Chapter 9

Concurrency Control on Search Structures

"I bade them search – search well." (Edgar Allan Poe)

9.1 Goal and Overview of the Chapter

Search structures such as B$^+$-tree indexes are among the most fundamental concepts for ensuring efficient data accesses. Therefore, we devote a separate chapter to the concurrency control issues for search structures. We will focus on B$^+$-tree indexes, but many of our considerations can potentially be generalized, or carried over to other search structures.

The framework for our discussion is layered schedules with the access layer and the page layer of a data server (see Chapter 1 for a brief overview of the layered architecture of database systems and similar data servers). At first glance, this suggests considering record operations and the resulting page accesses. As for the access layer, paying attention to record operations alone falls, however, short of ensuring serializability in the presence of "intensional", predicate-oriented record searches. In particular, merely locking the records that are fetched, updated, inserted, or deleted cannot prevent the so-called phantom problem. Recall from Chapter 1 that such problems arise when newly inserted, deleted, or updated records satisfy the search predicates of queries by concurrent transactions, although these records are not returned (and, in fact, possibly not even touched) by those queries themselves.

As an example, consider transaction t_1 that moves all 50-year-old and older people who live in Dallas to Phoenix, by issuing an SQL-style *Update Persons Set City = "Phoenix" Where Age* $\geq$ 50 *And City = "Dallas"*. Concurrently, a

second transaction t_2 first asks for all people who live in Phoenix and later submits a query to retrieve all people from Dallas; the update by t_1 is assumed to take place in between these two queries. Now suppose that the update affects only a single record x representing the person who moves from Dallas to Phoenix. Further suppose that t_2's first query returns records p and q, and its second query returns d and e. So, record x is not found by either of the two queries, although the two together would be supposed to find everybody from Dallas or Phoenix. If locking at the access layer is restricted to the actually accessed records, x for t_1 and p, q, d, and e for t_2, this phantom anomaly cannot be prevented. So rather one needs to lock also either the predicates of the queries and the update themselves or the appropriate information in the underlying access paths that are used to execute these query-level operations. In the example, the above SQL operations could use a B$^+$-tree index on the City attribute of the Persons table. Recall from Chapter 1 that the leaf pages of such an index contain searchable (key, RID) pairs where the key is one of the City values and RID is a pointer or a list of pointers to the records with this City value. So locking the keys "Dallas" and "Phoenix" (which appear in the above SQL operations) in this index, on behalf of both transactions, would render an explicit lock conflict and thus ensure a proper serialization.

Thus, concurrency control needs to consider index search keys in addition to the stored records. Furthermore, as with layered schedules, additional page locks are needed to ensure that the underlying search structures are properly traversed and manipulated, so as to ensure that the search and other operations of the access layer appear isolated. The default solution for this purpose would be to apply a protocol like 2PL (or even S2PL or SS2PL) to the subtransactions that correspond to index searches as well as insertions and deletions of (key, RID) pairs. Because of the special structure of index trees and the resulting highly specialized page access patterns, it is, however, possible to devise tailored protocols that allow more concurrency on the index pages than standard 2PL would. As concurrency control on search structures is a potential bottleneck in many performance-demanding applications, we should definitely strive for the best possible protocol.

This chapter provides solutions to the above issues that are structured according to our two-layer setting. Section 9.3 addresses the necessary locking of index keys for the operations at the access layer, postulating isolation of the operations themselves, e.g., by assuming 2PL for the duration of the corresponding subtransactions. Section 9.4 then improves the underlying page-layer protocol by introducing advanced techniques that are specifically geared for B$^+$-trees and similar structures. Finally, Section 9.5 looks into additional optimization techniques that mostly aim to reduce the overhead of the entire two-level protocol. To provide the necessary background on how search structures are typically implemented in data servers, we begin with a short review of B$^+$-trees in Section 9.2.

9.2 Implementation of Search Structures by B$^+$-trees

B$^+$-trees

In this section we briefly review implementation techniques for search structures to the extent that this background is relevant for the discussion of concurrency control. The prevalent data structure for the implementation of the ADT "search structure" is the *B$^+$-tree*. This is a page-structured search tree whose nodes correspond to pages and where all leaf nodes are at the same tree level (i.e., have the same distance from the root node). An example B$^+$-tree for an index on the Name field of a Person table is shown in Figure 9.1. This tree has height 2; it consists of a root node and

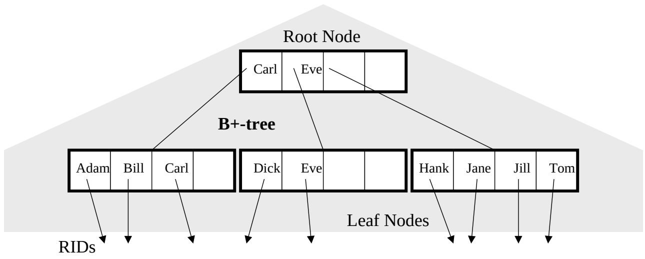


Figure 9.1: Example of a B$^+$-tree.

three leaf nodes at the next lower level. The leaf nodes contain all values of the indexed field that exist in the database. Each value is followed by a list of RIDs to the corresponding records. Such a combination of a key value and a RID list is called a *leaf-node index entry*. A leaf node consists of one or more of those entries, typically between ten and a few hundred. The non-leaf nodes of the index consist of *inner-node index entries*, each of which contains a value and a pointer to an index node of the next lower level. The semantics of such an entry is that the value is the largest key value that exists in the subtree to which the pointer refers. For example, the entry in the root that points to the leftmost leaf has the value "Bob", as this is the (lexicographically) largest key value in that subtree.

An index entry that points to the rightmost child of a non-leaf node does not have a key value but merely consists of a pointer. In the example of Figure 1.6, the third entry of the root simply points to the rightmost leaf, meaning that all key values in this rightmost subtree are higher than the key value "Eve" in the preceding

entry of the root. Note that this technique is applicable not only to the root, but avoids an explicit key value in the index entry for the rightmost subtree among the children of the same node at all levels. The highest possible key value within such a rightmost subtree is then determined by the index entry for all subtrees with the same parent, which can be found in that parent's parent. As a consequence, each non-leaf node has one more pointer to the next index level than key values.

search algorithm

Because of this invariant between index entries and subtrees, inner-node index entries can serve as *routing entries* for a root-to-leaf key search in the tree. A search for a given key value starts at the root node. It determines within the currently inspected node the routing entry whose value is the smallest value that is larger than or equal to the given search key, and then follows the pointer of this routing entry. If the search key is larger than the highest value among the routing entries of the index node, then the search follows the pointer to the rightmost child. Within an index page, a simple binary search is performed, assuming that the index entries are kept sorted. This step per node is recursively applied until the leaf level is reached. There, either the given search key is found, and then we directly obtain the desired RID list for the qualifying records, or we realize that the search key is not present at all in the database. In the example tree, a search for the key "Dick" would follow the "Eve" router in the root and find the key in the middle leaf. In contrast, a search for the key "Joe" would follow the router to the rightmost leaf, where it would detect that this key does not exist in the database.

A range lookup is implemented in a similar manner: it searches for the lower-bound key, which would lead to the smallest existing key that is larger than or equal to the lower bound. From there the range lookup can scan all leaf nodes to the right of this starting entry until it reaches a key that is larger than the upper bound of the range. To this end, a typical implementation of a B$^+$-tree embeds both forward and backward pointers in the leaf nodes, so that all leaf nodes are linked in the sorting order of the indexed field's domain. (These pointers are not shown in Figure 1.6.)

I/O cost of index lookups

Obviously, as long as it is guaranteed that the tree is balanced in the sense that all leaf nodes occur at the same level, the I/O costs of an index lookup are logarithmic in the size of the underlying data, namely, one disk I/O per index level (disregarding caching effects and the follow-up leaf accesses in a range lookup). In fact, the base of this logarithm is usually fairly high, as it coincides with the fan-out of the tree. Given that the inner nodes contain only short routing entries, fan-outs in the order of several hundreds are typical in practice. One may even further increase the fan-out by compressing the routing entries' values. As these merely serve routing purposes, they do not necessarily have to be equal to the largest key in the corresponding subtree. Rather any value that separates the key-value sets of two adjacent subtrees is eligible. For example, the routing value "Eve" in the root of Figure 1.6 could be compressed into a separator "Ex" or even "F", as both are lexicographically larger than “Eve” but still smaller than the smallest key value in the adjacent subtree to the right and thus satisfying (a slightly relaxed form of) the tree invariant that facilitates the logarithmic search. The resulting effect is a typical tree height of at most 3 in most database applications or at most 4 in some extremely large databases, and this is equal to the number of page accesses for an exact-match lookup (disregarding the case of very long RID lists per key value that would span multiple pages).

