To Lock and not to Block

Improving foreign key concurrency

Álvaro Herrera
Command Prompt Inc.

PGCon 2012
Ottawa, ON, CA

Introduction

I’m working on improving foreign key concurrency.

Introduction

I'm working on improving foreign key concurrency.

What does that mean, exactly?

Some history

- Foreign keys were introduced in 1999
- by Jan Wieck
- released with version 7.0
- using triggers and row-level locks

Some history

- Foreign keys were introduced in 1999
- by Jan Wieck
- released with version 7.0
- using triggers and row-level locks
- Back then, FOR UPDATE was the only row locking method we had

Some history

- Foreign keys were introduced in 1999
- by Jan Wieck
- released with version 7.0
- using triggers and row-level locks
- Back then, FOR UPDATE was the only row locking method we had
- FOR UPDATE is exclusive row locking
- you get the lock, everybody else waits behind you
- not very concurrent

Some history

- Foreign keys were introduced in 1999
- by Jan Wieck
- released with version 7.0
- using triggers and row-level locks
- Back then, FOR UPDATE was the only row locking method we had
- FOR UPDATE is exclusive row locking
- you get the lock, everybody else waits behind you
- not very concurrent
- it must have been great at the time

Why tuple locking

What does tuple locking have to do with anything?

- when you create a new reference, ensure your referenced tuple doesn't go away
- this assurance has to persist till the end of your transaction

Why tuple locking

What does tuple locking have to do with anything?

- when you create a new reference, ensure your referenced tuple doesn't go away
- this assurance has to persist till the end of your transaction
- until then, nobody can see your new tuple ...
- ... so existance of the referenced tuple is your problem

Why tuple locking

What does tuple locking have to do with anything?

- when you create a new reference, ensure your referenced tuple doesn't go away
- this assurance has to persist till the end of your transaction
- until then, nobody can see your new tuple ...
- ... so existance of the referenced tuple is your problem
- after that, your tuple will be visible ...
- ... so existance of your referenced tuple is the remover's problem

Foreign keys with FOR UPDATE

- FOR UPDATE grabs a tuple exclusive lock
- Nobody else can even *reference* the locked tuple until you finish
- Highly referred tables become a heavy point of contention

How does tuple locking work?

- Can’t keep tuple locks in regular lock table
- the reason: there might be too many of them
- workaround: store lock info in the tuple itself

How does tuple locking work?

- Can’t keep tuple locks in regular lock table
- the reason: there might be too many of them
- workaround: store lock info in the tuple itself
- Store TransactionId (Xid) of locking transaction in the tuple’s Xmax field

Tuple locking protocol

- obtain the tuple's Xmax value
  - if it's Invalid, there is no lock
  - if it's valid but the transaction is not running, there is no lock
- if there is no lock, grab it:
  - set Xmax to the locking transaction
  - set the HEAP_XMAX_EXCL_LOCK infomask bit

Tuple locking protocol

- obtain the tuple's Xmax value
  - if it's Invalid, there is no lock
  - if it's valid but the transaction is not running, there is no lock
- if there is no lock, grab it:
  - set Xmax to the locking transaction
  - set the HEAP_XMAX_EXCL_LOCK infomask bit
- if there's a lock, sleep on the value in Xmax
- when you are awakened, the locker is gone
- restart at the top

Introducing FOR SHARE

- version 8.1 saw the birth of shared row locking
- non-standard extension: SELECT FOR SHARE
- much better concurrency for FKs
- problem is: where to store locking info?
- certainly not the regular lock table
- certainly not the Xmax itself

FOR SHARE mechanism

- MultiXactId
- an array of Xids associated with an uint4 key
- instead of storing an Xid in Xmax, we store a MultiXactId
- each tuple stores whether its Xmax is a Multi or not
- infomask bits:
  - HEAP_XMAX_INVALID
  - HEAP_XMAX_EXCL_LOCK
  - HEAP_XMAX_SHARE_LOCK
  - HEAP_XMAX_IS_MULTI

FOR SHARE / possible cases

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

FOR SHARE / infomask bit states

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

Locking protocol

- is Xmax free?
  - just grab it
  - done
- if Xmax is taken, does it conflict with you?
- Yes: sleep on it. When you're awaken, start again.
- if not, note the locker, and
  - if a single xact, create a multixact with the two, set it as the xmax
  - if a multixact, expand it by adding yourself, set it as the xmax

FOR SHARE is great

- Shared locking improves things a lot ...

