Improving foreign-key locking

Álvaro Herrera

May 18, 2012

1 Historical and current state of affairs

The foreign key implementation has always been a bit problematic from a performance point of view. Underneath, it uses tuple (row) locks to ensure that referenced rows are still in existance at the end of the transaction. Prior to 8.1, it used SELECT FOR UPDATE on those tuples, which was really slow and deadlock-prone. This led to many people removing the foreign key declarations from their database schemas, to improve performance.

In 8.1 we introduced MultiXactIds and SELECT FOR SHARE; this improved concurrency considerably, and many users were very happy. Let's see how this works.

2 A quick tour on tuple locks

Locking tuples is not as easy as locking tables or other objects. The problem is that there might be many more tuples being locked at any one time, so it's not possible to keep the lock objects in shared memory. To work around this limitation, we use a two-level mechanism: the lock information is kept on the physical tuple itself. Whenever an unlocked tuple is to be locked, we set a "row mark" on the on-disk storage area of the tuple. This row mark is, in principle, the Transaction ID (Xid) of the transaction that owns the tuple lock, and is valid only as long as that transaction is in progress. When the transaction is gone, the tuple mark is no longer considered held and some other transaction can acquire it.

So, if a transaction determines that the tuple already has a row mark set (that is, there is another transaction holding the tuple lock), it acquires the tuple's heavyweight lock and then attempts to take a lock on the transaction that owns the row mark. That way, as soon as the lock holder transaction finishes, the lock waiter is awaken and can proceed with obtaining the row mark. Now, if the heavyweight lock on the tuple is also taken, it means that the tuple is locked and there is some other transaction waiting to acquire the row mark, so we also sleep then; we will be awakened when the waiting transaction releases the tuple's heavyweight lock, most likely only to continue sleeping until that transaction also releases the row mark when it finishes.

When a tuple is locked by more than one transaction (in share mode, obviously) there isn't enough space in the tuple header to store the Xid of each and every locking transaction. Instead, we store a number of the same width as an Xid, and set a flag on the row header that this number is not a simple Xid. We call this number a MultiXactId. Elsewhere (`pg_multixact`), the system stores a list of Xids that each MultiXactId is associated with — its members.

When a tuple is locked by multiple transactions, with MultiXactIds as currently implemented, there is no practical difference from what's described above. The row mark has an infomask bit (more on that below) that indicates that the locker is actually a MultiXactId; a transaction that needs to wait until the locker is gone has to grab the list of members of the MultiXactId, and then sleep on them one by one. However, if the lock mode it wants to acquire doesn't conflict with the existing locks (a situation that can only arise when the locks held are FOR SHARE and the new transaction also wants a FOR SHARE lock), then it grabs the list of members, adds itself to it, and create a new MultiXactId with the list so created, then marks the row with it[1].

2.1 Infomask bits

So where is this "row mark" thing stored? Each tuple has a small area that is used to store flag bits about tuple state. Row marks involve four of these bits: `HEAP_XMAX_INVALID`, `HEAP_XMAX_EXCL_LOCK`, `HEAP_XMAX_SHARE_LOCK`, and `HEAP_XMAX_IS_MULTI`.

In the current code, there are five interesting states to distinguish:

1. Tuple is not deleted, updated or locked
2. Tuple is updated or deleted
3. Tuple is locked in exclusive mode
4. Tuple is locked in shared mode by a single transaction
5. Tuple is locked in shared mode by multiple transactions

State	INVALID	EXCL_LOCK	SHARE_LOCK	IS_MULTI
untouched	X			
deleted or updated	(no bits set)			
exclusive locked		X		
share-locked by one			X	
share-locked by many			X	X

The Xid of the locking transaction (or the MultiXactId, if that's the case) is stored in the `Xmax` field of the row header.

[1] It follows that a transaction that sleeps on multis have to recheck the multi after they wake up, just in case someone added itself to the list in the meantime.

A great virtue of this scheme is the simplicity: the interested transaction needs only to check whether one of the HEAP_XMAX_EXCL_LOCK or HEAP_XMAX_SHARE_LOCK bits is set to determine whether the tuple is updated or just locked, which is crucial for visibility testing.

3 Implementing improvements

However, concurrency problems with foreign keys remain, and some users still remove them. These performance problems were known and understood and suffered by many. Deadlocks, however, weren't so commonly reported. However, they still showed up in some situations; they are, of course, much worse than concurrency loss because they cause transactions to abort that have to be retried.

Joel Jacobson from Glue Finance reported one such problem along with many details. His report was thoroughly analyzed and discussed on the pgsql-hackers list, during which discussion Simon Riggs came up with a proposal to rework the locking level that foreign keys acquire; this was supposed to fix the problem for good.