A B$^+$-tree is an extremely efficient index structure for large collections of data records, provided it is guaranteed that it always remains balanced in the presence of insertions and deletions. There are simple cases of insertions that do not pose any problem at all. Consider the cases illustrated in Figure 9.2. Two new keys,

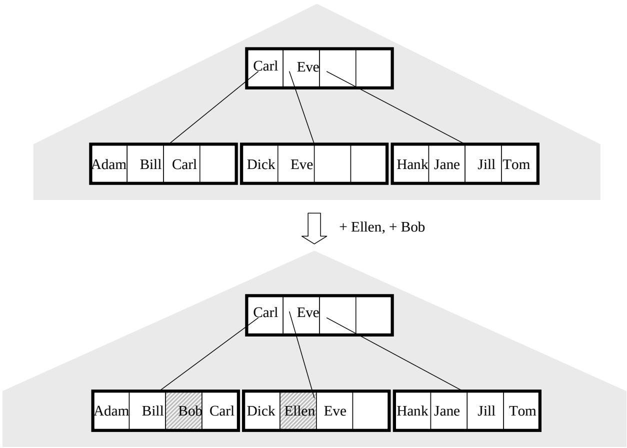


Figure 9.2: Insertion into a B$^+$-tree.

“Ellen” and “Bob”, along with RIDs of two new data records, are to be inserted into our example tree. In both cases, the key is first searched to check if it already exists. If it exists, then merely the new RID would be added to the RID list of

the key. Otherwise the (unsuccessful) search stops in the leaf node where the key should reside if it existed. Hence this is exactly the leaf into which the key should be inserted. If this leaf still has sufficient free space available, then the insertion is straightforward. This is exactly what happens in the cases of “Ellen” and “Bob”.

node split

Next consider the case when the leaf node into which a key is to be inserted does not have enough free space. This case is illustrated in Figure 9.3. The new key value

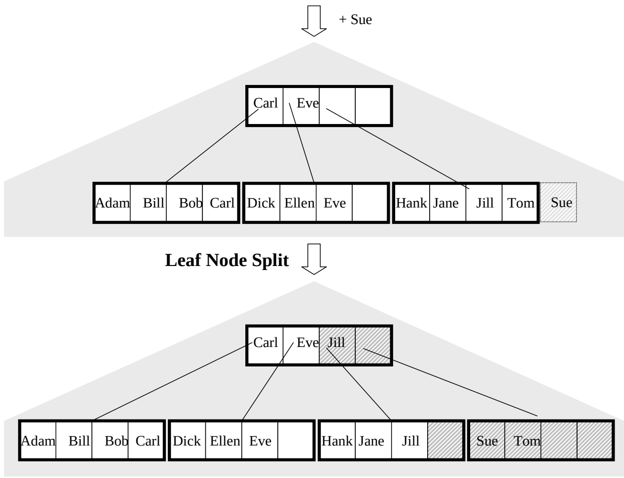


Figure 9.3: Insertion into a B$^+$-tree with a leaf node split.

“Sue” is to be inserted into the rightmost leaf of the example tree, illustrated by the dotted box, but this causes what is known as a *page overflow*. A simple-minded remedy could now start building up a chain of linked pages that are anchored at the overflowed leaf, but this would obviously be susceptible to degeneration in that arbitrarily long overflow chains and thus poor search performance could not be ruled out. The actual solution, which is efficient as well as elegant, is rather based on an incremental reorganization of the tree that is known as a *node split* and grows the tree in a way that it remains balanced. This incremental reorganization proceeds in three steps:

1. A new, initially empty page is allocated as a new leaf node. This page will become the right sibling of the overflowed page. *grow*

2. The index entries of the overflown page, including the new entry, are partitioned across the overflown and the newly allocated page. This is the actual split operation. A typical constraint here is that both pages end up with approximately the same number of entries or the same percentage of used space. *split* This serves to ensure that the filling of all tree nodes is above a certain threshold of 50 percent; otherwise the tree would stay nominally balanced but its height could become larger than necessary.

3. The partitioning of the index entries ensures that the largest key value that remains in the original page is smaller than the smallest key value in the new page. Then, the last step consists of "posting" the largest key values of the two pages, along with the corresponding page pointers, as new index entries in *post* the parent of the original node. So both pages will have the same parent node. The original index entry that pointed to the overflowed page is removed from the parent (or, equivalently, substituted by one of the two new index entries).

In the example, the keys "Hank", "Jane", and "Jill" remain in the split leaf node, and "Tom" and the new key "Sue" are placed in the newly allocated sibling. Consequently, the largest key that remains in the split page, along with pointers to the two sibling nodes that are separated by this router value, are posted in the parent node. As in the simple insertion case, all modified pages and entries are marked as hatched boxes in Figure 9.3.

The final case that we need to consider for insertions is now rather obvious: what if the parent node of a split page does not have sufficient space for the two new routing entries (after the removal of the original router to the split page) and becomes itself overflowed? The solution is to apply the above split operation recursively, going upward in the tree. In fact, we already described the three steps of a split operation in a generic way in that they can be applied not only to leaf pages but to all node levels in the tree. The recursive split propagation stops when the root level is reached. When the root node overflows, its split operation creates a sibling node. Since we cannot have two root nodes in a tree, we now have to create a new root node that points to the old root and its new sibling. In other words, the tree grows by adding another root level on top of the old root level. This case is illustrated in Figure 9.4, where the insertion of the new key "Betty" leads to a split of the leftmost leaf as shown in the top of the figure. The posting of "Bill" as a new router in the

root node then causes the root to split. The resulting situation with a tree of height 3 is shown in the lower part of the figure.

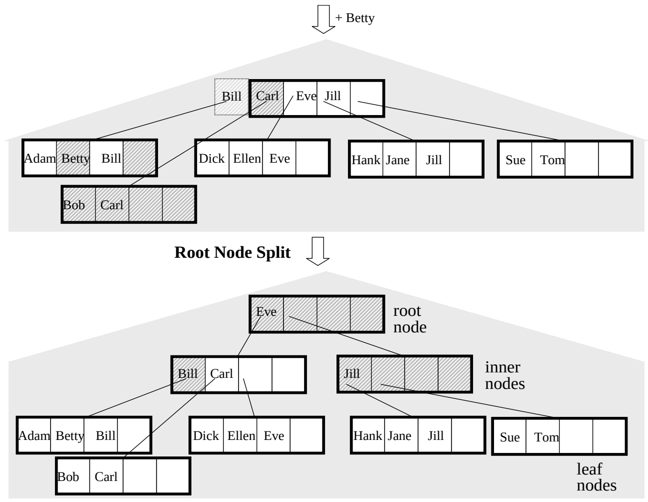


Figure 9.4: Insertion into a B^+-tree with a root node split.

The split operations ensure that a B^+-tree always remains perfectly balanced, while they themselves are very efficient operations, namely, with I/O costs that are logarithmic in the size of the tree. A dual solution can be devised for handling the deletions of keys in a tree. The counterpart to a split operation would be the merging of two adjacent siblings, and the subsequent removal of a router from the parent node. One of the two merged pages can then be deallocated from the tree and returned to the free space management. So altogether a B^+-tree can grow and shrink, both in width and height, dynamically in an incremental manner while guaranteeing that the filling of the allocated pages does not fall below a specified threshold of 50 percent. In practice, since insertions are so much more frequent than deletions, one often does not bother to implement the merging-based algorithms for deletions and rather allows pages to become underfull (i.e., filled less than 50 percent) due to deletions. Only when a tree node becomes completely empty would both the page and its routing entry in the parent be removed from the tree. Finally, updates of data records that cause the value of an indexed field in an existing data record to change are usually treated as a pair of key deletion and key insertion as far as the index structure is concerned.

The dynamic *"grow-and-post"* reorganization principle can be generalized to a much wider class of tree-based index structures including indexes for multidimensional, spatial, or temporal search operations. Modern database systems therefore offer a broad repertoire of index choices and other physical design options to the database administration and tuning staff. For B^+-tree indexes alone, this spectrum includes the choice of fields or field combinations that are to be indexed, a choice between storing only keys and RIDs versus entire data records in the index leaves, the latter often called a "cluster" or "clustered index", and many other fine-tuning options. Database systems make these options available to the database administrator in the form of parameters to the "Create Index" command of SQL.

9.3 Key-Range Locking at the Access Layer

We assume an ADT interface for a search structure that provides the following operations:

$$insert(key, RID)$$
$$delete(key, RID)$$
$$search(key)$$
$$range\_search(lowkey, highkey)$$

search structures as an ADT

The insert operation adds a (key, RID) pair to an index entry (stored in a leaf node of a search tree in the most common implementation of the index ADT). If the index is a unique one, i.e., allows at most one occurrence of each key, or if the new key is not yet present, then the insert operation creates a new index entry. Analogously, the delete operation removes a (key, RID) pair, possibly removing the entire index entry if no other RID exists for that key. The search operation returns all RIDs for a given key, by looking up the appropriate index entry (by means of top-down descent from the root to the appropriate leaf node if in the search-tree implementation). Finally, the range_search operation returns all RIDs of records that contain a value in a specified range between *lowkey* and *highkey* (including both of these bounds). Its implementation (on a search tree) would initially search for the smallest index entry whose key is equal to or larger than *lowkey*, and would then scan all following index entries leaf nodes of the index tree until it reaches an index entry with a key

that is larger than *highkey*. Recall from Chapter 1 that this procedure is known as an *index scan* or *range scan*.