FOR SHARE is ~~great~~ somewhat useful

- Shared locking improves things a lot ...
- ... but it still has problems

FOR SHARE is ~~great~~ somewhat useful

- Shared locking improves things a lot ...
- ... but it still has problems
- Consider:

```
CREATE TABLE pktable (pk INT PRIMARY KEY, somecol INT);
CREATE TABLE fktable (fk INT REFERENCES pktable);
INSERT INTO pktable VALUES (1);

BEGIN;
INSERT INTO fktable VALUES (1);
```

FOR SHARE is ~~great~~ somewhat useful

- Shared locking improves things a lot ...
- ... but it still has problems
- Consider:

```
CREATE TABLE pktable (pk INT PRIMARY KEY, somecol INT);
CREATE TABLE fktable (fk INT REFERENCES pktable);
INSERT INTO pktable VALUES (1);

BEGIN;
INSERT INTO fktable VALUES (1);

-- now on another session:
UPDATE pktable SET somecol=somecol+1 WHERE pk=1;
-- blocks
```

Deadlocks!

You can even get deadlocks. Example:

```
CREATE TABLE A (
   AID serial not null PRIMARY KEY,
   Col1 integer
);

CREATE TABLE B (
   BID serial not null PRIMARY KEY,
   AID integer not null REFERENCES A,
   Col2 integer
);

INSERT INTO A (AID) VALUES (1);
INSERT INTO B (BID, AID) VALUES (2, 1);
```

Deadlocks! (2)

```
Process 1:
                                Process 2:
BEGIN;
                                BEGIN;
UPDATE A SET Col1 = 1
WHERE AID = 1;
                                UPDATE B SET Col2 = 1
                                WHERE BID = 2;
UPDATE B SET Col2 = 1
WHERE BID = 2;
-- blocks
                                UPDATE B SET Col2 = 1
                                WHERE BID = 2;
                                -- deadlock!
```

First patch

- First attempt at fixing the problem
- reasonably simple patch — only 50kb
- Based on my misunderstanding of a proposal by Simon Riggs
- Theory of operation:
  - you are able to update a tuple that's locked
  - but you have to copy locking information forward

First patch — torn apart by Noah Misch

- Friendly neighborhood reviewer Noah Misch dissected the patch
- On functional review, he found it insufficient
- problem: it doesn’t let you lock a tuple that’s updated
- deadlocks persisted

First patch — torn apart by Noah Misch

- Friendly neighborhood reviewer Noah Misch dissected the patch
- On functional review, he found it insufficient
- problem: it doesn’t let you lock a tuple that’s updated
- deadlocks persisted
- Noah provided two genius ideas:
  - One: split lock conflict table
  - Two: store lock strength in MultiXacts

New lock modes

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

New Lock Conflict Table

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

Update 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.

Tuple lock 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.

Following the update chain

- When locking a row, it's important to also lock future versions
- this situation arises when the locker transaction has a snapshot older than the update
- Failing to lock the updated row would allow a future transaction to delete the updated row when the locking transaction is still running
- This leads to violated constraints

Following the update chain

- When locking a row, it's important to also lock future versions
- this situation arises when the locker transaction has a snapshot older than the update
- Failing to lock the updated row would allow a future transaction to delete the updated row when the locking transaction is still running
- This leads to violated constraints
- It's a pain to implement
- Needs a separate WAL record
- EvalPlanQual also walks update chains and also locks rows
- having both causes hard-to-reproduce spurious deadlocks

Implementation

- Actually implementing this is not simple
- The patch took much review and many revisions
- Latest one is 400kB

Implementation

- Actually implementing this is not simple
- The patch took much review and many revisions
- Latest one is 400kB

95 files changed, 5303 insertions(+), 1377 deletions(-)

Implementation

- Actually implementing this is not simple
- The patch took much review and many revisions
- Latest one is 400kB

95 files changed, 5303 insertions(+), 1377 deletions(-)

- **there are still some bugs**
- ... but it's getting close!

