Member only story

Postgres Locks — A Deep Dive

Jeremy Kevan
Software Engineer · Nov 24, 2021

A VACUUM FULL and a SELECT on a table are both blocked by a RW lock (Exclusive).

I used to think database locks are two-types, shared and exclusive. Readers acquire a shared lock so other readers can read the same data concurrently, and writer can acquire an exclusive lock. If a reader has an exclusive lock no one can acquire shared locks and as a result no one can read (but the writer). When I started a big data Postgres this binary view of locking changed completely and I understand why.

Because, in Postgres there are fine-lock categories (and over 13 individual lock modes). The former binary lock commonly used elsewhere obtains lock is less relevant than knowing which command can conflict which command. So conflict here I mean commands block each other and can't run concurrently.

In this blog I explore all types and categories of locks and at the end I show how I used this knowledge to fix a Postgres production issue where a simple conflict rule select/update blocks the 1m cat get huge looking at a marble.

All the information here are official Postgres docs, source code and tell two testing on the app.

Table Locks

Table locks are used on a table lock those years ago I would say its a lock you plant on a table so no one can do anything on that table while you hold that lock. No inserts, updates or deletes or any DDLs. But that's definitely far from the truth in Postgres. There are eight types of table locks in Postgres and transactions can have multiple table locks on the same table. Some of these locks conflict, some don't. Let us explore them one by one. You can view the table locks in pseudo column in pg_locks table.

Source Code for Table Locks

ACCESS EXCLUSIVE

ACCESS EXCLUSIVE is AccessExclusiveLock is the code's is the most aggressive table lock. If a command obtains this lock type on a table nothing can be done to this table as this lock type conflicts with all other table locks. Commonly, this lock is obtained by DDL commands like DROP TABLE, ALTER TABLE, alter a column, create an index and even system operations such as VACUUM cannot execute on the table. It is a complete block.

Source

Most operations and commands in Postgres obtain this lock? I went through the the pages and found all commands that obtain this kind of lock. Here they are, when anything in this list you, you can't do anything to this table.

```
LOCK [ TABLE ]
ALTER COLLATION
ALTER DOMAIN
ALTER FUNCTION
ALTER GROUP
ALTER INDEX
ALTER LANGUAGE
ALTER LARGE OBJECT
ALTER MATERIALIZED VIEW
ALTER OPERATOR
ALTER OPERATOR CLASS
ALTER OPERATOR FAMILY
ALTER POLICY
ALTER PROCEDURE
ALTER ROUTINE
ALTER RULE
ALTER SCHEMA
ALTER SEQUENCE
ALTER TABLE
ALTER TABLESPACE
ALTER TEXT SEARCH CONFIGURATION
ALTER TEXT SEARCH DICTIONARY
ALTER TRIGGER
ALTER TYPE
ALTER USER MAPPING
ALTER VIEW
COMMENT
CREATE CAST
CREATE INDEX
CREATE LANGUAGE
CREATE OPERATOR CLASS
CREATE OPERATOR FAMILY
CREATE PUBLICATION
CREATE RULE
CREATE SCHEMA
CREATE SEQUENCE
CREATE STATISTICS
CREATE SUBSCRIPTION
CREATE TABLE
CREATE TABLE AS
CREATE TABLESPACE
CREATE TEXT SEARCH CONFIGURATION
CREATE TEXT SEARCH DICTIONARY
CREATE TRIGGER
CREATE TYPE
DEFINE VIEW
DROP [ TABLE ]
DROP AGGREGATE
DROP CAST
DROP COLLATION
DROP CONVERSION
DROP DOMAIN
DROP FUNCTION
DROP GROUP
DROP INDEX
DROP LANGUAGE
DROP OPERATOR
DROP OPERATOR CLASS
DROP OPERATOR FAMILY
DROP OWNED
DROP POLICY
DROP PROCEDURE
DROP ROUTINE
DROP RULE
DROP SCHEMA
DROP SEQUENCE
DROP TABLE
DROP TABLESPACE
DROP TEXT SEARCH CONFIGURATION
DROP TEXT SEARCH DICTIONARY
DROP TRIGGER
DROP TYPE
DROP USER MAPPING
DROP VIEW
GRANT
IMPORT FOREIGN SCHEMA
REFRESH MATERIALIZED VIEW
REINDEX
REVOKE
SECURITY LABEL
TRUNCATE
VACUUM
```

This means for example, if you run VACUUM FULL for instance on a table, you can't select or update to this table.

You might say why this lock is so aggressive and not blocks, all the reason because different other tables obtain different types of locks which makes some jobs that serious operations while others might not.