Simple commutativity arguments tell us that insert and delete operations commute between each other, unless we consider a unique index. In the latter case, a delete followed by an insert for the same key would result in the key being present, whereas the other way round, the insert would fail as an attempt at adding a duplicate and the subsequent delete would remove the key from the index. For simplicity, we will mostly assume that the index under consideration is a non-unique one. The analogous derivations for the case of a unique index are left as an exercise (see Exercise 1). We can further infer that search operations do not commute with inserts or deletes if they refer to the same key; otherwise they are conflict-free from the viewpoint of the access layer. Finally, the range_search operation is in conflict with an insert or delete operation if the inserted or deleted key falls into the range from the *lowkey* to the *highkey* value of the range search.

simple key-range locking

With these considerations, we can immediately derive a possible locking protocol for the above search structure operations. Insert, delete, and search operations acquire locks on the specified keys, and range_search operations lock their corresponding key ranges. The lock conflict test must consider the equality of keys and the containment of a key in a key range. As for lock releases, the usual 2PL (or a stronger) discipline is used. This protocol, which we will refer to as the *simple key-range locking* method, can be viewed as a very restricted form of predicate locking: it considers only the predicates of the form $indexed\ attribute = key\ value$ or $lowkey \leq indexed\ attribute \leq highkey$. These simple conditions are the result of decomposing the actual query's predicates into the operations at the access layer. This form of locking at the access layer is indeed sufficient to prevent anomalies and guarantee full serializability.

At this point, one may wonder how phantoms can be avoided when no index is used for executing predicate-oriented queries. In this case, a table scan must be used for finding qualifying records, and that table scan needs to look up a list of pages on which the records of the corresponding table reside. This list, which is part of the data server's storage layer metadata (for addressing and free space management, see Section 1.5), can be viewed as a key range from minus infinity to plus infinity (assuming numerical attributes) which is effectively read-locked by the lookup. Any concurrent insertion or deletion would have to inspect and actually modify this list, too, and would therefore need a write-lock thus revealing a lock conflict. So key-range locking is a method that is not necessarily tied to the existence or usage of an index. An actual implementation may choose a different approach in the scenario sketched, but, in principle, all possible execution plans are correctly covered by the key-range locking protocol. For the following, we simplify the presentation by assuming that an index is used for all queries.

Simple key-range locking entails two major problems:

- First, it requires a lock manager that can manage value ranges, not just individual values, and has an efficient method for testing the containment of a value in an interval. Although this is surely not a dramatically challenging problem, the solution would inevitably incur more overhead than that of a standard lock manager for individual values only. As locking is part of a data server's innermost loops, such increased overhead is a concern.
- Second, a range search specifies the key range to be locked upon its start. At this point, the entire range is locked although values close to the interval's upper bound (i.e., the highkey value) will be reached only in a while. The effect is similar to preclaiming: we lock out update operations that refer to keys close to the upper bound for at least the entire duration of the range search. It would suffice, however, to start blocking updates once the range scan arrives at a certain key.

incremental key-range locking (previous-key locking)

So we would like to improve key-range locking with respect to its overhead and its concurrency by gradually building up the lock on a key range in an *incremental* manner; and, if possible, all locks should be implemented as if they were *individual key locks*. This will lead us to an *incremental and implicit key-range locking* protocol, also known as *previous-key locking* in the literature, for reasons that become clear in a short while. To this end, we first modify the interface of the index ADT by breaking up a range scan into multiple short steps that correspond to the actual implementation: a range search is viewed as a single-key search followed by a sequence of *next* operations that return the next index entry unless the specified *highkey* value were exceeded. So the *highkey* value of a range search serves as a stopping condition for the index scan. The *next* operation is obviously a context-sensitive operation; it always takes up from where the scan was left by the previous *next* or an initial *search* operation. A sequence of *next* calls must always be preceded by a *search* call. Each *next* call expects a current "cursor" position as an input parameter, in the form of the last visited key and the page number of the leaf node on which that key resided. So the detailed form of *next* operations and the preceding initial *search* is:

index scans as sequences of next operations

$search(searchkey)\ \uparrow key\ \uparrow page$
$next(currentkey, currentpage, highkey)\ \uparrow key\ \uparrow page$

with input parameters *searchkey*, *currentkey*, *currentpage*, and *highkey* and return parameters *key* and *page* where *key* is the found key and *page* is the page on which the returned key resides (which will become the *currentkey* and *currentpage* parameters for the subsequent *next* call). A $range\_search(lowkey, highkey)$ operation is mapped onto the following call sequence:

$search(lowkey)\ \uparrow currentkey\ \uparrow currentpage$
$next(currentkey, currentpage, highkey)\ \uparrow currentkey\ \uparrow currentpage$
$next(currentkey, currentpage, highkey)\ \uparrow currentkey\ \uparrow currentpage$
etc.

which terminates when a called operation finds that the next key in the scan is higher than the specified *highkey* and then yields *nil* for its return parameters. In the following we will often use these operations without all parameters explicitly listed; the missing parameters are either irrelevant or can be easily derived from the context.

The second consideration towards a more efficient variant of key-range locking is to restrict the lockable intervals to those that are actually present in the index. For example, if the index contains index entries with keys 22, 25, 29, and 30 but not the other values between 22 and 30, then the intervals [22,25), [25,29), and [29,30) are among the lockable ranges. In general, a lockable interval is a one-sided open interval that starts from a key present in the index, including this key itself, and extends to the next higher key that occurs in the index, excluding this one. The fact that these intervals are one-sided open serves to make different intervals disjoint and allows us to uniquely name the intervals as follows. By identifying an interval with its left end (i.e., lower bound), we can lock an interval by acquiring a lock on its left-end value. So this simplifies the actual lock management, but we nevertheless use the convention that such a lock covers the entire interval.

The choice of exactly these intervals is, of course, not arbitrary. Rather, these are the intervals that are touched by the sequences of *next* operations in a range scan. So now we are also close to the desired kind of incremental lock acquisition for range searches: each of the initial search and the subsequent next steps acquires a lock on the interval whose left-end value it returns to the caller. Using the above "encoding" convention of mapping intervals to their left-end value, this kind of locking eliminates both of our concerns about simple key-range locking: locks are no longer acquired prematurely, and they can be managed with very little overhead.

As an example, consider the B^+-tree index shown in Figure 9.5. A range search of the form $range\_search\ (23, 34)$ would lead to the following sequence of operations and lock requests:

operation	lock request
search (23) ↑ 25 ↑ p	lock (22), representing the interval [22,25)
next (25, p, 34) ↑ 31 ↑ p	lock (25), representing the interval [25,31)
next (31, p, 34) ↑ 33 ↑ q	lock (31), representing the interval [31,33)
next (33, q, 34) ↑ nil ↑ nil	lock (33), representing the interval [33,36)

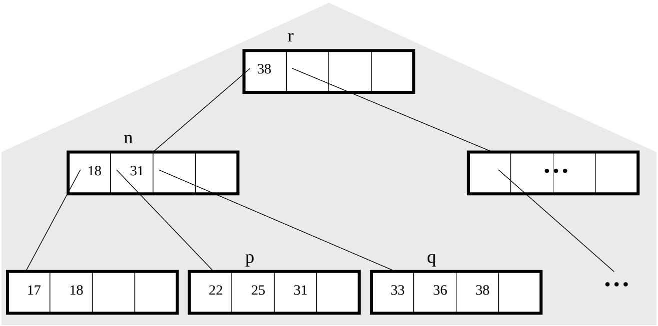


Figure 9.5: Sample B^+-tree index.

Note that, in the end, the range search will have locked the entire interval [22,36) thus properly covering the specified query range [23, 34]. The difference to the simple key-range locking is that this interval is comprised of the smaller intervals that represent the actually present keys and that the entire interval is incrementally locked by gradually acquiring the locks on the constituent intervals.

previous-key locks

Now we need to consider the necessary locking for insert and delete operations that would conflict with a range search, i.e., whose keys fall into the search operation's specified range. An insert operation for a key x would naturally lock x itself, representing the interval from x to the next present key. This holds regardless of whether x was already present or is newly inserted. For example, inserting the new key 27 into the index of Figure 9.5 should definitely acquire a lock for the interval [27,31). If the insert preceded the next operation of the range scan that proceeded forward from 25 (i.e., with 25 as the last value returned by the previous next call), the conflict between the two operations would be detected, as that next operation would then have to acquire a lock on key 27, too. However, if the insert operation followed the entire range search, no conflict would be signaled, as the formerly absent key 27 has, of course, not been locked by any of the next operations. So the problem is to detect that key 27 falls into the interval [25,31) by merely inspecting the left-end points of intervals.

The solution is to have an insert operation also lock the interval that precedes its primarily locked interval. In our example, insert(27) needs to lock the interval [25, 27), too; and the technical means would be to write lock the previous key 25, hence the name *previous-key locking* for this protocol. Note that it is, in fact, fairly natural for the insert operation to acquire this lock, too, as its effect is to split the former interval [25,31) into two intervals. The literature also considers a dual variant of previous-key locking that is known as *next-key locking*. That protocol essentially requires operations to lock the key on which they operate and the next higher key (with some relaxations of this base scheme). We leave a discussion of the "isomorphism" between previous-key and next-key locking as an exercise to the reader (see Exercise 5).

dual variant: next-key locking

Analogous considerations for delete operations suggest that a delete(x) operation should lock both x and the largest preceding key that is present in the index, as its removal of key x potentially merges the interval starting at x with the preceding interval, thus connecting the original interval's right-end point with a different, smaller left-end point. Although this intuitive explanation would strictly hold only if the index entry for x contained only a single RID, we apply the same locking rule also to the general case when x is still present after the delete of one (key, RID) pair. We will come back to possible relaxations of this rule for the latter case in Section 9.5.

