How do I do large non-blocking updates in PostgreSQL?

https://stackoverflow.com/questions/1113277/how-do-i-do-large-non-blocking-updates-in-postgresql

I want to do a large update on a table in PostgreSQL, but I don't need the transactional integrity to be maintained across the entire operation, because I know that the column I'm changing is not going to be written to or read during the update. I want to know if there is an easy way *in the psql console* to make these types of operations faster.

For example, let's say I have a table called "orders" with 35 million rows, and I want to do this:

```
UPDATE orders SET status = null;
```

To avoid being diverted to an offtopic discussion, let's assume that all the values of status for the 35 million columns are currently set to the same (non-null) value, thus rendering an index useless.

The problem with this statement is that it takes a very long time to go into effect (solely because of the locking), and all changed rows are locked until the entire update is complete. This update might take 5 hours, whereas something like

```
UPDATE orders SET status = null WHERE (order_id > 0 and order_id < 1000000);
```

might take 1 minute. Over 35 million rows, doing the above and breaking it into chunks of 35 would only take 35 minutes and save me 4 hours and 25 minutes.

I could break it down even further with a script (using pseudocode here):

```
for (i = 0 to 3500) {
  db_operation ("UPDATE orders SET status = null
                 WHERE (order_id >" + (i*1000)"
             + " AND order_id <" + ((i+1)*1000) " +  ")");
}
```

This operation might complete in only a few minutes, rather than 35.

So that comes down to what I'm really asking. I don't want to write a freaking script to break down operations every single time I want to do a big one-time update like this. Is there a way to accomplish what I want entirely within SQL?

postgresql transactions sql-update plpgsql dblink

I am not a PostgreSQL guy, but have you tried setting up an index on the status column? – Kirtan Jul 11 '09 at 8:59

It would not help much in this instance because the vast majority of time is spent in the effort to maintain transactional integrity. My example might be a bit misleading; Instead, imagine I just want to do this: UPDATE orders SET status = null; Everything I said above still applies (but an index here obviously wouldn't help) – S D Jul 11 '09 at 9:05

In fact, I just updated the question to reflect this. – S D Jul 11 '09 at 9:06
add a comment

7 Answers

Column / Row

> ... I don't need the transactional integrity to be maintained across the entire operation, because I know that the column I'm changing is not going to be written to or read during the update.

Any UPDATE in PostgreSQL's MVCC model writes a new version of *the whole row*. If concurrent transactions change *any* column of the same row, time-consuming concurrency issues arise.

Details in the manual. Knowing the same *column* won't be touched by concurrent transactions

avoids *some* possible complications, but not others.

Index

> To avoid being diverted to an offtopic discussion, let's assume that all the values of status for the 35 million columns are currently set to the same (non-null) value, thus rendering an index
>
> useless.

When updating the *whole table* (or major parts of it) Postgres *never uses an index*. A sequential scan is

faster when all or most rows have to be read. On the contrary: Index maintenance means additional

cost for the UPDATE.

Performance

> For example, let's say I have a table called "orders" with 35 million rows, and I want to do this:
>
> ```
> UPDATE orders SET status = null;
> ```

I understand you are aiming for a more general solution (see below). But to address *the actual*

question asked: This can be dealt with in *a matter milliseconds*, regardless of table size:

```
ALTER TABLE orders DROP column status
                 , ADD  column status text;
```

Per documentation:

> When a column is added with `ADD COLUMN`, all existing rows in the table are initialized with the column's default value (`NULL` if no `DEFAULT` clause is specified).

And:

> The `DROP COLUMN` form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated. (These statements do not apply when dropping the system oid column; that is done with an immediate rewrite.)

Make sure you don't have objects depending on the column (foreign key constraints, indices, views, ...).

You would need to drop / recreate those. Barring that, tiny operations on the system catalog table `pg_attribute` do the job. Requires an **exclusive lock** on the table which may be a problem for

heavy concurrent load. Since it only takes a few milliseconds, you should still be fine.