Command Prompt accepted the challenge to implement such a thing. We proposed a first patch[2], which was of reasonably limited scope. This patch wasn't modifying MultiXactIds at all, but instead it introduced a new locking mode for tuples (which was dubbed KEY LOCK), using a separate infomask bit, and changed the behavior of UPDATE when it detected such a lock on the tuple to be updated.

Noah Misch reviewed the patch in minute detail, and his most important conclusion was that while our patch did fix some of the deadlock scenarios, it didn't fully solve the deadlock problem – in particular, it didn't fix the problem reported by Joel. He pointed out that while the patch allowed an update to proceed on a tuple that was locked, it did not allow a lock to proceed on a tuple that was updated, which in hindsight was just plain weird.

Noah came up with the idea of differentiating lock modes at the MultiXact level. This would cause the lock conflict table to make more sense, and while at it, it would allow locks not to conflict with updates on tuples. We studied this idea and decided to implement it.

4 What the new multixactids look like

In order to support a more sophisticated lock conflict table, we need to extend MultiXactIds to store the lock type that each member transaction holds. That way, a future locker can inspect the list of members and determine whether it conflicts or not.

There are six interesting locking states to differentiate:

[2] http://archives.postgresql.org/message-id/1294953201-sup-2099@alvh.no-ip.org

- FOR KEY SHARE, used by foreign keys
- FOR SHARE, a legacy mode implementing normal share-lock behavior
- FOR UPDATE, an SQL-conformant lock mode
- FOR KEY UPDATE, stronger than FOR UPDATE
- UPDATE, acquired by updates that do not change the values of the columns of the tuple's key
- KEY UPDATE, acquired by updates that do change the values of key columns, and also by row deletion.

The conflict table is:

	FKS	KS	FU	FKU	U	KU
FOR KEY SHARE				X		X
FOR SHARE			X	X	X	X
FOR UPDATE		X	X	X	X	X
FOR KEY UPDATE	X	X	X	X	X	X
UPDATE		X	X	X	X	X
KEY UPDATE	X	X	X	X	X	X

Note that as far as the conflict table goes, FOR UPDATE and UPDATE behave identically, as do FOR KEY UPDATE and KEY UPDATE. The difference between the two modes in each pair is that the FOR variant is only a lock, whereas the other one is known to have updated the tuple (in other words, there is a newer version of the tuple elsewhere). We need to distinguish those two cases for performance reasons that will become clear later on.

4.1 The updating protocol

When you want to update a tuple:

- if the tuple is untouched, update normally
- if the tuple is locked and your lock doesn't conflict, grab the lockers list, add yourself to it, and set it as the lockers of the old version of the tuple. The new tuple must be marked with the old lockers list. If you notice that the lockers list is empty, proceed as above.
- if the tuple is locked and your lock conflicts, grab the lockers list and sleep on it. When you are awaken, proceed as above.
- if the tuple is updated, sleep normally until the updating transaction finishes, then
  - if if commits, fail normally (serializable) or grab updated version and restart (read committed)

– if it aborts, continue as above.

Note the main thing of interest here is to be able to quickly figure out if a tuple is locked or not, what's the strongest lock held, and whether there is an update or not.

4.2 The locking protocol

When you want to lock a tuple:

- if the tuple is untouched, just grab the lock.
- if the tuple is locked, and your lock doesn't conflict, grab the lockers list, add yourself to it, and set it as new locker.
- if the tuple is locked and your lock conflicts, grab the lockers list and sleep on it. When you are awaken, proceed as above.
- if the tuple is updated and your lock doesn't conflict, grab the lockers list, add yourself to it, set as new locker, then **follow the update chain** and lock the updated versions too.
- if the table is updated and your lock conflicts, grab the lockers list and sleep on it. When you are awaken, proceed as above.

Here we're also interested in the maximum locking strength, and whether there is an update or not. Note that part of the "locking strength" thing that we need to figure out is whether the tuple's key has been modified in an update: because if it hasn't, then a FOR KEY SHARE lock doesn't conflict and doesn't have to sleep; but if it has then it has to wait until the update is done.

4.3 The new infomask bit definition

In order to support all of this, and having it perform decently, some additions and changes to infomask bits were made. We added HEAP_XMAX_KEYSHR_LOCK to have a fast path to figure out conflicts in the very common case of foreign key checking. We also added HEAP_UPDATE_KEY_REVOKED which is set whenever an update (and a delete) "revoke" the key of a tuple, that is, whether it either changes it or deletes the tuple completely. Finally, we changed HEAP_XMAX_SHARE_LOCK to HEAP_XMAX_LOCK_ONLY; this is set whenever there is a lock on the tuple but it hasn't been updated.