Finally, an exact-match search(x) operations can proceed in the most natural way, by merely acquiring a read lock on the key x that it searches for. However, if this key does not exist yet, but is later added by an insert operation, it must still be possible to detect this conflict between the search and the insert operations. This is done by having the "unsuccessful" search operation lock the interval starting at the next lower, currently present key, as this one would be the previous-key lock to be acquired by a subsequent insert operation. Again, this is, in fact, fairly intuitive. Not having found a key is a specific result that must be reflected in a proper serialization order, and one way of capturing the missing key is by remembering what next key was found in the place where the missing key would have had to be. For example, for the index in Figure 9.5, a search (25) operation would lock key 25 (i.e., the interval [25, 31)), and a search (24) operation would lock key 22 (i.e., the interval [22, 25)).

Putting all of the above considerations together, we arrive at the following set of lock acquisition rules for the operations of the index ADT:

incremental key-range (previous-key) locking rules

- An operation *search*(x) requests a read lock on x if key x is present in the index; otherwise a lock on the largest key smaller than x is requested.
- An operation $next(currentkey, currentpage, highkey)$ requests a read lock on currentkey.

- An operation $insert(y, RID)$ requests write locks on y and the largest key smaller than y that is present in the index.
- An operation $delete(y, RID)$ requests write locks on y and the largest key smaller than y that is present in the index.

In addition, the 2PL rules on lock releases and the usual well-formedness rules for all locking protocols must be in effect. Somewhat more liberal rules are conceivable, but we postpone a discussion of such additional optimizations to Section 9.5. All rules together ensure the correctness of the protocol, as captured by the following theorem:

Theorem 9.1

Previous-key locking generates only conflict serializable schedules as far as index operations are concerned.

Proof. The proof shows that all potential conflicts between search, range_search, insert, and delete operations are correctly recognized by acquiring key-range locks. Since all locks are held according to the 2PL discipline, this is sufficient to prove serializability.

First, a conceptual conflict between a $search(x)$ operation and $insert(y, RID)$ or $delete(y, RID)$ exists only for $x = y$. Since all three operations acquire a lock on the primarily affected key, this conflict is surely detected. For an unsuccessful $search(x)$ operation, it is important for serializability to prevent a subsequent insertion of x as long as the seach transaction has not committed (e.g., if the search transaction repeated its search after the insertion or accessed some other data that is logically related to the presence or absence of x). Since both an unsuccessful *search* and and *insert* operation need to acquire a lock on the largest key smaller than x, this conflict situation is detected.

Second, there is a conceptual conflict between a $range\_search(lowkey, highkey)$ operation and $insert(y, RID)$ or $delete(y, RID)$ if the key y falls into the range from *lowkey* to *highkey* (including both bounds). This case is the decisive one that necessitates previous-key locks. To see that such conflicts are properly detected, we only need to observe that a range search gradually acquires locks, through its initial *search* and the subsequent *next* calls, on all existing keys starting from the largest existing key that is smaller than or equal to *lowkey* and ending with the largest existing key that is no larger than *highkey*. Since both $insert(y, RID)$ and $delete(y, RID)$ operations request a lock on the previous key of y, they must have at least one lock conflict with the sequence of *search* and *next* operations of a conceptually conflicting range search.

Third, two *insert* operations potentially conflict only if they refer to the same key, and even this is only a real conflict if the index is a unique one, disallowing duplicates. The remaining cases of two *delete* operations or an *insert* and *delete* pair are analogous. □

A further difficulty that arises once implementation considerations come into play is that the keys on which the various operations need to acquire locks are only partly known when the operation starts. The "previous keys" that need to be locked on behalf of inserts and deletes become known only during the execution of the operations. However, as the index operations themselves are performed as subtransactions, with an isolation guarantee by the underlying page layer, there is actually no problem with acquiring also access-layer (i.e., higher-level) locks in the course of an operation's execution. This means that the lock requests of the two relevant layers, the access and the page layer, are interleaved rather than being time-wise separated as in the layered 2PL presented in Chapter 7. Recall, however, from Chapter 7 that the exploitation of return-value commutativity also involved high-level lock acquisitions *after* the begin (and, in fact, not before the completion) of the corresponding high-level operations. So locks of both layers are dynamically acquired at the latest possible point, right when the "resources" to be locked become known. For example, an $insert(27)$ index operation on the index structure of Figure 9.5 would go through the following sequence of lock requests and execution steps, assuming a subtransaction-scope 2PL discipline on the page layer:

operation	lock request
read (r)	lock page r
read (n)	lock page n
read (p)	lock page p
find the previous key 25	lock key 25, representing the interval [25,31)
add the new key 27	lock key 27, representing the interval [27,31)
	release page locks on r, n, and p

We will come back to the interplay of index-key and page locks when discussing more relaxed, B^+-tree-specific page-lock release policies and other, similarly "liberal" concurrency control methods in the next section.

9.4 Techniques for the Page Layer

Before we embark on the presentation of various algorithms for the page-layer concurrency control of search structure operations, let us first illustrate once again the kinds of problems that arise. Throughout this section we focus on B$^+$-trees as the most important case and also a paradigmatic example of search structures for disk-resident data. Consider the B$^+$-tree index shown in the top third, labeled "stage 0", of Figure 9.6, and assume that an *insert*(30) operation takes place on this index. After the operation has located the leaf p into which the new key would have to be inserted but finds that page p is already completely full, it initiates a leaf-node split of p creating a new sibling p' ("stage 1" in Figure 9.6). The final stage, labeled "stage 2" in the figure, consists of posting this split in the parent of p and p', by adjusting the routing keys and pointers in node n (see Section 9.2).

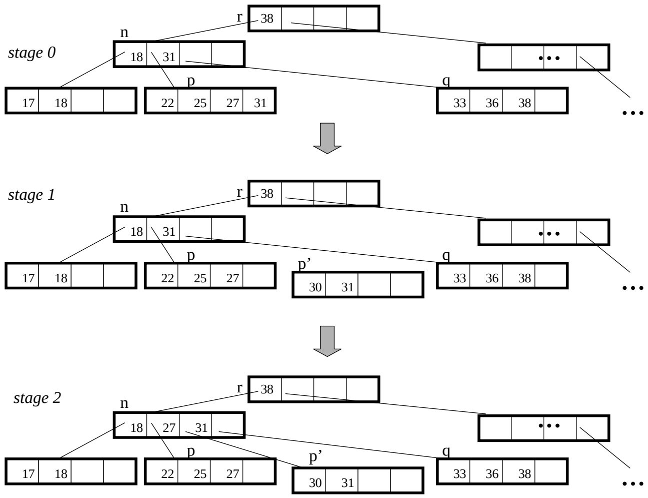


Figure 9.6: Stages of a B$^+$-tree leaf-node split.

Now suppose that a concurrent *search*(31) operation takes place. As a worst possible scenario, this search operation could perform its top-down descent of the tree in between the leaf-node split but before the split is posted in the parent node n. So this operations would see the intermediate state shown in stage 1 of the insert. The routing keys in nodes r and n would direct the search operation to

the leaf node p, but would it no longer find the key 31 there; so this search would erroneously return a "key not found" result. As an even worse scenario, suppose that an *insert*(31) operation rather than a search is interleaved in the above way with the split operation, and that the index is a unique index so that duplicate insertions should be disallowed. As the *insert*(31) would likewise be erroneously directed to the already split node p, nothing would prevent it from inserting another instance of the key 31 (together with some RID). As a result, the uniqueness condition would become violated and the database would be inconsistent.

The most obvious way to prevent this problems and ensure serializability of the subtransactions that correspond to index operations is to employ 2PL (or S2PL or even SS2PL) for such subtransactions. The purpose of this section, however, is to develop even better solutions in terms of the possible concurrency, taking specific advantage of the tree-traversal access patterns and the special structure of B$^+$-tree indexes. The various approaches are classified into three types:

- lock coupling, which is essentially a refinement of the tree locking protocol that we introduced in Chapter 4,
- the link technique, which exploits a specific implementation aspect of B$^+$-tree-like search structures, and
- the giveup technique, which is an optimistic variant.

9.4.1 Lock Coupling

The first method to be presented, coined *lock coupling*, follows the approach of the tree locking protocol introduced in Chapter 4. For simplicity, first consider only search and insert operations (i.e., no range searches or next operations, and also no deletions). These operations basically perform a top-down descent into the tree, following exactly one path of the tree, and the search operations even have the nice property that they never revisit a node again. This very specific page-access pattern immediately suggests the use of the tree locking protocol. Recall that the basic principle of this protocol is to lock a node only after holding a lock on its parent, but to release the parent lock as early as possible, once we know that the operation will not go up in the tree again.

