In an earlier document, subsequently posted to pgsql-hackers, I laid out the overall idea of a new heap backed by undo logging. The overall goal was to achieve better control over bloat. By storing old tuple versions in undo logs rather than in the heap itself, it becomes possible to discard old row versions without reorganizing the heap. By organizing undo as a set of per-transaction logs rather than a shared, interleaved log, all related undo entries can be discarded together and without creating fragmentation problems. However, that document only discussed the requirements for an undo log at a very broad level. The purpose of this document is to discuss the requirements for the undo subsystem in more detail.

Overall Design

As previously noted, each transaction should have a separate undo log. Because references to UNDO logs may persist long after the undo logs themselves are gone, undo logs should always be referenced using 64-bit transaction IDs; that is, the 32-bit epoch should be included along with the 32-bit XID. Since PostgreSQL has a txid data type that behaves this way, I will use the term txid to refer to a 32-bit XID augmented with a 32-bit epoch. Logically, one can think of an undo log as being named by a txid; that is, given a txid and no other information, it should be possible for the undo subsystem to locate all undo records pertaining to that transaction.

To find an individual undo record, it will be necessary to identify not only the transaction which created the record but also the position of the record within that transaction's undo log. We identify the position of a record within a transaction's undo log using a 64-bit byte offset. Therefore, a complete undo pointer is 16 bytes: an 8-byte txid, and an 8-byte offset within the undo log for that transaction. I will refer to this 16-byte quantity as an undo_pointer. When the txid is clear from context, it need not be stored; in that case, a single 8-byte quantity can be used to identify the position of an undo record. I will refer to this as an undo_offset.

It would be possible to reduce the size of an undo_pointer from 16 bytes to 12 bytes if we restricted a single txid to at most 4GB of undo. A transaction would needed to generate more than 4GB of undo could allocate an additional XID for each additional 4GB of undo. Since very few transactions will generate more than 4GB of undo, this would not increase XID consumption significantly. However, it wouldn't save very much space and would probably complicate the code significantly, so it is probably not worth it.

When a transaction aborts -- whether by explicit user action or implicitly because of an error or as a side-effect of a system crash, any undo actions pertaining to that transaction need to be performed. This needs to be done by reading that transaction's undo log in reverse order and calling an appropriate handler for each record. In the event of a crash, undo actions need to be performed only after WAL replay (i.e. redo) is complete, and only for transactions whose undo logs still exist at that point in time.

It is a bad idea for transactions to try to perform their own undo actions during transaction abort. An obvious problem with doing so is that it makes transaction abort slow. As far as possible,

we'd like abort cleanup to happen in the background, not synchronously while the user waits. A subtler but more serious problem is that undo actions might fail (e.g. because a disk block cannot be read). We cannot easily tolerate additional errors when the transaction is already in the process of aborting.

Instead, it seems best to imagine that the transaction aborts just as it does today, and then at a later time (but as soon as possible) we execute a separate transaction which attempts to perform the undo actions associated with the abort. If the undo transaction fails, it can be retried. When the system is not in single-user mode, undo actions can be performed by one or more background workers (the number can be configurable) dedicated to this purpose. In single-user mode, the system should begin performing the undo transaction as soon as the abort of the original transaction is complete. Since there is only one process, the user will have to wait for that process to complete abort processing before starting a new foreground transaction.

Source Code Organization

I suggest that we store the code related to undo in src/backend/access/undo; headers in src/include/access. Within that directory, I think the external APIs for this module can be more or less divided into three parts:

- Undo record management, including buffer management, durability, write ahead logging integration, lookup of existing records, and truncation and removal of undo logs.
- Undo processing, including management of background workers and execution of undo actions either by background workers or (when necessary) by the foreground process.
- Undo construction and decoding, or in other words routines that have to do with the contents of particular records rather than the way that the records are collected and stored.

Possibly a more granular division is needed, but I think it is important in any case that the divisions between these submodules are crisp. For example, the code that implements undo processing should be calling undo record management functions to obtain the undo records to be replayed and to remove undo logs that are no longer needed, and it should be calling undo decoding routines to understand what needs to be done for each record. It should not have ad-hoc logic to do those things within undo application per se.

