Why UPSERT is weird

Peter Geoghegan

PgCon 2014

Thursday, May 22

About me

- PostgreSQL major contributor
- Often work on performance stuff
- Work for Heroku – build internal infrastructure for Postgres database-as-a-service

What is this talk about?

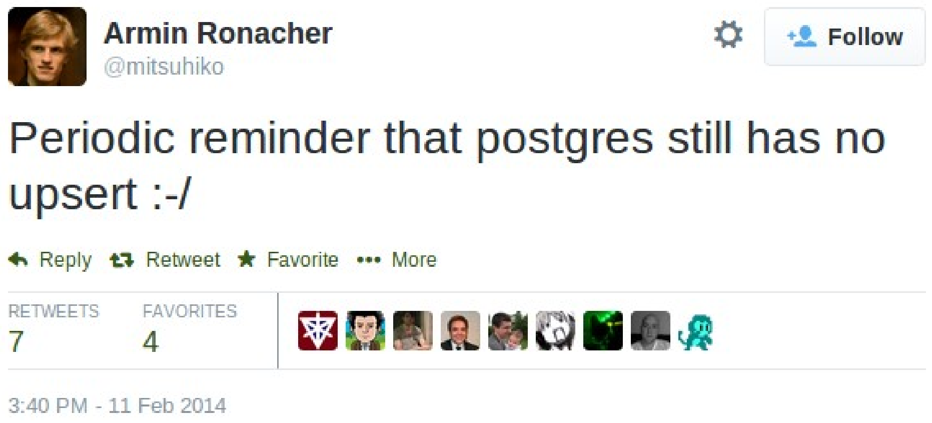
- INSERT or UPDATE – one or the other. Atomic. Popularly known as “UPSERT”.
- Duplicate defined in terms of (would-be) unique index violations
- Strategic implications
- Implementation considerations

What is this talk *not* about?

- Some will be aware of the fact that Heikki Linnakangas wanted me to go a different way with the implementation
- I will refer to my own implementation, but the concerns under discussion today are *identical* for both
- Heikki sketched a design with *identical* user-visible semantics, which are the real story here. This wasn't quite the case for a while.
- I believe that nothing I'll present as anything other than an opinion is actually disputed

Satisfying everyone

- Hackers recognize that this is an important project, but naturally strongly prefer something that comports with existing code and conceptual precepts
- Users want something broadly useful. There should be minimal gotchas.
- Lots of misinformation on the internet about how to do UPSERT manually. Very confusing.
- As an implementer, I want to keep everyone happy

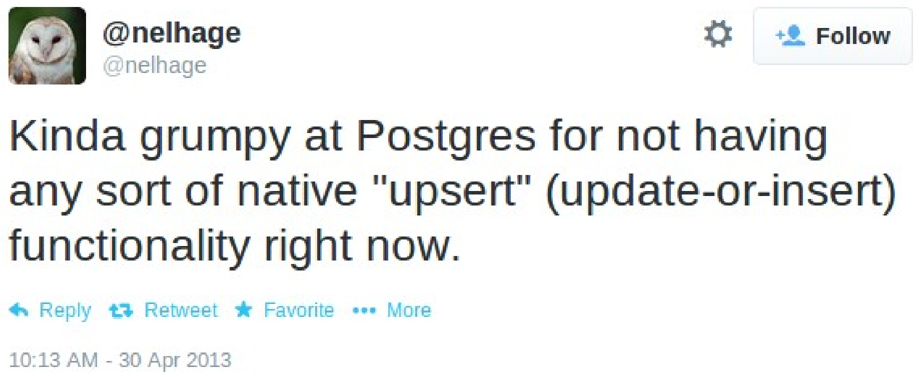


Armin Ronacher
@mitsuhiko

Periodic reminder that postgres still has no upsert :-/

RETWEETS 7 FAVORITES 4

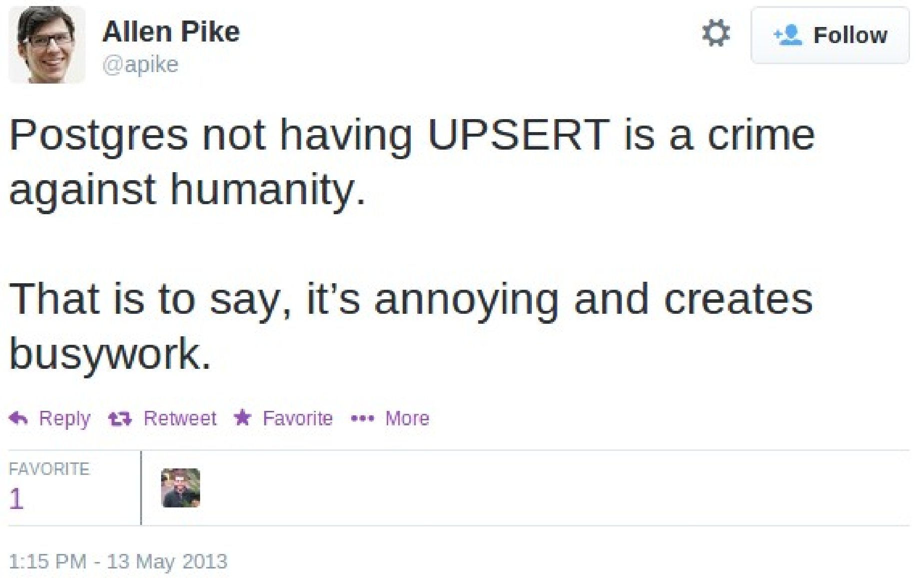
3:40 PM - 11 Feb 2014



@nelhage
@nelhage

Kinda grumpy at Postgres for not having any sort of native "upsert" (update-or-insert) functionality right now.