Unfortunately, the latter is exactly what insert operations may occasionally do, namely, when they have to split a leaf node and need to post the split along the ancestor chain. So, although search operations could release non-leaf-node locks early, inserts would have to hold all locks until they know that no split occurs. This consideration can be made stronger by exploiting the fact that most splits propagate only one tree level upwards; only very infrequently, inner nodes need to be split, too, which would then necessitate adjustments (or even further splits) in additional ancestors. Now, when the insert operations performs its top-down descent, it can observe information about the space utilization of the various nodes and whether a node can space-wise accommodate the posting of an additional routing key and child pointer or not. When the free space in a node suffices to hold an additional key and pointer from a node split at some level further below, then we know for sure that the split will not propagate further upwards in the tree. So the parent of the node under consideration will definitely not have to be updated at all, which means that it will for sure not be revisited again by the insert operation. We call such a parent node of a node with sufficient free space a *split-safe node*. In the example of Figure 9.6, assuming a node capacity of four keys and pointers, node r can be recognized as split-safe once the $insert(30)$ operation has proceeded to node n and finds that node n can accommodate the posting of a split further below without itself having to be split. In the implementation, the test for "sufficient free space" is a bit complicated by the fact that keys can be of variable length. However, by assuming conservative upper bounds for the length of a routing key in an inner node, a proper decision can always be made.

split-safe tree nodes

These considerations immediately lead to a locking protocol for search and insert operations that is known as *lock coupling* and is driven by the following rules:

1. Search operations need to request a read lock on a node before the node can be accessed, insert operations need to request a write lock.

2. A lock on a node can be granted only if no conflicting lock is currently held and the requesting operation holds a lock on the node's parent.

3. Search operations can release the lock on a node once they have acquired a lock on a child of that node.

4. Insert operations can release the lock on a node if (i) the node is split-safe and (ii) they have acquired a lock on a child of that node.

For the above example scenario with concurrent operations $insert(30)$ and $search(31)$ on the index of Figure 9.6, the following would be a possible execution under the lock coupling protocol:

insert(30)	search(31)
write lock r	
write lock n	
unlock r	
	read lock r
	request read lock on n
write lock p	
allocate new page p'	
write lock p'	
split contents of p onto p and p'	
adjust contents of n	
unlock n, p, and p'	
	acquire read lock on n
	unlock r
	read lock p'
	unlock n
	return RID for key 31
	unlock p'

The correctness of this protocol is reflected in the following theorem:

Theorem 9.2

The lock coupling protocol for search and insert operations generates only schedules that are OCSR.

The easy proof can be based on the fact that the protocol is a special case of the general tree locking protocol. Note that, although we allow both read and write locks, we do not encounter any problems with "pitfalls" in the sense of the read-write tree locking protocol of Chapter 4, as insert operations acquire only write locks from the beginning. An alternative that requires a bit more thought for ensuring correctness would be to have insert operations also start with acquiring merely read locks on non-leaf nodes (but still a write lock on the leaf, of course), and only upon realizing the need for a split, read locks would be converted to write locks. This is indeed a feasible option, but it obviously entails the danger of deadlocks among insert operations. Yet another alternative that has turned out to perform fairly well would be to start out with read locks and roll back an insert operation once it realizes the need for a split. Note that only the operation should be rolled back, not the entire transaction, and this is more or less for free anyway since no updates have been made yet. Then, the insert operation would be repeated, but now acquiring the necessary locks in write mode.

Now let us add range searches, or equivalently, next operations to our considerations. A range search follows the forward-linked chain among leaf nodes (see Chapter 1). So we are actually dealing with a DAG structure here, no longer a pure tree. Fortunately, as you may recall from Chapter 4, tree locking can be generalized to DAGs by requiring that a lock on a node can be granted only if the requestor holds locks on the majority of the node's predecessors. Strictly speaking, the leaf-linked B^+-tree considered here is a dynamically evolving DAG, not a static one as originally assumed by the DAG locking protocol, but we will ignore this potential complication for simplification. As each leaf node has two predecessors, its parent in the tree and the preceding leaf, range searches would have to visit and lock also all inner nodes that point to the accessed leaves, and operations like search and insert that perform a top-down descent need to acquire also a lock on the leaf that precedes their actual target leaf. So, although we can in principle construct a working protocol from these considerations, chances are that it would not be very efficient.

On the other hand, we have a special situation that we can exploit to simplify the rule about having to lock the majority of predecessors. First, we have a clear separation between operations that need write locks and those that need read locks: searches need only read locks, and insert operations could be implemented by acquiring only write locks. This suggests the following modification of the "majority rule": a read lock can be granted if a lock on at least one predecessor is held, and the prerequisite for granting a write lock on a node would then be holding write locks on all predecessors. This way, searches would have to acquire locks only along a single path to the leaf level, and range searches would only need locks for the leaf chain. There would still be a severe penalty on insert operations, having to acquire locks on two leaves rather than one (in the no-split case). Fortunately, however, there is another particularity that we can exploit: splits create only new successors of a node, never a new predecessor (i.e., we always split "to the right"). With this restriction we can safely relax the lock-all-predecessors condition for insertions. It suffices that inserts lock the path to their target leaf and possibly a newly created leaf node.

These considerations bring us to the following extended version of lock coupling, now including range searches:

lock coupling with range searches

1. Search and range_search operations need to request a read lock on a node before the node can be accessed, insert operations need to request a write lock.

2. For search and insert operations as well as the initial descent of range_search operations, a lock on a node can be granted only if no conflicting lock is currently held and the requesting operation holds a lock on the node's parent.

3. Search operations can release the lock on a node once they have acquired a lock on a child of that node.

4. Insert operations can release the lock on a node if (i) the node is split-safe and (ii) they have acquired a lock on a child of that node.

5. Range_search operations can acquire a lock on a leaf node only if they hold a lock on the leaf's parent in the tree or on the preceding leaf. The first case is relevant during the initial descent of the tree, locating the first relevant leaf; the second case covers the progress along the linked chain of leaf nodes.

Now recall from the previous section that a range search is effectively broken down into a regular search operation and a sequence of next operations. In the protocol described above, all these operations would still be considered as a single subtransaction as far as their page locks are concerned. There is nothing wrong with this interpretation; after all, next operations are context-dependent operations. The question, however, is whether this could be further relaxed. In particular, would it even be possible to release all page locks on behalf of a range search in between two successive next operations. The answer is positive. Because of the special structure of a B^+-tree and, particularly, the fact that splits are performed always "to the right", we can indeed treat each next operation as a separate subtransaction. This actually relaxed the lock-coupling rule assumed so far: a next operation first re-acquires a lock on the leaf that the previous next or the initial search operation had last locked, and it does so without holding a lock on the parent or preceding leaf. Once the next operation then finds that it has to access the next leaf to retrieve the following key, it needs to acquire a lock on that leaf, now under the "protection" of already holding a lock on the preceding leaf. In essence, this prevents the preceding leaf from being split right in the moment of traversing the link to the next leaf.

These specific considerations for next operations are captured by the following additional rule for lock coupling:

lock coupling with next operations

6. A $next(currentkey, currentpage, highkey)$ operation needs to acquire a read lock on *currentpage*, and it can acquire a read lock on any other leaf page only if it holds a read lock on that page's preceding leaf.

Theorem 9.3

Lock coupling with next operations generates only OCSR schedules.

Proof. First note that without next operations we would simply apply the tree locking protocol of Chapter 4. The special twist of not releasing locks on nodes that are not split-safe is fully in line with standard tree locking. So the only complication are next operations.

Assume that there is a conflict cycle among index operations, i.e., subtransactions, that involves a *next* operation reading the two adjacent leaves p and q. The only way of forming such a cycle is to have at least one insert or delete operation in between the $r(p)$ and $r(q)$ steps of the *next* operation. However, that insert or delete operation would not be able to acquire a lock on p once p has been read by the *next* operation. So the "worst case" is that the insert or delete takes place on q. If there is a split, it would affect only leaves to the right of q, never the left neighbor p. Therefore, such an insert or delete must follow the next operation in the serialization ordering.

The order-preservation of the protocol is easy to verify, as tree locking by itself ensures order-preserving conflict-serializability and the above consideration on next operations already shows that their addition does not undermine this property. □

Note that the equivalent serialization order for a schedule of search, next, and insert operations may have a range search interleaved with inserts, as the above theorem only ensures us that the individual next steps can be isolated from the inserts but this does not consider the broader context of an entire range search. In combination with the incremental key-range locks that an entire range search acquires and in fact holds for the duration of the entire transaction, we are able to guarantee that inserts that run concurrently with a range search can be "disentangled" by means of commutativity-based transformations and tree pruning. An important observation towards the proof of this claim is that an insert can never take place behind the current "cursor" position of an ongoing range search, unless the insert key is smaller than the the range search's *lowkey*. The reason for this property is that both the range search and the insert need key-range locks at the access layer. So inserts can only occur after the current cursor position, and in this case one of the subsequent next steps of the range search would become blocked by the insert's key-range lock (unless the inserted key is even higher than the *highkey* of the range search). This effectively means that such inserts should precede the range search in the resulting serialization order, but as all access-layer locks would be held until the end of the entire transaction, no incompatible ordering of other operations is possible. This yields the following theorem:

Theorem 9.4

Lock coupling at the page layer together with incremental key-range locking at the access layer ensures the tree-reducibility of all 2-level schedules with these two layers.