If you have a column default you want to keep, add it back *in a separate command*. Doing it in the

same command would apply it to all rows immediately, voiding the effect.
Follow the link and read the *Notes* in the manual.

General solution

`dblink` has been mentioned in another answer. It allows access to "remote" Postgres databases in

implicit separate connections. The "remote" database can be the current one, thereby achieving

"autonomous transactions": what the function writes in the "remote" db is committed and can't be

rolled back.

This allows to run a single function that updates a big table in smaller parts and each part is committed separately. Avoids building up transaction overhead for very big numbers of rows and, more importantly, releases locks after each part. This allows concurrent operations to proceed without much delay and

makes deadlocks less likely.

If you don't have concurrent access, this is hardly useful - except to avoid `ROLLBACK` after an exception. Also consider `SAVEPOINT` for that case.

Disclaimer

First of all, lots of small transactions are actually more expensive. This *only makes sense for big tables*.

The sweet spot depends on many factors.

If you are not sure what you are doing: *a single transaction is the safe method*. For this to work properly, concurrent operations on the table have to play along. For instance: concurrent *writes* can move a row to a partition that's supposedly already processed. Or concurrent reads can see inconsistent intermediary states. *You have been warned.*

Step-by-step instructions

The additional module dblink needs to be installed first:

- How to use (install) dblink in PostgreSQL?

Setting up the connection with dblink very much depends on the setup of your DB cluster and security policies in place. It can be tricky. Related later answer with more **how to connect with dblink**:

- Persistent inserts in a UDF even if the function aborts

Create a `FOREIGN SERVER` and a `USER MAPPING` as instructed there to simplify and streamline the connection (unless you have one already).
Assuming a `serial PRIMARY KEY` with or without some gaps.

```
CREATE OR REPLACE FUNCTION f_update_in_steps()
  RETURNS void AS
$func$
DECLARE
   _step int;   -- size of step
   _cur  int;   -- current ID (starting with minimum)
   _max  int;   -- maximum ID
BEGIN
   SELECT INTO _cur, _max  min(order_id), max(order_id) FROM orders;
                                        -- 100 slices (steps) hard coded
   _step := ((_max - _cur) / 100) + 1;  -- rounded, possibly a bit too small
                                        -- +1 to avoid endless loop for 0
   PERFORM dblink_connect('myserver');  -- your foreign server as instructed above

   FOR i IN 0..200 LOOP                 -- 200 >> 100 to make sure we exceed _max
      PERFORM dblink_exec(
       $$UPDATE public.orders
         SET    status = 'foo'
```

```
         WHERE  order_id >= $$ || _cur || $$
         AND    order_id <  $$ || _cur + _step || $$
         AND    status IS DISTINCT FROM 'foo'$$);  -- avoid empty update

      _cur := _cur + _step;

      EXIT WHEN _cur > _max;            -- stop when done (never loop till 200)
   END LOOP;

   PERFORM dblink_disconnect();
END
$func$  LANGUAGE plpgsql;
```

Call:

```
SELECT f_update_in_steps();
```

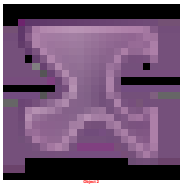
You can parameterize any part according to your needs: the table name, column name, value, ...

just be sure to sanitize identifiers to avoid SQL injection:

- Table name as a PostgreSQL function parameter

About avoiding empty UPDATE:

- How do I (or can I) SELECT DISTINCT on multiple columns?



I would use CTAS:

```
begin;
create table T as select col1, col2, ..., <new value>, colN from orders;
drop table orders;
alter table T rename to orders;
commit;
```

> Probably the best solution if (but only if) the table's other columns will not be modified in the time it takes to do this. – fzzfzzfzz May 5 '16 at 17:36

add a comment

Postgres uses MVCC (multi-version concurrency control), thus avoiding any locking if you are the only writer; any number of concurrent readers can work on the table, and there won't be any locking.