10:13 AM - 30 Apr 2013



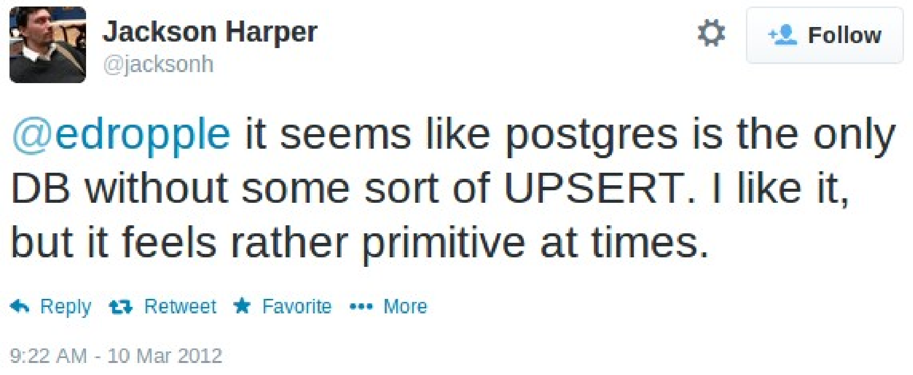
Allen Pike
@apike

Postgres not having UPSERT is a crime against humanity.

That is to say, it’s annoying and creates busywork.

FAVORITE 1

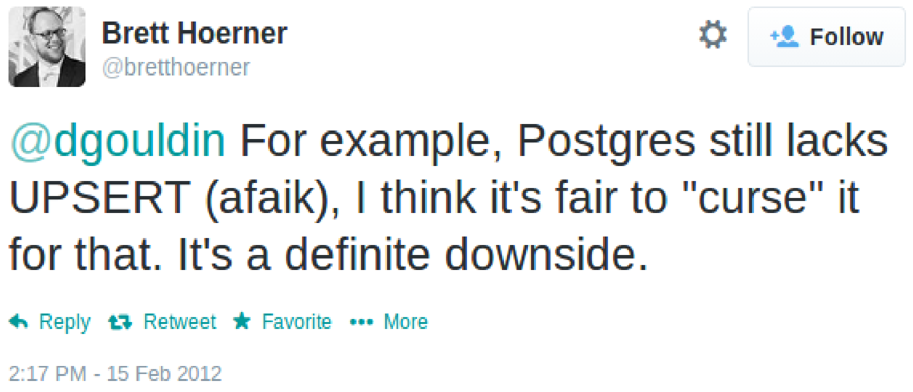
1:15 PM - 13 May 2013



Jackson Harper
@jacksonh

@edropple it seems like postgres is the only DB without some sort of UPSERT. I like it, but it feels rather primitive at times.

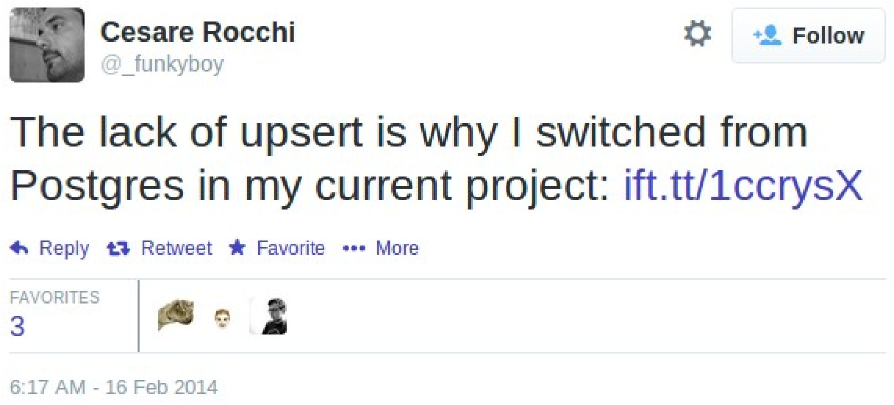
9:22 AM - 10 Mar 2012



Brett Hoerner
@bretthoerner

@dgouldin For example, Postgres still lacks UPSERT (afaik), I think it's fair to "curse" it for that. It's a definite downside.

2:17 PM - 15 Feb 2012



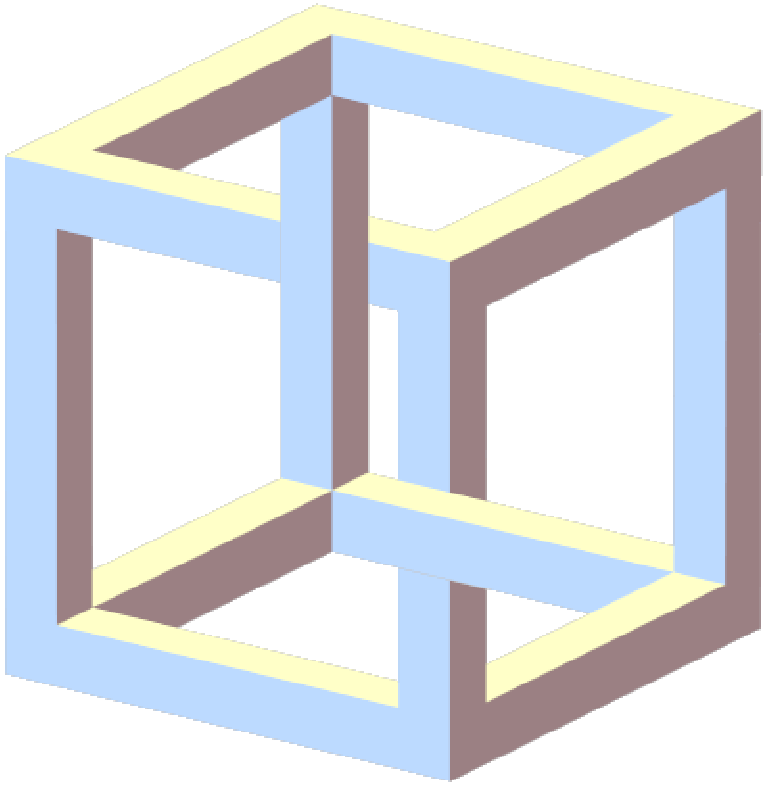
Cesare Rocchi
@_funkyboy

The lack of upsert is why I switched from Postgres in my current project: ift.tt/1ccrysX