Proof. Note that Theorems 9.1 and 9.3 together already show that schedules with search, insert, and delete operations alone are tree-reducible. This follows from the result of Chapter 6 that conflict-serializability at the higher level of a two-layered system and order-preserving conflict-serializability for the subtransactions of the lower level together guarantee tree-reducibility. So again the complication is caused by next operations on behalf of range_searches.

We already know from Theorem 9.3 that insert, delete, search, and next operations, if viewed as individual subtransactions, can be isolated by means of commutativity arguments. This argument is, however, not yet sufficient for isolating entire range_searches that consist of an initial search and a sequence of next calls. So the case to be considered is of the following form:

$\ldots search_i(lowkey)$
$\ldots insert_k(x, RID1)$
$\ldots next_i(currentkey1, highkey)$
$\ldots insert_l(y, RID2)$
$\ldots next_i(currentkey2, highkey) \ldots$

where a range_search of transaction t_i is interleaved with insert operations of two concurrent transactions t_k and t_l. The situation with interleaved deletes would be analogous and is not further discussed here (and an interleaved search or range_search of another transaction could trivially be pushed outside t_i's range_search to form an isolated subtree).

Because of Theorem 9.3, such a pattern can be transformed into one with no interleaving of page accesses for the search, insert, and next operations. Furthermore, the same arguments that we used in the proof of Theorem 9.1 show that the first insert operation's key x cannot fall into the range [lowkey,currentkey1], as this would have provoked a lock conflict. So this insert operation can be pushed to the left to precede t_i's initial search and thus becomes an isolated subtree. By the same token the second insert operation's key y cannot fall into the range [lowkey,currentkey2]; so it can again be commuted with the preceding steps of t_i's range_search. The general observation thus is that an insert operation that is interleaved with a range_search cannot refer to any key in the range that the range_search has seen so far. On the other hand, if it refers to a key that will be visited by later next steps of the range_search, then the insert can always be pushed to the front of the range_search, thus preceding it in the equivalent serialization order.

This consideration shows that inserts, deletes, searches, and entire range_searches can always be isolated, and in conjunction with Theorem 9.1 concludes the proof of tree-reducibility. □

As an illustrative example of the complete lock-coupling protocol and its combination with incremental key-range locking at the access layer, the following table shows the necessary lock requests, acquisitions, and releases for a range_search(24,35) operation that runs concurrently with an insert(30) operation on the B$^+$-tree shown as stage 0 of Figure 9.6.

range search (24, 35)	insert (30)
search (24)	
read lock r	
read lock n	
unlock r	
read lock p	
unlock n	
read lock key 22	
unlock p	
return 25, p	
	insert (30)
	write lock r
	write lock n
	unlock r
next(25, p, 35)	
read lock p	
read lock key 25	
unlock p	
return 27, p	
	write lock p
next (27, p, 35)	
request read lock on p	
	write lock key 30
	write lock key 27
	write lock p'
	⋮
	unlock p, p', n
acquire lock on p	
request read lock on key 27	
	⋮
	commit transaction
acquire lock on key 27	
read lock p'	
unlock p, p'	
return 30, p'	
next (30, p', 35)	
read lock p'	
read lock key 30	
unlock p'	
return 31, p'	
next (31, p', 35)	
read lock p'	
read lock key 31	
read lock q	
unlock p', q	
return 33, q	
next (33, q, 35)	
read lock q	
read lock key 33	
unlock q	
return nil, nil	

drain technique for empty nodes

Finally, let us add also delete operations to make the protocol truly comprehensive. If deletes were to initiate also complex mini-reorganizations of the tree, we would possibly need to revise our previous findings quite substantially. For example, if a leaf node whose space utilization falls below a certain threshold were to be merge with its left sibling, the correctness of the lock coupling for next operations would become questionable as reorganizations would no longer be confined "to the right" of the primarily affected node. Fortunately, these complications can be easily avoided. It is common practice (see also Chapter 1) that deletes do not trigger any such local reorganizations. Rather one would simply wait until a node becomes completely empty, leaving it connected within the B^+-tree structure, and only then it would be detached from both its parent and the linked chain of leaf nodes. But once we adopt such a lazy free-space reclamation policy, we can wait even a bit longer to ensure that no concurrent operation has already seen a pointer to the empty page from one of its predecessors and henceforth attempts to follow this pointer after the empty node has been deallocated. A simple sufficient condition to ensure the avoidance of such problems is to deallocate an empty node only when no transaction that was active at the time when the node became empty is active anymore. Because of this delayed deallocation, this simple trick has been called the *drain technique*. Note that until its deallocation, an empty node would still have its forward link to the next leaf; so if a concurrent next operation steps on such a node, it would simply continue traversing the linked chain until it finds the next existing key.

9.4.2 Link Technique

In the lock-coupling technique with *next* operations, an important point was that range searches with a cursor on a page to be split do not need to prevent the split, but rather have their next *next* operation be prepared to follow the leaf link to the newly created sibling node. For example, assume that a leaf node contains keys 11, 12, 14, 15, 16, 17, 18, and 19, and that the last *next* call of a range search for the interval

$$15, 18$$

was positioned on key 16 and a concurrent $insert(13)$ operation splits the page moving keys 16, 17, 18, and 19 to a new leaf. When the range search continues with another *next* call, it would reacquire a lock on the split page but then immediately follow the link to the new sibling to proceed with key 17. Note that in this case there is no access-layer conflict between the range search and the insert.

This forward-stepping behavior built into *next* operations can be generalized and adopted by single-key search operations as well. The principle is that, whenever the key that one is looking for is not found on the page where it is presumed to reside,

the operation simply follows the forward-link chain of the leaves until it either finds the key or realizes that the key is definitely not present regardless of whatever concurrent reorganizations may blur the picture. This behavior is called the *link technique*, and it allows us to avoid holding multiple node locks simultaneously for both range searches and exact-match searches. So unlike lock coupling, during the descent in the tree, only the currently processed page needs to be kept locked, and no parent lock is needed to acquire a lock on a node. Likewise, range scans do not need to hold a predecessor lock before they can acquire a lock on the next leaf. The reasons why this seemingly spectacular relaxation still works correctly is that all tree reorganizations work by pushing existing keys "to the right", so that they will inevitably be found by every sufficiently persistent search operation. Furthermore, the ordering of keys along the leaves is guaranteed at all times, even during splits. Even if a range search is initially directed to a page, say p, which is split multiple times into p', p'' etc. before the range search actually accesses p, the worst thing that could happen is that the *lowkey* that the range search is looking for has been moved to some node p'' or p''' more than one "hop" away from where the search expected it to be. But even in this case, all the search needs to do is to follow the leaf chain until the first key that is equal to or smaller than its *lowkey* parameter.

Deletes do not present any problems if we again simply use the drain technique (and avoid getting into the hairy business of merging adjacent nodes). Insert operations, however, would still need to follow the lock coupling protocol, so that they are properly synchronized among each other. Putting the resulting protocol together into a set of locking rules is left as an exercise to the reader (see Exercise 8). Here we restrict ourselves to illustrating the lock requests and releases for our initial problem scenario with $search(31)$ being executed concurrently to $insert(30)$:

insert(30)	search(31)
	read lock r
	unlock r
	read lock n
	unlock n
write lock r	
write lock n	
unlock r	
write lock p	
	request read lock on p
allocate new page p'	
write lock p'	
split contents of p onto p and p'	
adjust contents of n	
unlock n, p, and p'	
	acquire read lock on p
	unlock p
	read lock p'
	return RID for key 31
	unlock p'

As a final word of caution, one should notice that, although the link technique may superficially appear strictly superior to lock coupling, there are cases where the link technique reveals limitations whereas lock coupling is more robust with regard to possible generalizations. One such generalization that is of practical relevance are range searches that deliver their result in descending order. Such a search would most naturally locate its *highkey* first and then traverse the linked list of leaf nodes in backward order. Obviously this would require a bidirectionally linked chain among the leaves, which is indeed what many commercial implementations of B$^+$-trees have. Now the fact that splits always reorganize "to the right" is no longer that helpful as "backward scans" proceed from right to left (assuming that left to right means ascending key order). In this setting, the link technique in the presented form does no longer work correctly. Lock coupling, on the other hand, can be made to work again with moderate effort.

9.4.3 Giveup Technique

Another interesting technique for synchronizing index tree traversals with local reorganizations like node splits is the so-called *giveup technique*. This is essentially an

optimistic variant based on the premise that conflicts are rare. So tree traversals on behalf of both searches and inserts or deletes proceed in their top-down descent without any locks, or merely holding a short-duration lock or latch on the currently processed node (but none of the operations ever holds locks on two different nodes simultaneously).