ACCESS SHARE

ACCESS SHARE or AccessShareLock is the lightest weight lock type. Only two commands that I'm aware of acquire this lock and those are SELECT and COPY TO. It is an indication that someone is reading the table, whether it is a single row, all the rows and even no rows in your file. It is query on a table that returned nothing, that lock is also acquired (I tested it).

```
LOCK [ TABLE ]
SELECT
COPY TO
```

Select

This lock type only conflicts with the ACCESS EXCLUSIVE which makes it easy to understand, if you a transaction that does a select you will get a VACUUM FULL (instead VACUUM is fine). The reason is VACUUM FULL or any of the commands that acquire ACCESS EXCLUSIVE does original changes to the layout of the table which will break consistency when selects are executing against the table. Vacuum full actually changes rows, it creates tuples and rewriting data, we can't be having people reading the table while this is happening. This is as opposed to normal VACUUM which really (almost) are the both option and defrost.

EXCLUSIVE

The EXCLUSIVE or ExclusiveLock is very similar to the ACCESS EXCLUSIVE except it doesn't conflict with reads acquired by ACCESS SHARE. This means you can do selects while an EXCLUSIVE table lock is on the table.

Select

The odd thing I only found one command (REFRESH MATERIALIZED VIEW CONCURRENTLY) that acquires this lock. If I had to guess, this lock type was added for this command. This command allows concurrent updates and select from the table at the same time. The default materialized view acquires an ACCESS EXCLUSIVE blocking selects, so Postgres added both a new lock type Exclusive which conflicts with everything except ACCESS SHARE and then built a new command to allow refreshing the view concurrently. I'm sure more methods will fit into this exclusive lock type.

```
LOCK [ TABLE ]
REFRESH MATERIALIZED VIEW CONCURRENTLY
```

So you if you refresh your materialized view concurrently your table can't be edited but can be read.

ROW SHARE

Rarely (or to narrow only get more confusing from here on) ROW SHARE or RowShareLock is similar to ACCESS SHARE but was designed for the SELECT FOR commands family. That is probably why it has the same RW is. While SELECT FOR UPDATE, SELECT FOR SHARE and others work on rows remember this is still a table lock. So these kind of commands will acquire two types of locks locks (will see later) and the ROW SHARE one is table lock.

Source Code for Row Share

This lock type conflicts with ACCESS EXCLUSIVE and the EXCLUSIVE locks. Which means anything that is acquired by the ACCESS EXCLUSIVE can refresh materialized view concurrently.

Select Operations

Here is a list of commands that acquire ROW SHARE.

```
LOCK [ TABLE ]
ALTER TABLE [ NO ] INHERIT
SELECT FOR UPDATE NOWAIT
SELECT FOR SHARE
SELECT FOR KEY SHARE
```

So while row you can do normal SELECTs while refreshing your materialized view concurrently, you can't really do a SELECT FOR SHARE for instance.

ROW EXCLUSIVE

The ROW EXCLUSIVE or RowExclusiveLock is obtained by DML: Insert, Update, Delete, Merge and Copy From. If you care about write latency in your table watch out for operations that conflict with this lock. Thus the name ROW is the lock name because methods often operates on rows.

Source Code for Row Exclusive

Unity watch out again, you might you a update or a delete that and nothing happens. Well, it could be waiting for someone else. So if you have long running transactions watch out for deadlocks.

Select Operations

The methods acquiring ROW EXCLUSIVE are as follows

```
LOCK [ TABLE ]
UPDATE
DELETE
INSERT
MERGE
COPY FROM
```

SHARE ROW EXCLUSIVE

SHARE ROW EXCLUSIVE or ShareRowExclusiveLock is similar to EXCLUSIVE but is relaxed so that ROW SHARE don't conflict, so you may do the SELECT FOR with this type of lock but you still can't do any modifications through DML's. So methods that obtain this type of lock wear SELECT FOR and the SELECT FOR lock locks. In case you have long updates why not just use EXCLUSIVE that is why. It is all about relaxing and minimizing locks.

Source Code for Share Row Exclusive

What is interesting about SHARE ROW EXCLUSIVE is it does conflict with it self. So only one transaction can have this lock. This is a good thing because we don't want to run two create triggers (which is a method that requires SHARE ROW EXCLUSIVE) at the same time on the same table, my guess is one create trigger might overly rows in the table and the other create trigger might also change the table and we don't want that.

Again remember if a transaction obtains a SHARE ROW EXCLUSIVE it can still make modifications to rows, other transactions can't.