FAVORITES 3

6:17 AM - 16 Feb 2014

UPSERT in theory



Goals for UPSERT in Postgres

- *“Fundamental UPSERT property”*. At `READ COMMITTED` isolation level, you should *always* get an atomic insert or update.
  - No *unprincipled* deadlocking
  - No spurious unique constraint violations
- Although there will be a single, implicit “merge on” unique indexed column or columns, there should be no need to *explicitly* specify *which* unique index we mean. That would be a really ugly requirement for a DML statement.
- Acceptable performance. Quickly burning through XIDs (from aborting subtransactions) intractable for important multi-master replication (BDR) insert conflict handling use-cases.

How unique indexes work in Postgres

- Within `ExecInsert()` executor code, we handling the insertion of a single *table slot*
- A single heap tuple is physically inserted. We then have a physical tuple identifier (TID) for it.
- Insert index tuples for each index on the table, using the heap TID. Some may be unique indexes. In practice, only the `btree` AM is cataloged as supporting them (`amcanunique = 't'`).
- B-Tree code aborts transaction if there is a violation
- There may be other index tuples inserted when that happens. Everything becomes bloat.
- It's the AM's problem as to how duplicates are detected

How B-Tree code handles duplicates

- With trivial exception, Postgres B-Trees don't have any visibility information (i.e. no metadata with which to directly figure out if `IndexTuple` tuples are visible to a given transaction's snapshot under MVCC rules)
- MVCC rules aren't quite what we care about here anyway; *our* snapshot may not *yet* see a would-be duplicate, for example, but of course we still have to worry about them
- Code uses a `DirtySnapshot` (not our existing MVCC snapshot from executor state). These are used in just a few places in Postgres. Allows code to consider effects of current transaction, and in-progress transactions.
- In essence, code looks for conclusively-visible duplicate. May have to wait indefinitely pending outcome of *other* transaction (if it INSERTs, UPDATEs, or DELETEs would-be duplicate of interest).
- If there is conclusively such a duplicate after that wait, raises `ERRCODE_UNIQUE_VIOLATION`. Otherwise, proceeds with physical insertion of IndexTuple.

B-Tree structure

- We use buffer locks (which serve as page locks) to protect the physical structure of the B-Tree. Preserves invariants. This would probably be equally necessary within a filesystem that happens to use B-Trees (e.g. btrfs).
- We share lock the root page, inner pages and one or more leaf pages as part of an index scan. Usually only need to lock one at a time during tree descent for insert.
- Unique indexes *kind of* work by treating uniqueness as a part of the B-Tree structure that must similarly be protected as an invariant.
- Of course, it's a lot messier than that. We often have multiple *versions* of tuples in the same B-Tree, corresponding to each heap version, and they don't duplicate each other according to the semantics we find useful here, even though they're duplicates in the strict physical sense (i.e. a simple index ScanKey would indicate they're equal).
- Since we have to go to the heap (table) for visibility information to sort the mess out, we have to exclusive lock the buffer/page that is the *first* leaf page a would-be duplicate *could* be on.

B-Tree structure (cont. 1)

- We search the heap for duplicates on the first leaf page (and maybe subsequent leaf pages) with that original exclusive lock on the first leaf page held throughout.
- Exclusive buffer lock needed because we cannot reasonably later escalate from shared, and we usually need to actually perform an insertion into the leaf page without releasing that lock. Someone else might “get in ahead of us” if we released a shared to get an exclusive. They might then insert the integer value 5 when that was what we'd intended to insert, even though we've already concluded that it's okay to proceed. Index becomes corrupt.
- We must, in a limited sense, lock, say, the integer 5 – a value - *in the abstract.* We cannot lock some *existing* object, *because there is none*. That's the whole point.

B-Tree structure (cont. 2)

- You might say that through this buffer locking mechanism, Postgres already has a very limited form of *value locking*, which is sometimes known as *range locking* in 2PL systems that need it for all kinds of things. Some systems' `SERIALIZABLE` isolation levels depend on this, and even depend on appropriate indexes being available.
- Yes, with Postgres inserting a row *will* block would-be duplicate inserts, but the row lock here is still quite distinct from a value lock. For users of other systems, this salient distinction will seem natural. For Postgres hackers, perhaps less so.

