Problem Statement

One of PostgreSQL's greatest architectural deficiencies is that tables and indexes not infrequently become bloated, and this bloat is difficult to reverse once it has occurred. When a row is updated, a new row version is created and the space occupied by the old version can be eventually be recycled. Since there is always a short delay and sometimes a very long delay (hours) between the time when the new row version is created and the time the old row version can be removed, tables grow to accommodate storing both the old and new row versions. Indeed, it can sometimes be necessary to store many versions of the same tuple, rather just two (old and new). When the old versions are reclaimed, there is no easy way to shrink the table, because the free space is spread through the table's data blocks rather than concentrated at the end of the file. Moreover, since the space used by old row versions are only reclaimed lazily -- either when a page is accessed (HOT pruning) or by periodic VACUUM operations -- the table may be further extended even space for the new row version could have been made available by more timely cleanup.

Bloat also affects indexes. Inability to consolidate free space is a factor here just as it is for tables, although at least in theory there is more latitude to reorganize indexes by splitting and combining pages. Similarly, the fact that dead row versions are not necessarily removed from indexes at the soonest possible time can cause additional index growth. In additional to these two problems which are broadly analogous to those that affect tables, indexes also suffer from a third problem: any non-HOT update to a heap tuple requires an update to every index even if the values of the indexed columns have not changed. Broadly, index bloat is a less serious problem than table bloat, because in many cases it is possible to create a new index and drop the old one without affecting foreground operations, whereas removing table bloat generally requires a full table rewrite under an ACCESS EXCLUSIVE lock.

Possible Approaches

There are two obvious approaches to attacking these problems. The first is to make it easier to reorganize the table. For example, in an index-organized table -- one where the heap's physical storage is clustered by the primary key index and secondary indexes reference the primary key rather than the tuple's physical location -- tuples can be moved around to consolidate internal freespace. Other approaches to the problem of simplifying table reorganization are also possible. The second approach is to somehow segregate old row versions from current row versions, such as by placing them in a separate data structure from the table itself, so that cleanup can happen more efficiently and without expanding the table itself. While there is probably useful work to be done in both areas, the second approach seems superior overall, because the first approach only makes it easier to salvage a table where internal free space has become badly fragmented, while the second avoids creating such a table in the first place.

DO-UNDO-REDO

To see how such a system might work, let's first consider the general concept of UNDO. Most database systems have both REDO -- what PostgreSQL typically calls the write-ahead log (WAL) -- and UNDO. REDO is responsible for restoring physical consistency after a database crash, while UNDO is responsible for reversing the effects of aborted transactions. When a transaction performs an operation (DO), it also writes it to the write-ahead log (REDO) and records the information needed to reverse it (UNDO). If the transaction aborts, UNDO is used to reverse all changes made by the transaction. Both the actions performed (DO) and the UNDO created for those actions are protected by REDO. Thus, UNDO must be written to disk at checkpoint time; UNDO since the most recent checkpoint prior to a crash is regenerated by WAL replay (REDO). So, after a crash, we first replay WAL (REDO) and then reverse out any transactions implicitly aborted by the crash (UNDO). This basic DO-UNDO-REDO protocol has been well-understood for decades, but PostgreSQL does not currently use it. It would likely be worthwhile to implement DO-UNDO-REDO in PostgreSQL independently of any design for reducing bloat, because it would provide a systematic framework for handling cleanup actions that must be performed at the end of recovery. For example, if a transaction creates a table and, while that transaction is still in progress, there is an operating system or PostgreSQL crash, the storage used by the table is permanently leaked. This could be fixed by having the operation that creates the table also emit UNDO which would unlink the backing storage in the event of an abort.

Proposed Approach

Let's create a new type of heap where we emit UNDO for each INSERT, UPDATE , or DELETE. For an INSERT, we will insert the new write and emit UNDO which will remove it. For a DELETE, we will immediately remove the old row version but emit UNDO which will put it back. For an UPDATE, we can proceed in two ways, depending on the situation. First, we can handle an UPDATE much as if it were a DELETE combined with an INSERT, removing the old version, adding the new one, and emitting UNDO to reverse both changes. Second, in some cases, we can instead perform an in-place update, modifying the existing record in place and emitting UNDO to restore the old version. It will be desirable to use the second strategy as often as possible, because an in-place update does not bloat the heap.

A few areas need further discussion. First, there is the problem of making MVCC work. Scans may need to see old row versions that have been removed from the heap; they will need an efficient way to find these old row versions in the UNDO. This will also require us to consider how UNDO is organized and for how long it must be retained, as well as how scans will find the UNDO records which are relevant to them. Second, the question of when we can safely use in-place updates needs further discussion; see below.