So if it really takes 5h, it must be for a different reason (e.g. that you *do* have concurrent writes,

contrary to your claim that you don't).

The times that I have quoted above (5 hours, 35 minutes, ~3 minutes) are accurate for the scenarios I described above. I didn't state that there were no other writes happening in the database; just that I know that no one is going to be writing to the *column* while I'm doing the update (this column is not being used by the system at all, the rows are read/written though). In other words, I don't care if this work is processed in one huge transaction or in smaller pieces; what I'm concerned about is speed. And I can increase speed using the methods above, but they are cumbersome. – S D Jul 11 '09 at 9:30

It's still not clear whether the long run-time is due to the locking, or, say, vacuuming. Try acquiring a table lock before the update, locking out any other kind of operation. Then you should be able to complete this update without any interference. – Martin v. Löwis Jul 11 '09 at 10:06

If I lock every other kind of operation, then the system risks being stalled until it's complete. Whereas the two solutions I have posted for reducing the time to 35min/3min do not prevent the system from functioning normally. What I'm looking for is a way to do so without having to write a script each time I want to do an update like this (which would save me 5 minutes each time I wanted to do one of these updates). – S D Jul 11 '09 at 19:34

add a comment

You should delegate this column to another table like this:

```
create table order_status (
  order_id int not null references orders(order_id) primary key,
  status int not null
);
```

Then your operation of setting status=NULL will be instant:

```
truncate order_status;
```

add a comment

First of all - are you sure that you need to update all rows?

Perhaps some of the rows already have `status` NULL?

If so, then:

```
UPDATE orders SET status = null WHERE status is not null;
```

As for partitioning the change - that's not possible in pure sql. All updates are in single transaction.

One possible way to do it in "pure sql" would be to install dblink, connect to the same database using dblink, and then issue a lot of updates over dblink, but it seems like overkill for such a simple task.

Usually just adding proper where solves the problem. If it doesn't - just partition it manually. Writing a script is too much - you can usually make it in a simple one-liner:

```
perl -e '
    for (my $i = 0; $i <= 3500000; $i += 1000) {
        printf "UPDATE orders SET status = null WHERE status is not null
                and order_id between %u and %u;\n",
        $i, $i+999
    }
'
```

I wrapped lines here for readability, generally it's a single line. Output of above command can be fed to psql directly:

```
perl -e '...' | psql -U ... -d ...
```

Or first to file and then to psql (in case you'd need the file later on):

```
perl -e '...' > updates.partitioned.sql
psql -U ... -d ... -f updates.partitioned.sql
```

I appreciate your response, but it is basically identical to my #3 solution in my question; basically, this is what I already do. However, it takes 5 minutes to write out a script like this, whereas I'm trying to figure out a way to just do it within psql and therefore do it in 20 seconds or less (and also eliminate potential typos/bugs). That's the question I'm asking. – S D Jul 11 '09 at 19:40

And I thought I answered it - it is not possible to do it in SQL (unless using tricks like dblink). On the other hand - I wrote that one-liner that I showed in around 30 seconds, so it doesn't look like too much time :) It's definitely closer to your 20 second target, than hypothetical 5-minute script writing.
– user80168 Jul 11 '09 at 23:02

Thanks, but I misspoke when I said 'SQL'; in fact I'm asking how to do it in the psql console in PostgreSQL, using any tricks possible, including plgpsql. Writing the script as above is exactly what I'm doing now. It takes more than 30 seconds because you have to write a custom mini-script every time you do one of these updates, and you have to do a query to find out how many rows you have, and you have to make sure there are no typos, etc etc. What I'd like to do is something like: # select nonblocking_query('update orders set status=null'); That is what I am trying to accomplish. – S D Jul 12 '09 at 10:44

And this is what I already 2 times answered: it's not possible, unless you will use dblink, but this is even more complicated than those one-liners you don't like. – user80168 Jul 12 '09 at 12:28
add a comment

I am by no means a DBA, but a database design where you'd frequently have to update 35 million rows might have… issues.