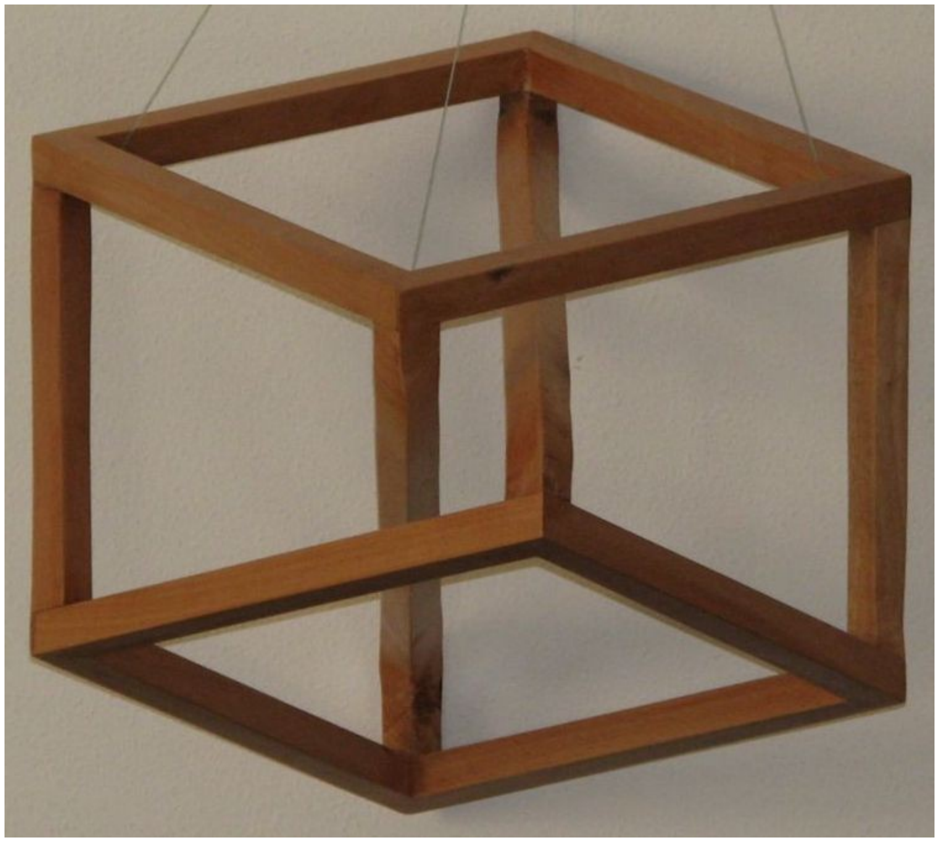
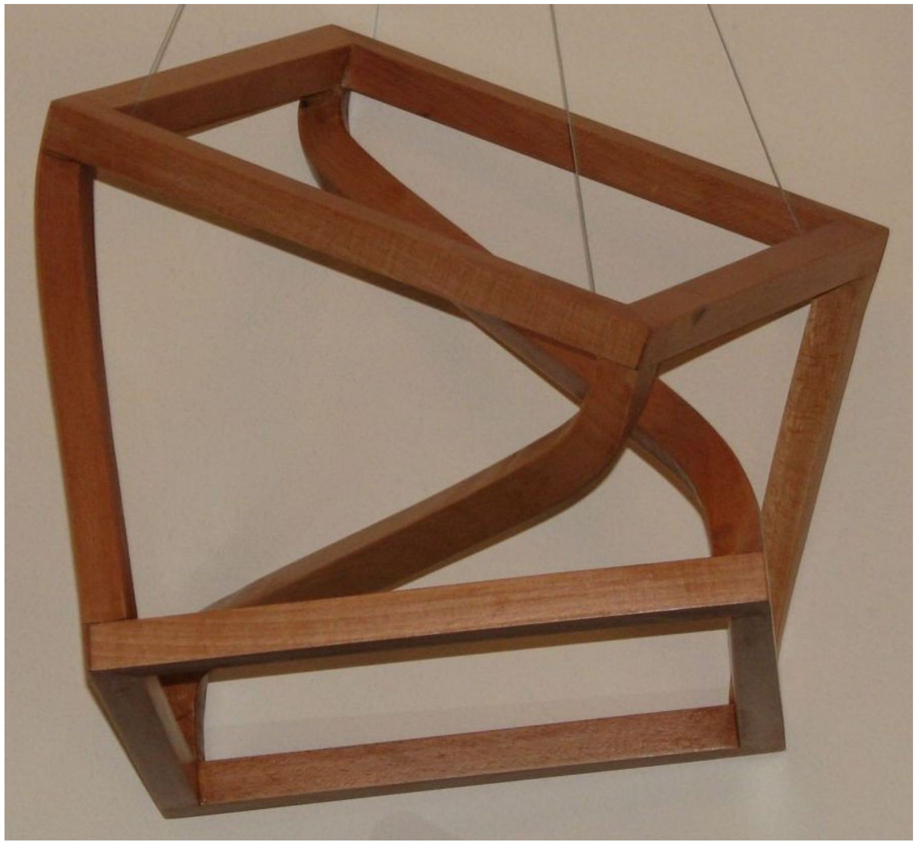
B-Tree value locking

- We can't hold that exclusive buffer lock all day long, or even for more than an instant.
- So when we see that there is an inconclusively-committed conflict tuple, but need to wait pending the end of some other transaction, we release the exclusive lock, having obtained an XID to wait on from the heap (table) while page was exclusive locked.
- Sleep, and when woken by other transaction's commit/abort, start from scratch (just for that unique index). Repeat until a *conclusive* yes/no is obtained. When we physically insert into the index, it may then become some other session's problem to worry about that value and our transaction.
- So today we kind of link the low-level, high performance buffer locks to longer term lock-manager managed XID locks, *without* row locking style lock arbitration.
- By which I mean: The first session to wait on other transaction *may not be the first to get a second try*.
- Works fine for unique index enforcement, but that's about it

A naive approach to UPSERT

- Tie Index value locking to conventional row locking
- Don't insert heap tuple first; go and “value lock” all unique indexes
- If there is consensus to proceed, only then insert heap tuple, and lastly insert index tuple(s), finally releasing value locks
- Otherwise, go lock/update row (I lock the conflict row in my patch, but that really isn't too distinct from actually updating it for our purposes). Then release the value locks when you're done.
- Since an escalation from value locking to row locking occurs, that must mean everything works out...right?