Buffer Management and Durability

In general, undo generation needs to happen atomically with the operation to be undone; therefore, the same WAL record that covers the action itself must also cover the generation of related undo. This requires that pages which are part of undo segments have a standard page header containing an LSN which must be flushed to disk before the associated undo buffer can be flushed to disk. In this sense, undo pages need to be treated much like heap or index pages.

In fact, it seems very desirable to treat undo pages like heap and index pages not just in this particular way, but in general. It will be useful for pages containing undo data to have buffer content locks, buffer I/O locks, and pins just like any other data page. It is desirable that they occupy storage taken from the same pool of buffers, so that the user does not need to manually configure separate pool sizes for undo pages and other data. It is also seems likely to be desirable that they should follow page eviction rules similar or identical to those for data pages of other types (although it's possible that some changes will be needed). Indeed, even checkpointing needs to work in pretty much the same way: it's desirable to avoid writing out undo pages for as long as possible, but at checkpoint time we must do so.

Despite all of these similarities, there are some important differences as well. First, undo pages are identified by a txid, not a standard relfilenode. Therefore, they probably need their own equivalent of the buffer mapping table. Second, it's important to postpone creating undo files on disk for as long as possible. Even for regular heap and index files, it would be possible to postpone creating the file on disk until at least one block actually gets evicted; in some cases, such as for short-lived temporary tables, the performance benefits of such a change might be substantial. For undo files, this is far more important. Transactions will create undo logs which, in many cases, are only needed for very short periods of time. If no checkpoint intervenes, the undo log can be logically created and then destroyed in memory without ever performing any file system operation. This optimization will be very important for good performance. In addition, it may be useful in some cases to combine several small undo logs into a single file to reduce the number of filesystem metadata operations. For undo logs which are (or become) large, it seems wise for each undo log to be stored in a separate file, perhaps divided into 1GB segments just as we do for ordinary relation files. But a checkpoint which catches 100 open transactions each with 8kB of undo will probably find it better to write a single 800kB file (or a little more, because of included metadata) than to write 100 separate 8kB files each of which will require a separate fsync().

Write-Ahead Logging

When a write-ahead log record covers undo generation, the replay of the write-ahead log record must also regenerate the undo. It seems that we will need to proceed as follows:

(1) Before entering the critical section, allocate and pin the undo buffer or buffers into which the new undo record will be written. This implies that an undo record should not be more than a few pages in length, and that the length should be known before entering the critical section.
(2) Inside the critical section but before calling XLogInsert(), lock the buffers pinned in the previous step, write the undo record into the available buffer space, and mark the relevant buffer or buffers dirty.

(3) Note the undo_pointer at which the record was written in the body of the write-ahead log record, and ensure that sufficient information is included to reinsert the same exact undo bytes at the same byte offset during redo.
(4) Inside the critical section, after calling XLogInsert(), set the LSN of the undo buffers to the value returned by XLogInsert().
(5) After exiting the critical section, release the locks and pins on the undo buffers.

The undo subsystem should provide convenience routines for each step of this process (e.g. PrepareUndoRecordSpace, WriteUndoRecord, SetUndoPageLSNs, UnlockReleaseUndoBuffers).

We should not need any equivalent of XLogRegisterBuffer for undo records provided that the routines that write them don't depend on the validity of the previous buffer contents. That seems like a very good thing, because undo generation could become very costly indeed if it were to increase the volume of full page writes.

The problem of unlogged and temporary tables needs some study, because we can't issue write-ahead log records for some undo records and not others within the same undo log. One idea is to allow each transaction to have two undo logs -- one WAL-logged which is recovered on crash and another that is not WAL-logged which is lost after a crash but can be used for as long as the system remains up.

Reading Undo Records

It will be necessary to provide various APIs to access undo records previous written. For example, we probably need a function to look up an undo_pointer (e.g. UndoRecordFind), a function to look up the last record for a transaction (e.g. UndoRecordFindLast), a function to look up the record immediately preceding the one we've currently got (e.g. UndoRecordFindPrevious), and a function to look up the preceding record for thes same block (e.g. UndoRecordFindPreviousForBlock).