Challenges

Some implementation notes about things that bit us while working on this patch.

Infomask bits

State	EXCL LOCK	KEYSHR LOCK	LOCK ONLY	KEY REVOKED	IS_MULTI
deleted or updated				X	maybe
updated, key untouched					maybe
key-exclusive locked	X		X	X	maybe
exclusive locked	X		X		maybe
share-locked			X		X
key-share-locked		X	X		maybe

Getting the correct representation required several iterations.
Some hackers do not seem happy with some of the names.
Improvement suggestions are accepted.

WAL

- Not all that interesting
- Added more detailed WAL logging
- probably not really necessary
- can be trimmed later (hopefully)

- UPDATE
- DELETE
- SELECT FOR [KEY] SHARE | UPDATE

pg_upgrade

Two parts to this:

1. Upgrading from current version into patched version
   - Naïve: convert old files by tweaking the contents.
     - too messy
   - Medium: Set epoch to last used value+1. Values queried before that always return empty set

pg_upgrade

Two parts to this:

1. Upgrading from current version into patched version
   - Naïve: convert old files by tweaking the contents.
     - too messy
   - Medium: Set epoch to last used value+1. Values queried before that always return empty set
2. migrating from a patched version to another patched version.
   - simply copy the files, just like we handle pg_clog

Visibility Rules

- tqual.c has to change to adapt to the new reality
- Some cases which returned false (or something functionally equivalent) now allow caller to continue
- Requires obtaining the MultiXactId member list from pg_multixact
- Only happens in cases that would block.

Visibility Rules

- tqual.c has to change to adapt to the new reality
- Some cases which returned false (or something functionally equivalent) now allow caller to continue
- Requires obtaining the MultiXactId member list from pg_multixact
- Only happens in cases that would block.
- (Not really).

Visibility Rules

- tqual.c has to change to adapt to the new reality
- Some cases which returned false (or something functionally equivalent) now allow caller to continue
- Requires obtaining the MultiXactId member list from pg_multixact
- Only happens in cases that would block.
- (Not really).
  - some cases are now slower.
  - Needs optimization work. Maybe hint bit (XMAX_COMMITTED) rethinking
  - Affects other areas as well (SSI, vacuuming)

EvalPlanQual

- This part needs more thought
- The problem: EPQ does its own update chain walking
- its locking seems to conflict with what heap_lock_tuple is doing
- current fix is just to shut down its recursion in certain places
  - (not really sure this is correct/sufficient)

Performance improvements?

- pg_bench shows a 9 % performance *regression* with no FKs
  - this is the main reason the patch didn't make it to 9.2
  - I'll research this more to make a final submission

Performance improvements?

- pg_bench shows a 9 % performance *regression* with no FKs
  - this is the main reason the patch didn't make it to 9.2
  - I'll research this more to make a final submission
- No measurements have been made on real-world cases being fixed ...

Performance improvements?

- pg_bench shows a 9 % performance *regression* with no FKs
  - this is the main reason the patch didn't make it to 9.2
  - I'll research this more to make a final submission
- No measurements have been made on real-world cases being fixed ...
- ... but not having to retry deadlocked transactions is a huge gain

Performance improvements?

- pg_bench shows a 9 % performance *regression* with no FKs
  - this is the main reason the patch didn't make it to 9.2
  - I'll research this more to make a final submission
- No measurements have been made on real-world cases being fixed ...
- ... but not having to retry deadlocked transactions is a huge gain
- Time gained by not having to wait when the current code blocks, is largely application-dependant, but it might well be huge too

Questions?

Thanks for listening!

`http://github.com/alvherre/postgres/tree/fklocks`

Questions?

Thanks for listening!

`http://github.com/alvherre/postgres/tree/fklocks`

Any questions?