Overview

Prior approach to B-Tree leaf page merging retained copies of the merged records on the left page (L) after the merge, so that backward scans already positioned on L could read them without returning to the right page (R). This produced duplicate ("ghost") copies of the same records on both L and R, which resulted in a corrupted index.
This patch eliminates ghost records entirely. After a merge, L becomes an empty tombstone (the MA page); no records are retained on it. Instead, scans are taught to detect the MA page and navigate correctly to R to retrieve the merged records.
The patch also relies on `currPos.items`, which retains the records of the last read page so they remain available for processing even after the scan moves on. To process the merged pages group before `currPos.items` is overridden by the next page read, the records are saved into a separate array and used for elimination.
To make this navigation safe and unambiguous across concurrent operations, the patch assigns each merge operation a unique group identity encoded as the MA page's block number, which is stamped on every merged member page (M pages) belonging to that group.

1. Group Identity & Page Format Changes

A merge can only involve two adjacent pages sharing the same parent; records always move from the left page (L) to the right page (R), which is marked MERGE (M). L is marked MERGE_AWAY (MA) and has its content replaced entirely with a single safemergexid value, which VACUUM uses to determine when the group is safe to reclaim. To uniquely identify the group, the MA page's block number is stored in the otherwise unused pd_prune_xid field of every M page in the group.
A merge group must be contiguous: no normal page may appear between the MA page and its M pages, as that indicates corruption. If an M page later splits, the new right page inherits the M flag and the same group ID, which is the only way a group can grow beyond a single M page. M pages may also be deleted at any point without breaking the design since they retain the M flag and are skipped by scans. The MA page, however, cannot be deleted until all M pages in its group have been cleaned up first.

2. Scan Navigation & Merge Recovery

State Variables

The scan state is extended with five new fields in `BTScanOpaqueData`: `skipMergeRecovery, needMergeRecovery, savedMergeTids, nSavedMergeTids,` and `mergedAwayBlkno.` Together they track whether the scan is inside a merge group and whether it needs to filter duplicate tuples.

The Role of currPos.items

Every call to `_bt_readpage` overwrites `currPos.items` with the tuples from the newly read page. When merge recovery is triggered, `_bt_copylastreadpagedata` copies all heap TIDs from the current currPos.items into `savedMergeTids` and sorts them before the array is overwritten. Later, `_bt_removeduplicates` filters each M page's items against this sorted array using binary search, removing any TID already returned to the caller.

Forward Scan

```
[MA] <--> [M] <--> [M] <--> [normal]
  L         R        A
```

If the scan sees the MA tombstone before any M page, it sets `skipMergeRecovery = true` and records `mergedAwayBlkno`, then skips the tombstone and reads the M pages normally with no filtering needed. If instead the scan had already read L before the merge happened and now lands on R with no tombstone seen, it saves L's TIDs from `currPos.items` into `savedMergeTids`, sets `needMergeRecovery = true`, and calls `_bt_removeduplicates` on R and every subsequent M page in the group.

Backward Scan

```
[MA] <--> [M] <--> [M] <--> [M] <--> [normal]
  L         R        A        B          C
```

The normal backward case is that the scan reads B, A, and R first (each sets `skipMergeRecovery = true`), then passes the MA tombstone stepping left without any extra work.

The hard case is when a merge happens after the scan has already read pages on the right side and the scan steps left onto the MA tombstone without having seen any M page. In this case, the scan saves the TIDs it already has via `_bt_copylastreadpagedata`, sets `needMergeRecovery = true`, and triggers a rightward walk via `_bt_find_merge_tail` to locate the rightmost M page in the group (B in this example). The scan then walks backward through B, A, and R, calling `_bt_removeduplicates` on each page before returning tuples to the caller.

Race Condition: Mismatched Merge Groups

A subtle race condition arises when the scan reads M pages from one merge group, but by the time it steps left onto an MA tombstone, that tombstone belongs to a different merge group entirely. This can happen if the original MA page was deleted and the pages to its left underwent a new merge.

The group identity solves this: every M page stores its MA tombstone's block number in `mergedAwayBlkno`. When the scan reaches an MA page, it compares the page's own block number against the saved `mergedAwayBlkno`. If they do not match, both `skipMergeRecovery` and `needMergeRecovery` are reset, and the scan treats the tombstone as belonging to a fresh merge group.

3. VACUUM and Flag Lifecycle

Transaction Horizon

When VACUUM visits an MA tombstone page, it reads the `safemergexid` stored in the page's content and calls `GlobalVisCheckRemovableFullXid` to check whether the transaction horizon has passed. If it has not, the page is left untouched. If it has, VACUUM begins the group cleanup sequence.

The Cleanup Traversal

```
[MA] <--> [M] <--> [M] <--> [M] <--> [normal]
  L         R        A        B         C
 pin                        tail
```

VACUUM immediately drops the write lock on the MA page but retains its pin. This is necessary because the forward and backward walks will acquire write locks on M pages. Holding the MA lock across multiple buffer lock acquisitions would violate PostgreSQL's lock ordering rules and risk deadlock. Since no other VACUUM can run concurrently on the same index and no scan will modify the MA page, the page remains stable after the lock is dropped.

VACUUM then walks forward from `btpo_next` using lock coupling to locate the tail, which is the rightmost M page that still belongs to the same group. Each page is verified via `BTPageIsMergedMember`, which checks that the stored MA block number in `pd_prune_xid` matches the current MA page's block number. Once the tail and the right anchor (the first normal page to the right) are recorded, the forward walk is complete.

VACUUM then walks backward from the tail, acquiring a write lock on each M page and clearing the `BTP_MERGED` flag and the stored MA block number. It steps left after each page, stopping just before the MA tombstone. This backward direction guarantees the group invariant holds at every step. If VACUUM is interrupted during the walk, the pages to the right that were already cleared are now normal pages, and the remaining M pages to the left still correctly point to the MA tombstone. A normal page is therefore never stranded between an MA and an M page.

If any page encountered during the backward walk is not a member of the group (wrong MA block number), VACUUM logs a warning and defers the remaining cleanup to a future VACUUM run via goto `skip_merge_cleanup`.

Finalizing the Tombstone

Once all M flags are cleared, VACUUM acquires an exclusive write lock on the MA page again and verifies that the page has not changed concurrently (checking both the `BTP_MERGED_AWAY` flag and the `safemergexid` value). It then initializes the page again with PageInit, installs a dummy truncated high key (required by `_bt_pagedel` stage 2, which needs a high key to search for the parent downlink), and clears `BTP_MERGED_AWAY | BTP_HAS_FULLXID` while setting `BTP_HALF_DEAD`. From this point, the standard page deletion routine (`_bt_pagedel`) handles the rest of the deletion process.