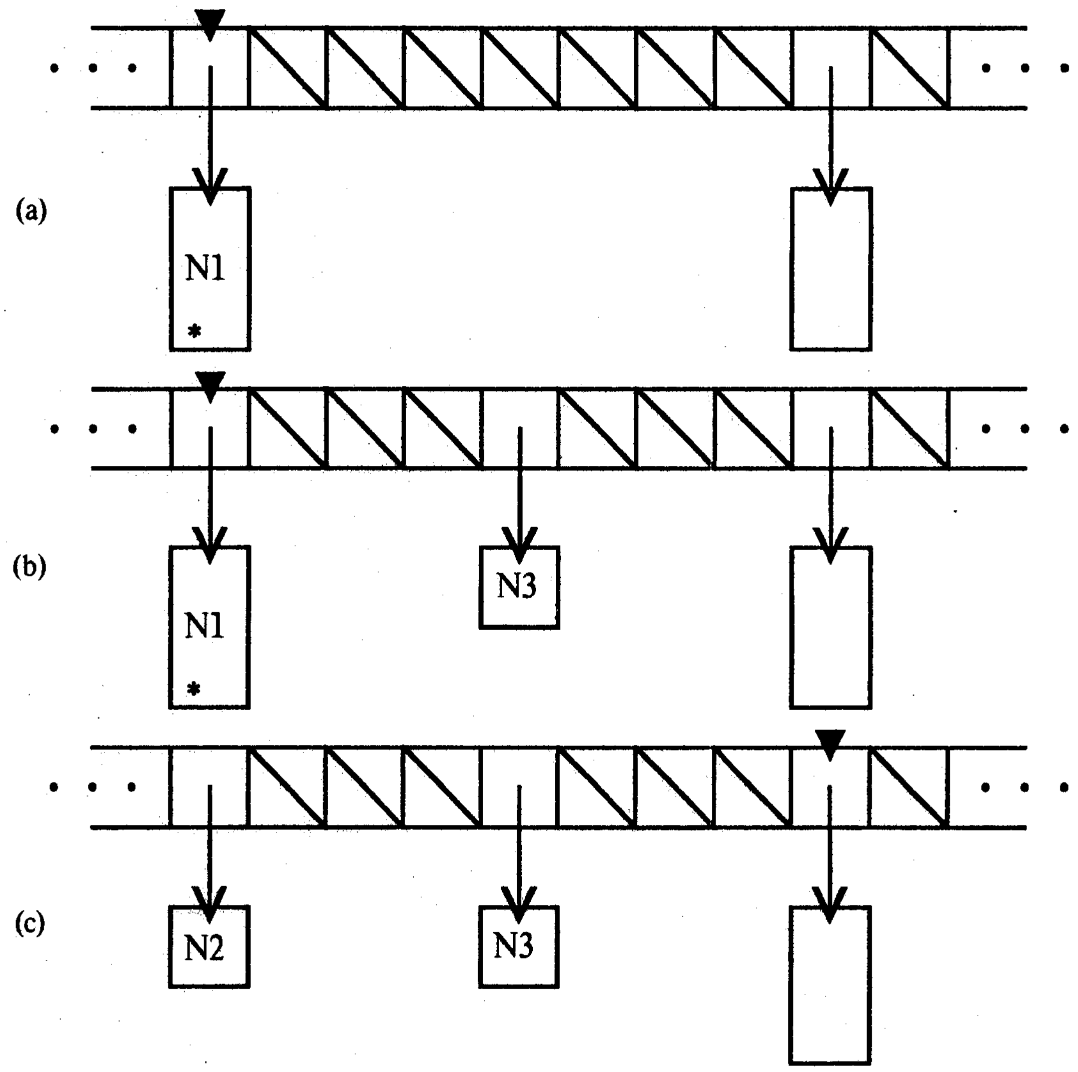

Figure 4. Example of a Node Split (a through c) and a Node Join (c through a)

(a)

Interval Lock \ Monitor Lock	No	Yes
No	--	--
Yes	R/W	R/W

Node Access

(b)

Interval Lock \ Monitor Lock	No	Yes
No	--	R
Yes	R	R/W

Directory Access

Figure 5. Lock Requirements