UPSERT in practice

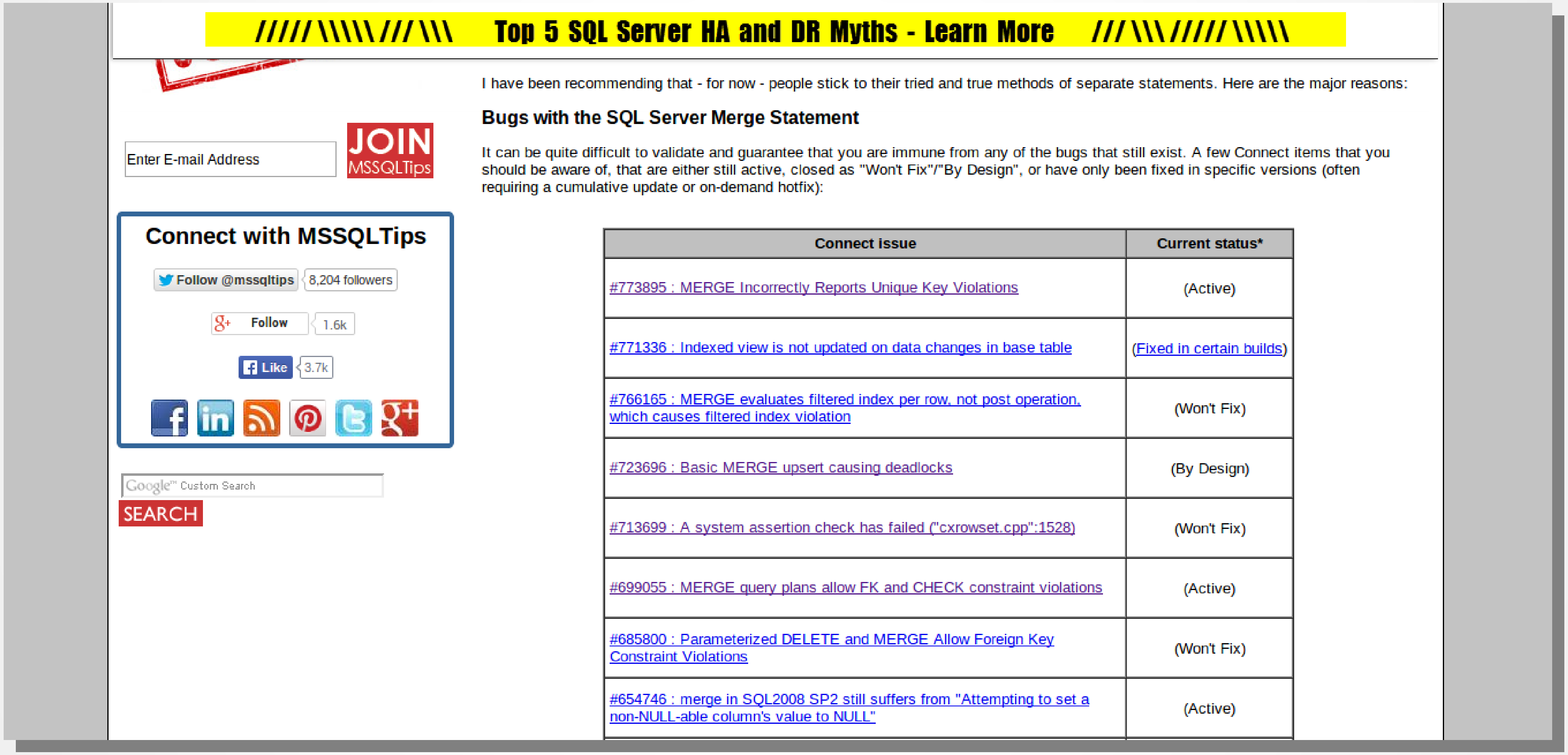



Problems with overlapping “value” and row locks

- If user locks two things at once, and is not careful about the ordering, there may be deadlocks. Value locks (of whatever form) are one thing, and row locks are another thing.
- Tends to consistently deadlock in a way that isn't defensible to Postgres users
- We still have to worry about regular inserters, and they too are only going to be prevented from *finishing* insertion in respect of conflicting table slot (i.e. `ExecInsert()` call only blocks when we finally go to insert into indexes).
- Determining if predicates are compatible ahead of time, which is what it would take to make this actually work (short of a full table `ExclusiveLock`), is known to be NP-hard. No actual implementation can do this.

Conclusion: **You cannot tie value locking to row locking (i.e. have the two overlap when upserting a slot, managing upserters in a strict queue) if you want to consistently avoid throwing *some* error**

Other systems, SQL MERGE



//////\\\\\///\\\ Top 5 SQL Server HA and DR Myths - Learn More ///\\\//////\\\\\

Enter E-mail Address

JOIN MSSQLTips

Connect with MSSQLTips

Follow @mssqltips 8,204 followers

Follow 1.6k

Like 3.7k

Google™ Custom Search

SEARCH

I have been recommending that - for now - people stick to their tried and true methods of separate statements. Here are the major reasons:

Bugs with the SQL Server Merge Statement

It can be quite difficult to validate and guarantee that you are immune from any of the bugs that still exist. A few Connect items that you should be aware of, that are either still active, closed as "Won't Fix"/"By Design", or have only been fixed in specific versions (often requiring a cumulative update or on-demand hotfix):

Connect issue	Current status*
#773895 : MERGE Incorrectly Reports Unique Key Violations	(Active)
#771336 : Indexed view is not updated on data changes in base table	(Fixed in certain builds)
#766165 : MERGE evaluates filtered index per row, not post operation, which causes filtered index violation	(Won't Fix)
#723696 : Basic MERGE upsert causing deadlocks	(By Design)
#713699 : A system assertion check has failed ("cxrowset.cpp":1528)	(Won't Fix)
#699055 : MERGE query plans allow FK and CHECK constraint violations	(Active)
#685800 : Parameterized DELETE and MERGE Allow Foreign Key Constraint Violations	(Won't Fix)
#654746 : merge in SQL2008 SP2 still suffers from "Attempting to set a non-NULL-able column's value to NULL"	(Active)

Long-standing SQL Server MERGE issues - from *SQL Server Tips*, “Use caution with SQL Server's MERGE Statement”: http://www.mssqltips.com/sqlservertip/3074/use-caution-with-sql-servers-merge-statement

Issues with other implementations

- Duplicate key violation errors
- Blog author shows that SQL Server *doesn't* hold initial “intent update page” lock, and “update key” lock when it goes to do insert.
- One workaround is to use a hint that makes those locks last until end of transaction (or use 2PL-style `SERIALIZABLE` isolation level).
- Why don't they just do this all the time, given these locks are granular? Presumably has something to do with the deadlock implications, and/or performance.