UNDO Organization And Retention

We could have a single shared undo log to which all transactions write, but the insert point would almost certainly become a point of contention. Moreover, it's not very desirable to

interleave UNDO from different transactions, because it makes cleanup difficult. Suppose one transaction aborts while another continues to run; if their UNDO records are interleaved, it will be difficult to recover any space if one set of UNDO records can be discarded but the other must be kept. Instead, it seems best to have a separate UNDO log for each transaction. Each of these UNDO logs individually will be written sequentially and will consist of a series of variable-length records.

Since the only way to find an old row version is to look at UNDO, we will need to retain UNDO for as long as it might contain row versions that some overlapping transaction needs to see. This means that we must retain UNDO for (1) all transactions which are in progress, (2) for aborted transactions until such time as all of the UNDO actions have been performed, and (3) for committed transactions until they are all-visible. Note that (1) and (2) would be required no matter what; UNDO is fundamentally a system for tracking actions to performed during transaction abort. (3) is only required because of our desire to keep old row versions out of the heap.

When a particular UNDO log is no longer needed, we can remove the whole thing at once. Unless the UNDO log spilled to disk because it was large or because of a checkpoint during the transaction, this is just a matter of deallocating or recycling whatever shared memory we're using to store it; otherwise, we'll need to remove or recycle the file, or the portion of a file, that was used to persist it on disk.

Page-Level MVCC

PostgreSQL's current MVCC implementation requires each tuple to carry a very wide header. Each tuple has carries a 24-byte header -- or wider if the null bitmap is large -- and 18 of those bytes plus assorted flag bits are there to service the needs of the MVCC implementation. For static data, all of that space is wasted. Moreover, we can easily end up dirtying the same page multiple times to set hint bits and, later, to freeze XMIN and XMAX, resulting in repeated page writes. We can do better. If all UNDO for a given page have been removed as described in the previous section, then every tuple on that page is all-visible. Therefore, the page itself does not need to contain any per-tuple visibility information; it only needs to be possible to find any UNDO pertaining to that page. Conveniently, the UNDO will be removed at exactly the same time that the visibility information is no longer needed, so we can store any required visibility information in the UNDO itself.

Just as a particular WAL record is referenced using a byte position, a particular UNDO record is referenced by uniquely identifying the transaction and the byte position within that transaction's UNDO log. To uniquely identify a particular transaction, it seems we will need 8 bytes: the 4-byte epoch and the 4-byte XID. To uniquely identify a position within that transaction's UNDO log, we will perhaps need another 8 bytes. Thus, in total, an UNDO pointer will be 16 bytes. We could perhaps reduce the size to 12 bytes by stipulating that a transaction can only write

4GB of UNDO using a particular XID; if it needed to generate more UNDO than that, it would need to allocate 1 additional XID for every 4GB of UNDO generated.

Pages in this new type of heap will have space to store an UNDO pointer (epoch + XID + byte offset) in the page header. UNDO pointers where the XID is 0, 1, or 2 cannot refer to a real UNDO log, because those XIDs are reserved by PostgreSQL. Accordingly, we reserve the all-zeroes UNDO pointer as an invalid UNDO pointer. Also, we reserve all UNDO pointers where the XID is 1 as temporary page data (TPD) pointers. When a page has been recently modified by only one transaction, the page's UNDO pointer points to the most recent UNDO record generated by that transaction for that page. There's no need to reset the pointer when the UNDO log is removed; instead, we interpret a pointer to an UNDO log which no longer exists as a sure sign that the page is all-frozen. When a page has been recently modified by more than one transaction, the page's UNDO pointer is a TPD pointer.

A TPD pointer is a 64-bit reference to a TPD record. An UNDO pointer will be either 12 or 16 bytes, so there's definitely room to store 64 additional bits there in addition to the XID of 1. TPD records are stored in the TPD log, which is managed much like the write-ahead log: records are accessed by byte offset, the log begins at byte position 0 and grows upward forever, and the older portions of the log can discarded once they are no longer needed. Unlike WAL, however, an already-written TPD record can be modified in place. New TPD records and changes to TPD records must be protected by WAL. A TPD will contain one UNDO pointer per recent transaction (in progress, aborted but not yet cleaned up, committed but not yet all-visible), indicating the most recent change to the page by that transaction. Therefore, it's always possible to find all of the recent transactions which have modified a page and the most recent change by each one. Each UNDO record for a particular page should contain a back-pointer to the prior change to that same page by the same transaction, if any, so once we've found the most recent change to the page by each recent transaction we can use these back-pointers to find all of the other ones as well. And that lets us find any old tuple versions we need.

