Chapter 8
B-Tree Structure Modifications

A B-tree structure modification is an update operation that changes the tree structure of the B-tree, so that at least one index record (a parent-to-child link) is inserted, deleted, or updated. The structure modifications are encompassed by the following five types of primitive modifications: page split, page merge, records redistribute, tree-height increase, and tree-height decrease. Each of these primitive modifications modifies three B-tree pages, namely, a parent page and two child pages.

Structure modifications are triggered by insert or delete actions on leaf pages that cannot accommodate the action: an attempt to insert into a leaf page that has no room for the tuple to be inserted or an attempt to delete from a leaf page that would underflow by the deletion. In such cases, to make the insertion possible, a sequence of one or more page splits, possibly preceded by a tree-height increase, is needed, and to make the deletion possible, a sequence of one or more page merges or records redistributes, possibly preceded by a tree-height decrease, is needed.

In this chapter we show how B-tree structure modifications are managed in the ARIES-based transaction-processing environment developed in the previous chapters, using the traversal algorithms presented in Chap. 7. Following the principle of redo-only structure modifications outlined in Sects. 3.5 and 4.11, we give algorithms for the five primitive structure modifications. We also show how sequences of these modifications, when performed in a top-down fashion, retain the B-tree in a consistent and balanced state during normal transaction processing with any number of concurrent forward-rolling and backward-rolling transactions and that in the event of a process failure or a system crash the B-tree is brought back to a consistent and balanced state in the redo pass of ARIES recovery.
8.1 Top-Down Redo-Only Modifications

In Sects. 7.4 and 7.5, when discussing B-tree traversals for insertions and deletions, we presented the procedures \textit{find-page-for-insert}(p, p', x, v, y) and \textit{find-page-for-delete}(p, p', x, y) (Algorithms 7.8 and 7.9) that locate the leaf page \( p \) covering key \( x \) and the leaf page \( p' \) containing the next key \( y \) and also arrange that page \( p \) can accommodate the insertion or deletion of the tuple with key \( x \).

In the case of an insertion, if it is found that the leaf page covering key \( x \) does not have room for the tuple \((x, v)\), the calls

\[
\text{start-for-page-splits}(p, x, v) \\
\text{top-down-page-splits}(p, x, v)
\]

are performed in order to make room for the tuple. The former call returns the page-id \( p \) of the highest-level page on the root-to-leaf path to the leaf page covering \( x \) that needs modification (page split or tree-height increase). The latter call then performs the splits, in top-down order, starting at the child of page \( p \) on the path.

\textit{Example 8.1} In the case of Fig. 8.1a, the call \textit{start-for-page-splits}(p, x, v) returns with \( p = p_2 \), because pages \( p_3 \) and \( p_4 \) have to be split. Figure 8.1b shows the situation after calling \textit{top-down-page-splits}(p_2, x, v) and inserting the tuple into page \( p_4 \). First, page \( p_3 \) was split by allocating a new page \( p'_3 \), moving the upper half of the records in \( p_3 \) to \( p'_3 \) and linking \( p'_3 \) as a child of \( p_2 \). Second, page \( p_4 \) was split by allocating a new page \( p'_4 \), moving the upper half of the records in \( p_4 \) to \( p'_4 \) and linking \( p'_4 \) as a child of \( p_3 \). Third, the tuple \((x, v)\) was inserted into page \( p_4 \) that now has room for the tuple. \hfill \Box

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{page_splits.png}
\caption{Page splits are needed to accommodate the insertion \( I \{x, v\} \) because the tuple \((x, v)\) does not fit into the leaf page that covers key \( x \). (a) Before insertion (b) After insertion}
\end{figure}