http://weblogs.sqlteam.com/dang/archive/2009/01/31/UPSERT-Race-Condition-With-MERGE.aspx

What my proposed patch does

- Escalates buffer locks held as part of regular unique index enforcement to heavyweight page locks
- Reaches consensus across unique indexes if possible
- Staggers what is essentially the traditional unique index enforcement behavior across all unique indexes in attempt to reach consensus
- If not, goes to lock duplicate row *opportunistically*
- Original heavyweight page locks dropped
- So somewhat like SQL Server, except...
- ...theoretical lock starvation hazards exist, because we loop

Row locking and value locking conclusions

- It's kind of confusing that heap rows can *kind of* work like value locks just by having values (if and only if there is *already* a physical index tuple). I hope that the salient distinction is not lost on anybody.
- You can't reasonably tie the two, or expect to escalate from one to the other (i.e. hold value locks at the same time). Doing so doesn't help, and causes unprincipled deadlocks. This view seems to have won acceptance on -hackers.
- You can grab value locks and proceed with insertion proper if there is consensus to proceed. Otherwise, release value locks and go lock conflicting row.
- Value locks are only useful for when you actually insert. That's it.

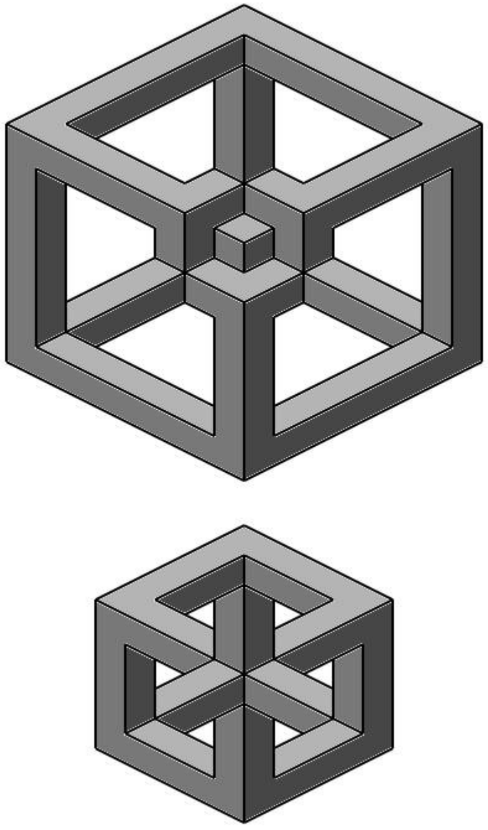
An aside on the proposed syntax

It looks something like this:

```
WITH rej AS
(
        INSERT INTO test(a, b)
        VALUES(123, 'Ottawa'),
              (456, 'Vancouver')
        ON DUPLICATE KEY LOCK FOR UPDATE
        RETURNING REJECTS a, b
)
UPDATE test SET test.b = rej.b FROM rej
WHERE test.a = rej.a;
```

- MERGE-like, but in some ways more flexible (in others, perhaps less).
- Like MERGE, this may DELETE rather than UPDATE
- Unlike MERGE, we can detect *where* conflicts occur

Visibility and the logically still-in-progress conflict TID problem



Problem Statement

“But let's back up and talk about MVCC for a minute. Suppose we have three source tuples, (1), (2), and (3); and the target table contains tuples (1) and (2), of which only (1) is visible to our MVCC snapshot; suppose also an equijoin. Clearly, source tuple (1) should fire the MATCHED rule and source tuple (3) should fire the NOT MATCHED rule, but what in the world should source tuple (2) do? AFAICS, the only sensible behavior is to throw a serialization error, because no matter what you do the results won't be equivalent to a serial execution of the transaction that committed target tuple (2) and the transaction that contains the MERGE.

Even with predicate locks, it's not obvious how to me how to solve this problem. Target tuple (2) may already be there, and its transaction already committed, by the time the MERGE statement gets around to looking at the source data.”

- Robert Haas (2010 SQL MERGE discussion)

http://www.postgresql.org/message-id/AANLkTineR-rDFWENeddLg=GrkT+epMHk2j9X0YqpiTY8@mail.gmail.com

Failures

- I think it is unacceptable to have `READ COMMITTED` isolation level throw serialization errors
- When upserting using the proposed syntax, clearly if a conclusively-committed conflict tuple is not visible to our snapshot, because its originating command logically “occurred later”, there isn't much point in locking it. Our snapshot cannot see it “yet” (i.e. only a new snapshot will).
- We ought to have the subsequent (say) UPDATE succeed.

`EvalPlanQual()`

- Postgres has always allowed `READ COMMITTED` level `UPDATE` (and `DELETE`) statements to “reach into the future” (rather than have a serialization failure) when another open/concurrently committed transaction modified the tuple to be updated or deleted.
- This is the `EvalPlanQual()` mechanism. Predicate re-evaluated. Query “re-run” for each modified tuple.
- `UPDATE`s may in effect “reach into the future”.
- This is rather odd, and certainly complicated, but ultimately the reason this happens is: Do you have a better idea?