These functions can return a pointer directly into the disk buffer, provided that the undo record doesn't cross page boundaries, or they can return a pointer to a copy of the undo record allocated using palloc(). We will need a way of releasing any resources still held (e.g. UndoRecordRelease, mirroring SysCacheRelease).

Truncation and Removal

In simple cases, undo logs can be removed in their entirety after replay. However, that might not be desirable when a transaction has generated a large quantity of undo, because undo execution might be interrupted. For example, the entire system might go down during undo, or an ERROR or FATAL condition might interrupt (e.g. out of memory, unreadable disk block, pg_terminate_backend on the undo process). On restart, if there is no record of which undo

actions have already been performed, the system would start over at the beginning. Since that would be inefficient for a large undo log, it seems likely that we should at least periodically truncate the undo log while performing undo actions. Undo actions will be replayed in reverse insertion order, so periodically removing the end of the undo log serves as a way to track which actions have already been done while at the same time freeing up disk space.

However, there's another problem: some undo actions might not be idempotent. For example, suppose there is an undo action which marks a particular line pointer unused, reverting the addition of a tuple at that TID during a transaction that went on to abort. Once the undo action is performed, the TID might be reused by some other process; if the undo action is performed again, it would be removing the wrong data. In at least some cases, it seems likely to be necessary to keep careful track of exactly which records have been executed. One way to solve this problem is by truncating the undo log after each such record. However, as the undo log is page-organized, it can't be truncated in the middle of the page although, for example, the records to be truncated away could be zeroed.

However, this approach seems likely to perform poorly. Using ftruncate() to individually remove blocks from the file seems likely to be too expensive if performed for every 8kB block. Instead, it seems like a good idea to store a bit in each undo record indicating whether that record has been executed. As each record is executed, the bit for that record can be set. If the record is idempotent, setting the bit can be skipped; otherwise, the bit must be set atomically with performing the associated action. With this approach, the log can be truncated occasionally (perhaps every 16MB or so) but the status of each record can be individually tracked.

Setting the "executed" bit on an undo record, or truncating the undo log, are operations which must be covered by write-ahead log records.

The Main Undo Loop and Background Workers

There should be a main undo loop that proceeds as follows:

(1) Find the oldest transaction which has an undo log that is not already being handled by some other process and which is either (a) committed and all-visible or (b) aborted.
(2) If the transaction committed, remove the undo log.
(3) If the transaction aborted, perform undo actions in reverse order, starting with the last undo record generated and working towards the beginning of the undo log. If the undo log is large, periodically truncate the log in case undo is interrupted and must be resumed.
(4) Go to step 1.

It should be possible to execute this main undo loop in any PostgreSQL backend in the system, so that (for example) it can be executed in single user mode after each command. However, it should normally be executed by one or more background workers. Nevertheless, the undo loop

itself should be clearly separated from any code specific to background workers, so that there is no confusion between the details specific to background workers and undo per se.

Under ordinary circumstances, it seems sufficient to have a single undo process which is responsible for all undo on a system-wide basis. However, in some cases we may need multiple processes to keep up. Note that, since the undo operations for a single process must be performed in a certain order, undo for a single process can't be done in parallel (although asynchronous I/O to bring in either undo or data blocks about to be needed might be possible). However, undo logs for different transactions can be processed at the same time. Therefore, when many transactions are aborting, multiple undo processes can help to move things along more quickly.

I suggest that every time the main undo loop finishes undo for a transaction and every time it truncates the undo log for a transaction which has generated a large amount of undo, it should consider whether to request a new undo process from the postmaster. Perhaps we could have a GUC min_undo_workers and another max_undo_workers. If the backlog of transactions waiting for undo is large and the number of undo workers is less than the value of max_undo_workers and the amount of time since an undo worker was last started exceeds some threshold (say, 5 seconds), it can request another one from the postmaster. On the other hand, when one or more undo workers are idle for any significant period of time (say, 5 seconds) and the number of undo workers is greater than min_undo_workers, one worker should exit. Care should be taken that no race conditions exist; for example, if min_undo_workers = 1 and there are two workers, both idle, it would be bad if both decided to exit simultaneously.