A simple `WHERE status IS NOT NULL` might speed up things quite a bit (provided you have an index on status) – not knowing the actual use case, I'm assuming if this is run frequently, a great part of the 35 million rows might already have a null status.

However, you can make loops within the query via the LOOP statement. I'll just cook up a small example:

```
CREATE OR REPLACE FUNCTION nullstatus(count INTEGER) RETURNS integer AS $$
DECLARE
    i INTEGER := 0;
BEGIN
    FOR i IN 0..(count/1000 + 1) LOOP
        UPDATE orders SET status = null
WHERE (order_id > (i*1000) and order_id <((i+1)*1000));
        RAISE NOTICE 'Count: % and i: %', count,i;
    END LOOP;
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

It can then be run by doing something akin to:

```
SELECT nullstatus(35000000);
```

You might want to select the row count, but beware that the exact row count can take a lot of time.

The PostgreSQL wiki has an article about slow counting and how to avoid it.

Also, the RAISE NOTICE part is just there to keep track on how far along the script is.

If you're not monitoring the notices, or do not care, it would be better to leave it out.

This will not help as function call will be in single transaction - so, the locking issue will be still there. – user80168 Jul 11 '09 at 10:11

Hmm, I had not considered that – still, I think this will be faster than UPDATE orders SET status = null;, since that would mean a full table scan. – mikl Jul 11 '09 at 10:15

I understand the interest in the query running faster with an index, but that's not really my concern, as in some cases every value of the column is the same, rendering an index useless. I'm really concerned in the difference in time between running this query as one operation (5 hours) and breaking it up into pieces (3 minutes) and wanting to do so within psql without having to write a script every time. I do know about indexes and how to possibly save even more time on these operations by using them. – S D Jul 11 '09 at 19:37

Oh, and to answer the first part of your question: it is indeed rare to have to update 35 million rows. This is mostly for cleanup; for example, we might decide, "why does order_status = 'a' mean 'accepted' for the orders table and 'annuled' for the shipping table? we should make these consistent!" and so we need to update the code and do a mass update to the database to clean up the inconsistency. Of course this is an abstraction, as we don't actually have "orders" at all. – S D Jul 11 '09 at 19:42

add a comment

Are you sure this is because of locking? I don't think so and there's many other possible reasons. To find out you can always try to do just the locking. Try this: BEGIN; SELECT NOW(); SELECT * FROM order FOR UPDATE; SELECT NOW(); ROLLBACK;

To understand what's really happening you should run an EXPLAIN first (EXPLAIN UPDATE orders SET status...) and/or EXPLAIN ANALYZE. Maybe you'll find out that you don't have enough memory to do the UPDATE efficiently. If so, SET work_mem TO 'xxxMB'; might be a simple solution.

Also, tail the PostgreSQL log to see if some performance related problems occurs.

PHP

How do I (or can I) SELECT DISTINCT on multiple columns?
35
How to use (install) dblink in PostgreSQL?
41
Table name as a PostgreSQL function parameter
38
Simulate CREATE DATABASE IF NOT EXISTS for PostgreSQL?
9
Does Postgres support nested or autonomous transactions?
11
Executing a trigger AFTER the completion of a transaction
6
PostgreSQL obtain and release LOCK inside stored function
1
Trigger to insert rows in remote database after deletion
0
Table Locking in PostgreSQL
0
Fastest way to anonymize and sanitize field in db table?
see more linked questions…

Related

1131
PostgreSQL “DESCRIBE TABLE”
12
Update VERY LARGE PostgreSQL database table efficiently
1
Sync transactions between databases in PostgreSQL using dblink
1223
How to exit from PostgreSQL command line utility: psql
2
Drop or create database from stored procedure in PostgreSQL
2
PostgreSQL return table with update and select statement causing ambiguity
0
Update between 2 databases using dblink not working
0
How to INSERT/UPDATE/DELETE in a nested transaction in PostgreSQL (plpgsql)
4
Calling a function for each updated row in postgresql
0
Update large amount of data postgresql