Simple READ COMMITTED UPDATE

```
postgres=# explain analyze update orders set netamount = netamount + 1 where orderid = 5;
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Update on orders  (cost=0.29..8.31 rows=1 width=66) (actual time=0.768..0.768 rows=0 loops=1)
   ->  Index Scan using orders_pkey on orders  (cost=0.29..8.31 rows=1 width=66) (actual time=0.067..0.089 rows=1 loops=1)
         Index Cond: (orderid = 5)
 Planning time: 0.557 ms
 Execution time: 0.848 ms
(5 rows)
```

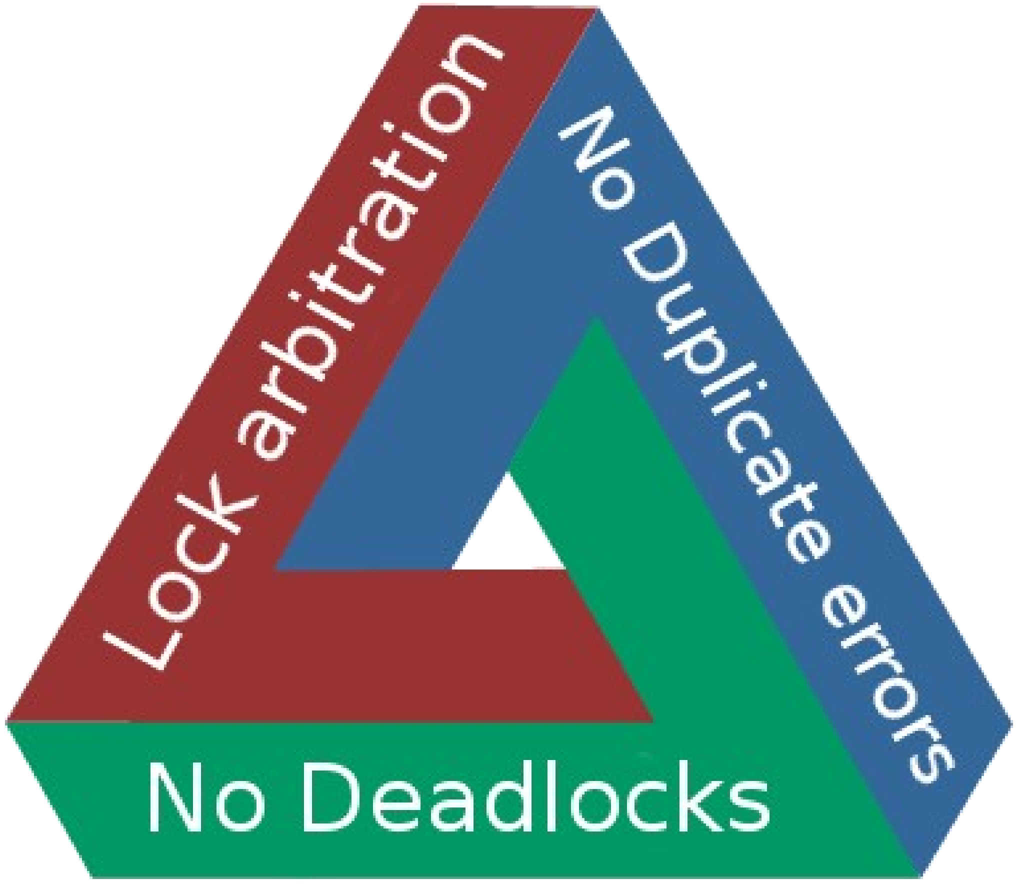
- An `Index Scan` node feeds an `ModifyTable/Update` node here.
- Ultimately, `ExecUpdate()` is called by the `ModifyTable` node.
- That's where the `EvalPlanQual()` magic may occur, where we "reach into the future" using a `DirtySnapshot`, etc.
- There is nothing special about the `Index Scan` node, though. It's using an ordinary MVCC snapshot.
- While we can "reach into the future", we still need to grab something in the present to get there. We can walk the update chain to re-evaluate, but we need to see *some* row version, fed to us by the `Index Scan` node, in order to even try to update anything in the first place.

See postgresql/src/backend/executor/README for description of EvalPlanQual()

What this means for UPSERT

- The `EvalPlanQual()` mechanism is a complex means of avoiding having to throw a serialization failure, and to do something reasonably non-surprising instead. It has been called “an MVCC violation”.
- A new “MVCC violation” may be called for. Altering the semantics of MVCC snapshots, again for the sole benefit of `READ COMMITTED` solves the problem.
- The idea here is that if you lock a tuple “in the future”, and therefore not visible under conventional MVCC rules, that row version becomes visible to you simply by virtue of having been locked by you (unless and until you UPDATE it, in which case only that new version is visible).
- So we'll “look into the future” to find a duplicate in the first place (much like the present unique index enforcement mechanism, with a `DirtySnapshot`), with a new novel behavior that has us “look into the future” on a limited basis a second time using an ordinary MVCC snapshot.
- This really has nothing in particular to do with the proposed syntax. If there was the full MERGE syntax, and you passed a TID around internally, that would be 100% equivalent, at least morally.