Note that a FOR SHARE lock no longer has a way to distinguish itself from other locks using only the infomask bits, so we force those locks to always use MultiXactIds. This is a bit of a performance regression, but since we expect FOR SHARE locks to be seldom used, we don't feel this is a serious problem.

1. Tuple is not deleted, updated or locked
2. Tuple is deleted, or updated with key columns changed (the key is revoked)

3. Tuple is updated, but no key columns are changed
4. Tuple is locked in key-exclusive mode
5. Tuple is locked in exclusive mode
6. Tuple is locked in shared mode
7. Tuple is locked in key-shared mode

State	INVALID	EXCL_LOCK	KEYSHR_LOCK	LOCK_ONLY	KEY_REVOKED
untouched	X				
deleted or updated					X
key-exclusive locked		X		X	X
exclusive locked		X		X	
share-locked				X	
key-share-locked			X	X	

Note that there are several cases where MultiXactIds can be involved here where they weren't previously. In particular, we can have them on updates or deletes. This is necessary so that a transaction can lock a tuple when it's being updated by another one. However this is also useful for other reasons: consider a transaction that locks a tuple, then a subtransaction deletes it and aborts. The current code just "forgets" the lock, and so after the subtransaction abort the tuple is no longer locked. This is considered acceptable in the current code, though the documentation contains a warning about it, and it is certainly a violation of the so-called "principle of least surprise". Fixing this problem is a nice side-effect of this patch.

5 Implementation notes

5.1 WAL

A new WAL message had to be added, so that whenever an updated tuple is locked, the updated copy is also locked. Also, the existing UPDATE, DELETE and LOCK TUPLE records had to be expanded to include the possibility of a MultiXactId being involved, and what the new infomask bits are.

5.2 Visibility rules (tqual.c)

Visibility rules are the ones that take a tuple header and a snapshot, and determine whether the tuple is visible to that snapshot. They are really tricky to get right: they have to consider the multiple different cases of infomask bits being set or not, the values of the Xmin and Xmax fields, and data from other parts of the system such as pg_clog and pg_multixact lookups.

The bits that we're interested in changing for this patch are basically whether a tuple is visible or not depending on updated/locked conditions. Previously,

figuring this out was pretty easy: if the "locked" bits were set, then it was not an update; and if it was locked, then it was visible. If it's an update, then we grab the Xmax value and test it in pg_clog for visibility.

In the new code, a tuple might be locked and updated at the same time, so the appropriate bit to check now is HEAP_XMAX_LOCK_ONLY. But the real complexity is that when HEAP_XMAX_IS_MULTI is set, and LOCK_ONLY is not set, we need to resolve the MultiXactId to its member transactions and scan them to find what's the Xid of the update transaction – the "effective Xmax", so to speak. That Xmax is the value we need to test in pg_clog. This means that visibility testing, which previously only needed to access the infomask bits and occasionally pg_clog, might now also need to access pg_multixact.

5.3 pg_upgrade

Changes in MultiXactId on-disk storage mean that pg_upgrade support is needed. There are two parts to this. The thing to keep in mind is that pg_upgrade currently does not copy pg_multixact data files, which is okay for the current code, because the data is not useful after a database restart. However, with the patched code, they are necessary because some of the MultiXactIds in there might contain updates and so are essential for visibility testing.

The first issue is upgrading from a version that doesn't have this patch to a version that has; this needs some way to "protect" tuples that are marked as locked in the old code, so that when the visibility code tests them, the result is the same as if it was running in the old version. The solution to this problem is to have pg_upgrade set an "epoch" variable in the MultiXact subsystem to the latest MultiXactId value assigned in the old install, so that any value tested prior to that (which by definition is only going to come from the old installation) is reported as "only locked", which is consistent with the fact that FOR SHARE locking is all that the old installation had.

The second problem is upgrading for a version that has the patch, to a newer version that also has it. This one is simple – just copy the pg_multixact files from the old cluster into the new, just like pg_upgrade already handles pg_clog files.

5.4 EvalPlanQual

EvalPlanQual (EPQ) is a complex mechanism used mainly by the READ COMMITTED rules to obtain and verify qualifications for tuples that have been updated (it's also used for things like triggers and others, but I'm going to ignore these).

The relationship that EPQ has to the patch at had, is that it also walks an update chain to do something about it – exactly what our lock-updated-tuple code does. Since EPQ walks the chain and ends with locking the final version, and lock-updated-tuple also walks the chain locking each version, what we end up with is that concurrent updates to the same tuples might end up dying with deadlocks.

The way we ended up working around this problem, was to shut down EPQ from recursing into the updated tuple in certain conditions. We're not really certain that this fix is correct, or whether there are ways to make it fail.