If a transaction modifies a page whose existing UNDO pointer is invalid, or one whose existing UNDO pointer points into its own UNDO log, it can just overwrite the existing UNDO pointer with a pointer to the new change. Likewise, if the UNDO pointer is a TPD pointer and the modifying transaction is already present in the TPD record, it can just update the TPD record with the UNDO pointer for the new modification. Alternatively, if the UNDO pointer is a TPD pointer and one of the transactions in the TPD is no longer recent, it can grab that slot in the existing TPD for its own UNDO pointer. However, if the page points to some other transaction's UNDO log and that transaction is still recent, or if the page points to a TPD all of whose members are still recent, then the modifying transaction must create a new and larger TPD for the page. The old TPD can then be marked as garbage and reclaimed. Otherwise, a TPD entry can be reclaimed when every UNDO pointer it contains points to an UNDO log which has already been discarded. We can treat a page with a pointer to a TPD entry which has been discarded just like a page with a pointer to an UNDO log that no longer exists: the page is all-visible.

In-Place Updates

In-place updates are fairly obviously safe and possible when no index columns have been modified and the new tuple is no larger than the old one. The indexes don't need to know anything about the change, and the new tuple will definitely fit. In other cases, a bit more thought is needed.

If the new tuple is larger than the old one, it may still be possible to do the update in place. The simplest case is when there happens to be unused space following the existing tuple, and the new and larger tuple will fit into that space. In that case, we do not need to do anything special. Alternatively, it may be that there's enough free space elsewhere on the page to accommodate the new tuple. In that case, we could take a cleanup lock and reorganize the page. However, if we do, we must be careful to leave enough space to permit a successful UNDO. For example, imagine a completely full page where we try to shrink one tuple in one transaction and enlarge a different tuple in some other transaction. If we allow both updates to be in place, we will have a serious problem if the shrinking transaction aborts and the enlarging transaction commits.

If any indexed columns are changed, an in-place update might confuse the index AM. Index AMs like BRIN which do not store TIDs don't care, but those that do will get confused, because there will now be entries for multiple index values pointing to the same TID. That could be handled by forbidding index-only scans and requiring rechecks in all cases, but that would probably be quite inefficient. Instead, we would likely wish to adopt the solution used by other database systems which work on principles similar to those described here: whenever we update or delete a tuple, use the old tuple to re-find any index entries that might no longer be valid, and apply a delete-mark to them. When scanning an index, perform rechecks for all delete-marked index items. Alternatively, one could perform index updates only when no indexed columns have been changed.

If we adopt the delete-marking system, index vacuuming requires only scanning the index for delete-marked tuples and removing all of those that no longer point to a tuple or which point to a tuple that does not match the index. This would be substantially better than our current system, since it could be performed incrementally and wouldn't require a heap scan or working memory to hold a list of dead TIDs. If we don't adopt the delete-marking system, indexes must be vacuumed as they are today.

Advantages and Disadvantages

Performing updates in place where possible prevents bloat from being created. It does not prevent all bloat: aside from any updates that are not dead in place, a transaction that inserts many rows and aborts or deletes many rows and commits will still leave the table larger than the optimum size. However, many workloads have more updates than they do inserts or deletes, so this is probably a significant win.

Old tuple versions are removed from the heap eagerly (by UNDO cleanup at the end of a transaction) rather than lazily (by HOT pruning and VACUUM). This increases the chance of being able to reuse internal free space rather than extending the relation. If we adopt the delete-marking system, index vacuuming gets cheaper, too.

Reading a page that has been recently modified gets significantly more expensive; it is necessary to read the associated UNDO entries and do a bunch of calculation that is significantly more complex than what is required today. The problem is more severe when TPD entries are required. There are probably some steps that can be taken to mitigate this problem, such as setting aside a bit per tuple to mean "this tuple has not been recently modified", or caching additional details in the TPD when one is used.

Most things that could cause a page to be rewritten multiple times are eliminated. Tuples no longer need to be frozen; instead, pages are implicitly frozen by the removal of associated UNDO. Similarly, hint bits are gone. The page-level all-visible bit needs some thought; we'll still need a visibility map to support index-only scans, but we probably don't want the page level bit any more, because rewriting the whole heap just to set that one bit would be terrible.

If we adopt the delete-marking system, deletes are more expensive, because they have to delete-mark all of the index entries. Updates are more expensive when all of the index columns change, because now each index needs to be updated in two ways (new index entries and delete-marking of old ones) rather than one. However, if some indexed columns are updated but only a small minority of the indexes on the table include those columns, updates are substantially cheaper, because new index entries and delete-marking of old ones are required only for those indexes where some column has been updated.

Delete-marking assumes that we’re able to re-find index tuples. The system today does not require such an assumption.

Tuple headers become much smaller. The page header becomes bigger, but there's only one page header per page, and there are many tuple headers per page, so on average we come out ahead.

Transaction abort can be lengthy if much UNDO is required. This problem can be mitigated by pushing the UNDO steps into the background.