Pick any two



More on Spurious duplicate violations

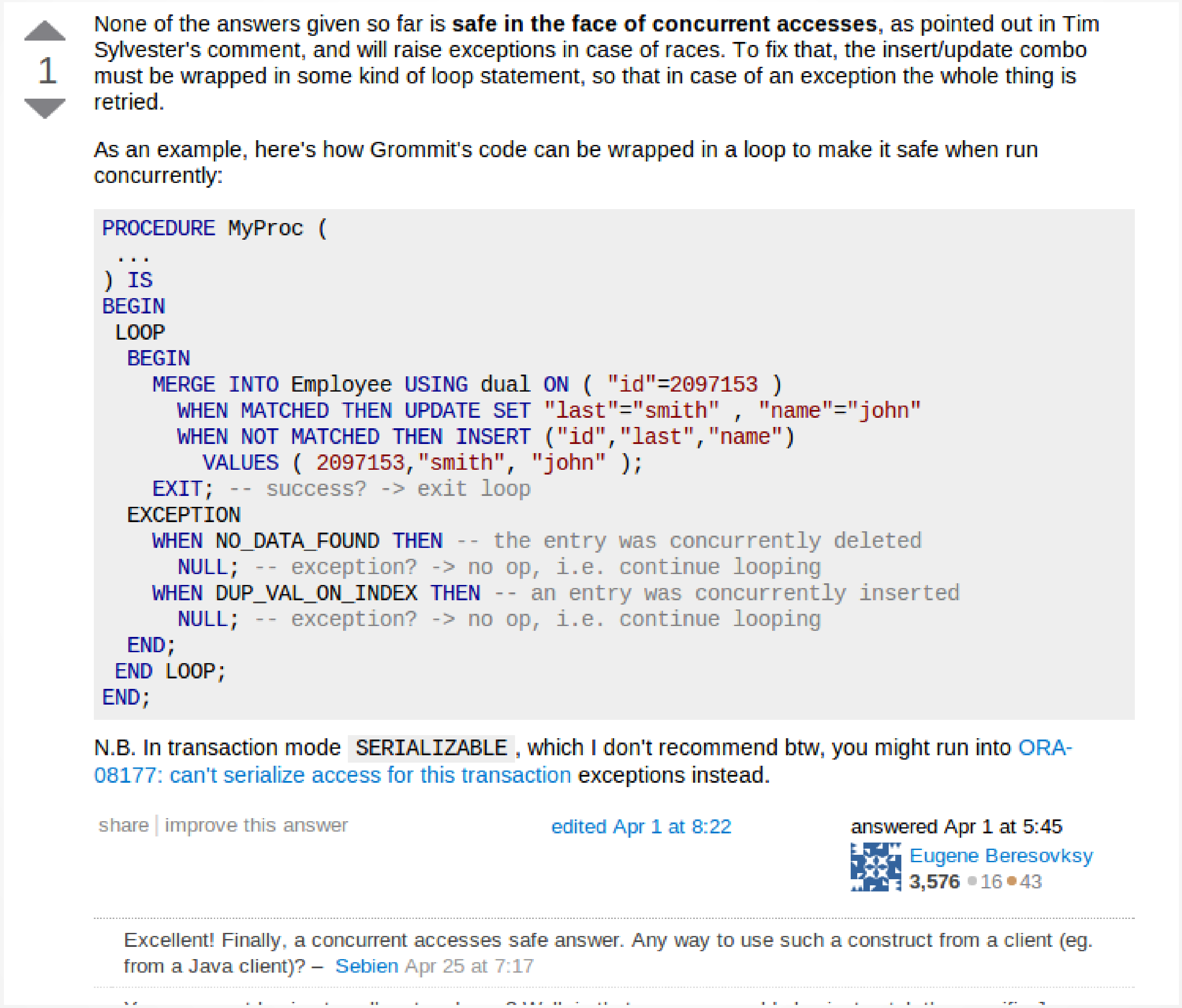
"It is possible for concurrent MERGE statements to cause duplicate INSERT violations because of a race condition between when we check whether the row is matching/not matching and when we apply the appropriate WHEN clause, if any. This is just the same as what we do now with try-UPDATE-then-INSERT logic."

- Simon Riggs (2008 SQL MERGE discussion)

- This is only true of current ad-hoc approaches if you don't loop – the PL/PgSQL example in the docs does, though (as does the proposed patch)
- Both SQL Server and Oracle SQL `MERGE` implementations seem to exhibit this behavior, even for trivial UPSERTs that don't `JOIN` tables (with only inline `using()...ON()` values, all distinct).

http://www.postgresql.org/message-id/1208372338.4259.202.camel@ebony.site
http://stackoverflow.com/questions/21904005/oracle-merge-constraint-violation-on-unique-key

Concurrency-safe UPSERT in Oracle



1

None of the answers given so far is **safe in the face of concurrent accesses**, as pointed out in Tim Sylvester's comment, and will raise exceptions in case of races. To fix that, the insert/update combo must be wrapped in some kind of loop statement, so that in case of an exception the whole thing is retried.

As an example, here's how Grommit's code can be wrapped in a loop to make it safe when run concurrently:

```
PROCEDURE MyProc (
 ...
) IS
BEGIN
 LOOP
  BEGIN
    MERGE INTO Employee USING dual ON ( "id"=2097153 )
      WHEN MATCHED THEN UPDATE SET "last"="smith" , "name"="john"
      WHEN NOT MATCHED THEN INSERT ("id","last","name")
        VALUES ( 2097153,"smith", "john" );
    EXIT; -- success? -> exit loop
  EXCEPTION
    WHEN NO_DATA_FOUND THEN -- the entry was concurrently deleted
      NULL; -- exception? -> no op, i.e. continue looping
    WHEN DUP_VAL_ON_INDEX THEN -- an entry was concurrently inserted
      NULL; -- exception? -> no op, i.e. continue looping
  END;
 END LOOP;
END;
```

N.B. In transaction mode `SERIALIZABLE`, which I don't recommend btw, you might run into ORA-08177: can't serialize access for this transaction exceptions instead.

share | improve this answer   edited Apr 1 at 8:22   answered Apr 1 at 5:45
Eugene Beresovksy
3,576 ●16 ●43

Excellent! Finally, a concurrent accesses safe answer. Any way to use such a construct from a client (eg. from a Java client)? – Sebien Apr 25 at 7:17

http://stackoverflow.com/a/22777749

The trade-off

- Pick any two (for `READ COMMITTED`):
  - 1) No deadlocking
  - 2) No unique constraint violations
  - 3) No lock starvation hazards
- Postgres users seem to already like having 1) and 2), and seem fine with the theoretical risk of 3).
- `SERIALIZABLE` isolation level can serve use cases that won't tolerate this, though
- *Maybe* you actually can “square the circle” and get all 3 with full predicate locking, or a full table lock, since either is a logical choke-point, but both of those alternatives are clearly nonstarters, mentioned here only for completeness

Summary

- Implementation involves intersection of some rather complex parts of the system
- Solutions proposed are pragmatic trade-offs
- Implementation is better in every way than the work-around that the documentation recommends, which is to perform ad-hoc looping with an inner subtransaction to manually implement UPSERT. Similar semantics, though.
- There are only weak lock arbitration rules. However, this doesn't seem to matter in practice, if the lack of actual complaints around our looping UPSERT example is anything to go on.
- Yes, I'm making a plausibility argument.

Thanks for listening!

Questions?