Undo Logs vs. Other Concurrent Activity

I think that we will need to track in shared memory the set of undo logs that are currently in existence and the status of each undo log: whether or not it is persisted on disk, whether the associated transaction is aborted, all-visible, or neither, and which process if any is currently performing undo for that undo log. The most obvious data structure here is an array.

There will be difficulties if the array fills up. In that case, no new undo logs can enter the system until some undo logs go away. There are several possible approaches to this problem. For example, the process that wants to create an undo log could simply wait until an existing undo log goes away. That's appealing, but if any undo action acquires a relation lock, this approach could result in an undetected deadlock. One possible idea is to make a rule that undo actions are never allowed to take relation locks. However, that has problems, too. Currently, an undo action that wants to modify a buffer for that relation must take a lock on that relation to guard against a concurrent drop of the relation. We could make a rule that dropping a relation has to wait for any undo actions related to that relation to complete, but that's not very desirable, because it means forcing a drop operation to wait for work that won't need to be done after the drop has been completed.

Similar problems can happen in other circumstances. For an undo record that simply cleans up a dead relation file by removing it, no process other than one executing the main undo loop will access those files for any purpose whatsoever, so there is no issue. However, for an undo record that modifies data blocks, there is the possibility - probably relatively easily handled - that the data block might be modified after the undo record is generated and before we attempt to perform the undo action. The undo record must be careful not to get confused. A harder case is when the data block has ceased to exist since the undo record is generated. This can happen when the relation is (a) dropped, (b) rewritten using a new relfilenode, or (c) truncated to a shorter length (and perhaps subsequently re-extended). Either such operations need to wait for pending undo to be applied -- and it's not clear how they would know about pending undo -- or undo routines need to be able to cope with the possibility that those kinds of operations have intervened. The case where a relation has been truncated and subsequently re-extended seems particularly thorny. This is an area that needs further thought.

Undo Construction and Decoding

Insofar as is possible, undo records should share a common format. This makes coding simpler both for core developers and for tool authors. (For example, tools like pg_filedump and pg_xlogdump take advantage of commonalities in block and WAL record formats respectively.)

Each undo log, when persisted on disk, should include include certain metadata identifying the transaction that generated it. Most importantly, it should include the txid of that transaction, likely in the file name (except when multiple undo logs are consolidated into a single physical file). The metadata should also include the OID of the database to which the corresponding transaction pertained. Otherwise, this information would need to be repeated in nearly every record.

The undo record format should include the following:

- Type code. Indicates what kind of undo record this is. Generally, these names should probably be closely related to the names of the corresponding write-ahead log records (perhaps with UNDO in place of XLOG) for ease of understanding.
- A bit indicating whether this record has been executed, always initially 0.
- Length of the current record.
- Length of the previous record, so that we can work backwards through the log when executing undo actions. Required in all cases.
- The tablespace OID and relfilenode to which this undo record pertains. This could be made optional if we have any undo actions that don't pertain to a specific relfilenode, but since it will be very commonly needed, it may make more sense to include it always, unless records that don't need it are thought likely to be frequent.
- The fork number, block number, and item ID to which this record pertains, in each case only if applicable. It might be a good idea to try to omit the main fork number whenever it is zero, which will be common if not universal, or to encode it using only a few bits.

- A pointer to the previous undo record for the same block within this undo log, so that code which only cares about a single block can find the undo records for that block without having to traverse the entire block. This should be omitted when it doesn't apply.
- The tuple data for this undo record, if any.
- Any other payload data for this undo record, in a record-specific format.

There should be a dedicated set of functions which are used to construct undo records for insertion into the transaction's undo log, and another dedicated set of functions which help decode an undo record that has been looked up either for the purpose of performing undo actions or for MVCC purposes.

Extensibility

We could probably relatively easily allow extensions to register undo handlers for otherwise-unused type codes. It's not clear how useful that is, but it's something to consider.