For detecting conflicts with a possibly ongoing reorganization like a node split, a special *range field* is maintained in the page header of each node (both leaves and inner nodes). This range field contains a lower bound an upper bound for the keys that could potentially reside in the subtree rooted at the given node. So when no reorganization is in progress, then for a node n with left sibling l and right sibling r such that the parent of n, l, and r has routing entries that form the ordered sequence $(\ldots, @l, x, @n, y, @r, \ldots)$ (with $@l$ denoting a pointer to node l) the range field should be the interval $(x, y]$. For example, for the stage 0 tree of Figure 9.6, the range of nodes r, n, p, and q are $(-\infty, +\infty)$, $(-\infty, 38]$, (18, 31], and (31, 38], respectively. These range fields are redundant information, but their point is that they can be maintained (namely, shrunk) by splits on a per node basis. So in stage 1 of the split shown in Figure 9.6 the range field of node p has been modified to (18, 27] and the range field of the new sibling p' has been set to (27, 31]. Note that these modifications are done, and can indeed be done, before the new routing key 27 is posted to p's parent n. Note that also that in this particular example, the range field of the inner node n stays the same, but when splits propagate across multiple tree levels range fields have to be adjusted at all affected levels, one node at a time.

The range fields of nodes enable a search operation to detect when a split (or, more generally, some tree reorganization) is in progress in the part of the tree that the search is traversing. When a top-down descent searching for key k is directed to a certain subtree and then finds that the range field of the subtree's root node n does not contain the search key k, this indicates that node n has been split but the necessary posting further up in the tree has not been completed. In this case the descent is simply rolled back and retried a short while later. We say that the tree-traversal operation "gives up" at this point, and repeats the entire descent, giving the ongoing split some time to be posted in the parent (or ancestors in general). Since such reorganizations should not involve disk I/O in the "critical phase", the retrial of the given-up traversal would typically have to wait for only a few milliseconds.

The giveup technique has obviously extremely low overhead, and it is fairly attractive in terms of its performance as long as tree reorganizations are infrequent. For example, when the workload mostly consists of searches with occasional inserts that may trigger splits only once in a while, then the giveup technique would perform very well. For workloads with many concurrent inserts and a higher rate of node splits, however, the two other techniques presented above are likely to perform better.

9.5 Further Optimizations

9.5.1 Deadlock-free Page Latching

Lock coupling at the page layer is still sufficiently close to the basic tree locking protocol of Chapter 4 that it can be implemented in a deadlock-free manner. Recall from Chapter 4 that deadlock-freedom can be guaranteed when all lock request patterns follow the node ordering of a tree or DAG. If, during the upward propagation of leaf splits, additional locks are acquired, this guarantee can no longer be given. So for deadlock-freedom either the version of lock coupling with lock releases only for split-safe nodes must be used, or splits have to be performed in a second tree traversal with more stringent locking upfront (as briefly discussed in Section 9.4).

Once deadlock-freedom is ensured, it is no longer necessary to manage the short-term locks on index pages in an explicit lock table. Rather a common trick to reduce the overhead of such special page locks is to use so-called *latches* instead of locks. Latches are light-weight semaphores for mutual exclusion of concurrent processes or threads within a process. Latches are usually implemented in the form of flags in page headers that are manipulated through special, indivisible machine instructions of the test-and-set style. Both shared and exclusive access can be provided this way. So in contrast to full-fledged semaphores as provided by the operating system or a thread package, latches do not necessarily support fair queueing or deadlock detection (although some variations in this regard are conceivable). Within the context of index concurrency control, using latches instead of page locks effectively turns the lock-coupling protocol into a latch-coupling protocol.

latch coupling

9.5.2 Enhanced Key-Range Concurrency

The incremental key-range locking protocol demands insert operations to acquire exclusive locks on both the inserted key (or key to which a new (key, RID) entry is added) and the previous key. These two locks serve to alert concurrent readers, i.e., search and next operations. However, they also prevent concurrent inserts in the locked key range even if these insertions refer to different keys. Obviously this is unnecessarily restrictive. An initial solution is relatively straightforward: we simply make the specific semantics of insert operations more explicit by introducing an *insert* lock mode and requiring inserts to request the two key-range locks in *insert* mode rather than exclusively. *Insert* locks on the same key ranges are compatible with each other, but incompatible with read locks. This allows concurrent key insertions without any blocking among each other. Note that this is perfectly legal even if two insert operations add entries to the same key. (Some subtleties may arise if the key to which the two inserts refer was not present at all in the index before the two operations were executed, but we do not delve into all these details here.)

fewer and less restrictive locks for inserts

Once we conceptually view the locking requirements for insert operations under the two facets of synchronizing inserts with other inserts and deletes and synchronizing inserts against concurrent readers, we should reconsider whether we may also relax the second type of synchronization to some extent. It is important for insert operations to realize when they would insert a new key in a key range that has already been visited by a concurrent but not yet committed reader; this is what requires the previous-key lock. However, when an insert operation precedes a reader, it is sufficient that the reader will eventually be blocked at the inserted key. In other words, reading the previous key should still be admissible. This asymmetry in the execution ordering of readers and inserts can be exploited as follows. The insert operation merely needs to check if a read lock is held on the previous key. If this is the case, then the insert must wait until the read lock is released, to guarantee serializability. If no read lock is currently held, however, the insert can proceed without really having to acquire a lock on the previous key. Technically, this protocol can be implemented by having insert operations acquire a so-called "instant-duration" lock in *insert* mode: once the lock is granted, it is immediately released again. Note that this relaxation holds only for the previous key; the *insert* lock on the newly inserted key must be held until commit.

instant-duration lock on previous key for inserts

Although it may seem intriguing to introduce an analogous relaxation also for delete operations, there is an inherent asymmetry between inserts and deletes that would lead to correctness problems if we applied the corresponding optimization to deletes. The asymmetry lies in the fact that a deleted key disappears from the index and thus will no longer cause concurrent inserts or readers to realize the conflict at the point of the deleted key itself. Therefore, locking the previous key until commit is crucial for deletes (at least in certain situations).

fewer locks for "deferred" deletes

The situation becomes different, however, if we could distinguish when a delete operation merely removes one RID from a key's RID list without becoming empty so that the key continues to exist in the index. This would indeed reestablish the symmetry with insert operations. From this observation we can derive a technique that we will refer to as "deferred" deletes, where delete operations never remove a key from the index even if they delete the last RID from the corresponding RID list. In such a situation, the key would be kept with an empty RID list, essentially serving as a marker for concurrent inserts and readers to realize conflicts. Some systems refer to such a marker as a "ghost key", which will be garbage-collected in the background at some appropriate later time, namely, when all transactions that may have ever seen the ghost key (or, for more efficient testing of this situation, the page on which the ghost key resides) are terminated. With this technique of deferred deletes it turns out that it is not even necessary to acquire an instant-duration lock on the previous key; rather the lock on the key to which the delete refers is fully sufficient to synchronize deletes versus inserts and readers.

Note that the write lock on a deleted key is not compatible with insert locks. Of course, we could introduce an operation-specific *delete* lock mode, but its compatibility with other lock modes would have to be as restrictive as an exclusive lock. Otherwise, allowing the insert of a new, seemingly non-existing key right after the deletion of that key and before the commit of the deleting transaction would incur correctness problems if the index has a uniqueness constraint and the deleting transaction aborts and thus needs to re-insert the deleted key.

In summary, the outlined optimizations relax the incremental key-range locking rules for insert and "deferred" delete operations as follows:

- An operation $insert(y, RID)$ requests an *insert* lock on y, to be held until transaction commit, and an instant-duration *insert* lock on the largest key smaller than y that is present in the index.

- An operation $delete(y, RID)$ requests a write lock on y, to be held until transaction commit.

A final opportunity for further optimization lies in distinguishing unique versus non-unique indexes, i.e., indexed attributes that either disallow or allow duplicate key values. It is fairly obvious that unique indexes may require more stringent concurrency control measures to preserve the uniqueness of keys in the presence of concurrent insert and delete operations. Conversely, non-unique indexes can be more liberal in terms of concurrency. A case analysis along these lines is left as an exercise to the reader (see Exercise 1).

9.5.3 Reduced Locking Overhead

Key-range locking at the access layer of an index is complementary to the locking of the actual data records, and it applies to all indexes affected by an operation. Thus, every index lookup for an exact-match query on a single indexed attribute incurs two object-level locks to be held for transaction duration, the index key lock and the record lock. For an insert or delete operation on a table with n indexes, $n + 1$ locks need to be maintained in total (not counting additional short-term locks on pages). Under certain circumstances the time and especially the space overhead of managing this number of locks may be critical. The alternative then is to combine the index-key and data-record locks for an operation into a single lock, possibly trading a potential loss of concurrency for reduced bookkeeping overhead.

RID locks only

A concrete protocol along these lines would treat the (key, RID) pairs of an index as the actual locking granules rather than locking keys or key ranges and RIDs separately, or would even lock only RIDs. In both variations, all index operations still need to follow the rules of the incremental key-range locking (or its optimized

versions). For example, a delete operation for RID 1133 with key value 25 has to acquire an appropriate previous-key lock in addition to the lock on the entry (25, 1133) itself. If this key has duplicates, say RIDs 1117, 1129, and 1137, the previous-key to be locked would be the entry (25, 1129); if there are no duplicates, then the last RID of the preceding key, for example, (22, 1135), would need to be locked. These locks could be managed in a lock table by the corresponding RIDs alone, effectively locking simply the RIDs 1133 and 1129 in the above scenario with duplicates or 1133 and 1135 in the scenario without duplicates.

RID locks can be interpreted either in conjunction with a specific index or globally with regard to all indexes. In the first case, the RID lock for an index lookup is relevant only for other operations on the same index, thus allowing concurrent updates on other attributes of the same record. However, this still requires n locks for an insert or delete operation on a table with n indexes. This overhead is further reduced by the second option, at the expense of disallowing concurrent updates on the same record regardless of which attributes are affected. So in this second case, only a single RID lock is required for inserting or deleting a record.

The main incentive for combining index key and record locks into RID locks is to reduce overhead, possibly at the expense of less concurrency. However, in specific cases this technique can even improve concurrency, because RIDs for keys with long RID lists are locked in a more incremental manner compared to the baseline version of incremental key-range locking.

9.5.4 Exploiting Transient Versioning

Yet another enhancement of index concurrency control at the access layer is to extend the presented incremental key-range locking protocol by keeping transient versions of index entries and exploiting them for read-only transactions along the lines of the ROMV protocol described in Chapter 5. Here the unit of versioning is a (key, RID list) pair. So each insert or delete operation leads to a new version of the index entry, and now it is mandatory that a delete operation that removes the last RID for a given key leaves a "ghost" entry as a special marker. For each index entry, the most recent version should reside directly in the appropriate index leaf page, whereas older versions may reside somewhere else and could, for example, be connected to their "successor" version in a doubly-linked chain. More details about the implementation of transient versioning in general can be found in Chapter 10.

9.6 Lessons Learned

Concurrency control for search structures like B$^+$-tree indexes may appear highly intricate upon first glance, because of the interplay between steps at the access layer and the page layer. By making these two layers explicit and separating their concerns, the state-of-the-art protocols become more comprehensible and easier to verify.

At the access layer, a limited form of predicate locks is needed to ensure the serializability of transactions. These locks refer to keys or key ranges, and can be cleverly encoded into what appears to be single keys by interpreting a key as a semi-open interval from one key in the index to the next higher existing key. Range queries should acquire such key-range locks incrementally for higher concurrency, hence the name incremental key-range locking. Further analysis shows that insert and delete operations then each have to acquire two of these key-range locks as they change the existing key ranges in the index. The presented version of such protocols required inserts and deletes to lock also the largest key smaller than the one inserted or deleted, i.e., the "previous key". In Section 9.5 we then discussed how this principal locking pattern at the access layer can be relaxed again by exploiting specific properties of insert and delete operations.

At the page layer short-term locks or merely page latches are needed to isolate the individual index operations, i.e., to ensure order-preserving serializability of the corresponding subtransactions. In particular, one has to guarantee that tree traversals arrive at the correct leaf in the presence of concurrent node splits. The main technique for this purpose, coined lock or latch coupling, builds on the tree locking protocol, presented earlier in Chapter 4, to avoid 2PL for subtransaction duration. The basic principle is to allow acquiring a lock on a tree node only if the lock requestor holds a lock on the node's parent (or leaf-level predecessor in the case of index leaf scans); other than that, locks can be released as early as possible and do not have to follow the 2PL discipline. For insert operations, the possibility of leaf splits that propagate upward in the tree requires additional care in that locks on "unsafe" nodes that may possibly be affected by a split must be held further on. Alternatively, inserts that trigger splits could release all page locks or latches and simply initiate a second tree traversal with more stringent locking upfront. This approach can be implemented in a deadlock-free manner, so that light-weight latches may be used instead of explicit page locks. Other concurrency control techniques for the page layer of B^+-tree-like indexes include the link and giveup approaches, and also various combinations of these techniques are feasible. Although we did focus our presentation on B^+-trees as the prevalent form of search structures in data servers, many of our considerations can be carried over to a broader class of search structures including indexes for multidimensional range queries or similarity search (with additional adjustments, however).

Exercises

1. Discuss how the incremental key-range locking protocol on index keys has to be changed for a unique index. Discuss also how unique and non-unique indexes may be treated differently by the various optimizations sketched in Section 9.5.

2. Discuss which lock(s) need to be acquired by an unsuccessful exact-match query for a key that is smaller than the smallest key that exists in the index. How can you ensure that a subsequent insertion of this particular key "steps" on a lock conflict?

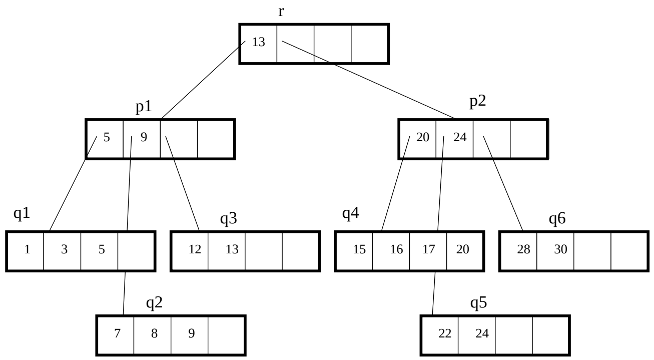
3. Incremental key-range locking based on read and write locks on index keys, as explained in Section 9.3, does not allow two insert operations in the same, previously unoccupied key interval to proceed concurrently. For example, in the index of Figure 9.5, an insert(27) would block a later insert(28) operation, although none of these keys are initially present and the two operations should be considered as commutative. Discuss whether the resulting blocking of one of the two operations can be safely avoided, and how this could be accomplished. Also discuss the analogous situation for delete operations, and the combination of insert and delete operations.

4. Discuss the locking requirements for an unsuccessful insert operation on a unique index, i.e., when the key to be inserted already exists in the index.

5. In the presented form of the incremental key-range locking protocol of Section 9.3, insert and delete operations are required to lock previous keys (hence the protocol's alias name "previous-key locking"). There is a dual variant that requires an operation of these two types to request a next-key lock, i.e., a lock on the interval that immediately follows the interval that is primarily affected by the operation. Elaborate the detailed locking rules of this dual variant, which is known as *next-key locking* in the literature.

6. Consider the following B^+-tree index on the attribute AccountNumber of an Accounts table. Assume that all tree nodes have a space capacity to hold up to four entries.

   Which locks need to be requested for the execution of the following transaction, assuming the incremental key-range locking at the access layer lock coupling at the page layer?




```
begin transaction;
      Select Count(*) From Accounts
           Where AccountNumber Between 11 And 25;
      Insert Into Accounts (AccountNumber, ...)  Values (27, ...);
commit transaction;
```

7. Consider a situation where a node p in a B$^+$-tree is not split-safe, but has a split-safe descendant q, with both p and q on the traversal path of an insert operation. Discuss how the lock-coupling technique should handle this situation to minimize lock contention.

8. State explicit rules for lock requests and lock releases for the link technique (in a form similar to our presentation of locking rules for the lock-coupling technique).

9. Assume that the leaf nodes of a B$^+$-tree index are linked together bidirectionally (rather than only in forward direction). This may be useful to support range scans in descending key order, which in turn is of interest for returning qualifying records in descending sort order (as requested, for example, by a decision-support query). Discuss the consequences of this bidirectional link-

age with regard to the lock-coupling and the link techniques for the page-level concurrency control. If possible, generalize these techniques to accommodate the new feature.

Bibliographical and Other Remarks

Historically, page-level concurrency control for single B$^+$-tree index operations was studied first, with Bayer and Schkolnick (1977) and Kung and Lehman (1980) as well as Lehman and Yao (1981) being the first papers on the lock-coupling and link techniques, respectively. Kwong and Wood (1982) as well as Mond and Raz (1985) contributed to the improvement of lock coupling, and Manber and Ladner (1982) as well as Sagiv (1986) further enhanced the link technique. The textbook by Gray and Reuter (1993) describes a more elaborated version of lock coupling under the name "lock crabbing". Shasha and Goodman (1988) finally came up with a taxonomy for the various techniques and proposed a formal framework for thorough reasoning on the correctness of these subtle algorithms.

Shasha (1985) also realized that the page-level measures for a single index operation were only one aspect of index concurrency control, and already suggested a layered approach in the direction of what we have presented in this chapter. The current state-of-the-art solution based on incremental key-range locking for the access-layer concurrency control has mostly been developed by Mohan (1990, 1995); the fundamentals of this work are described also in the textbook by Gray and Reuter (1993) who further credit additional independent work of other architects of commercial database systems which employ similar methods. Gray and Reuter describe the variant that we have referred to as previous-key locking, whereas Mohan prefers the next-key locking dual variant. Mohan presents full-fledged algorithms, coined ARIES/KVL and ARIES/IM, in great detail, based on highly optimized versions of incremental key-range locking at the access layer and a combination of latch coupling (on index pages) and the giveup technique (using a "structure modification" bit in each page and a global tree latch) at the page layer. The ARIES/IM variant uses the RID-only locking optimization for reduced overhead. Mohan (1995) also provides solutions for several of the exercises.

Optimizations that strive for reduced locking overhead while retaining the high concurrency of incremental key-range locking have been developed by Mohan (1992, 1995) and Lomet (1993). The latter also discusses additional concurrency enhancements that can be achieved by treating key ranges and individual keys as separate resources for locking purposes. Generalizations towards applying such high-performance methods to multidimensional and other search structures beyond B$^+$-trees have recently been proposed by Lomet and Salzberg (1997) and Kornacker et al. (1997). Experimental and analytical performance studies of B$^+$-tree concurrency control methods have been conducted by Srinivasan and Carey (1991) and Johnson and Shasha (1993).