Select Operations

Here are the methods that obtain this type.

```
LOCK [ TABLE ]
ALTER TABLE [ NO ] TRIGGER
ALTER TABLE [ NO ] ENABLE ALWAYS | REPLICA
CREATE TRIGGER
CREATE COLLATION
CREATE DOMAIN
CREATE FUNCTION
CREATE INDEX
CREATE OPERATOR
CREATE OPERATOR CLASS
CREATE OPERATOR FAMILY
CREATE PUBLICATION
CREATE RULE
CREATE SCHEMA
CREATE SEQUENCE
CREATE STATISTICS
CREATE SUBSCRIPTION
CREATE TABLE
CREATE TABLE AS
CREATE TABLESPACE
CREATE TEXT SEARCH CONFIGURATION
CREATE TEXT SEARCH DICTIONARY
CREATE TRIGGER
CREATE TYPE
DEFINE VIEW
DROP [ TABLE ]
DROP AGGREGATE
DROP CAST
DROP COLLATION
DROP CONVERSION
DROP DOMAIN
DROP FUNCTION
DROP GROUP
DROP INDEX
DROP LANGUAGE
DROP OPERATOR
DROP OPERATOR CLASS
DROP OPERATOR FAMILY
DROP OWNED
DROP POLICY
DROP PROCEDURE
DROP ROUTINE
DROP RULE
DROP SCHEMA
DROP SEQUENCE
DROP TABLE
DROP TABLESPACE
DROP TEXT SEARCH CONFIGURATION
DROP TEXT SEARCH DICTIONARY
DROP TRIGGER
DROP TYPE
DROP USER MAPPING
DROP VIEW
GRANT
IMPORT FOREIGN SCHEMA
REFRESH MATERIALIZED VIEW
REINDEX
REVOKE
SECURITY LABEL
TRUNCATE
VACUUM
```

SHARE

The SHARE (Share) lock is similar to the SHARE ROW EXCLUSIVE in a sense it blocks concurrent modifications but it doesn't conflict with itself. Only CREATE INDEX obtain this type of lock, which means while creating an index you can't change the data (because the index is reading the table and building the b-tree) but technically nothing stopping. Another interesting point running CONCURRENTLY on the same table, so if you are blocking updates and you want to create an index you can do it. Well, not quite, read more. Of course you can also use the CREATE INDEX CONCURRENTLY which allows concurrent modifications but that doesn't run in a transaction.

```
postgres=*> begin;
BEGIN
postgres=*> lock test;
postgres=*> create index c1dx on test(c1);

CREATE INDEX
postgres=*>

postgres=*> begin;
BEGIN
postgres=*> create index c2dx on test(c2);
CREATE INDEX
postgres=*>
```

Source Code for Share

```
LOCK [ TABLE ]
CREATE INDEX
```

SHARE UPDATE EXCLUSIVE

The SHARE UPDATE EXCLUSIVE or ShareUpdateExclusiveLock is designed for methods who want to allow concurrent writes and reads but prevent schema changes and VACUUM runs. Normal VACUUM for instance acquire this lock which is why you can run VACUUM and still do edit to your table otherwise it will be a disaster.

Source Code for Share Update Exclusive

CREATE INDEX CONCURRENTLY is another interesting one where you can create an index and allow writes. This lock conflict with itself so no two VACUUMs can run concurrently and no two CREATE INDEX CONCURRENTLY will. This also explains why many forms of ALTER TABLE can't run concurrently. This type of lock, you want to allow edits but no tweaks of the same table.

Following are the commands that acquire SHARE UPDATE EXCLUSIVE.

```
LOCK [ TABLE ]
ALTER DATABASE
ALTER INDEX
ALTER SCHEMA
ALTER SEQUENCE
ALTER TABLE
ALTER TABLESPACE
ALTER TEXT SEARCH CONFIGURATION
ALTER TEXT SEARCH DICTIONARY
ALTER TRIGGER
ALTER TYPE
ALTER USER MAPPING
ALTER VIEW
COMMENT
CREATE INDEX
CREATE STATISTICS
CREATE SUBSCRIPTION
CREATE TABLE
CREATE TABLE AS
CREATE TABLESPACE
CREATE TEXT SEARCH CONFIGURATION
CREATE TEXT SEARCH DICTIONARY
CREATE TRIGGER
CREATE TYPE
DEFINE VIEW
DROP [ TABLE ]
DROP AGGREGATE
DROP CAST
DROP COLLATION
DROP CONVERSION
DROP DOMAIN
DROP FUNCTION
DROP GROUP
DROP INDEX
DROP LANGUAGE
DROP OPERATOR
DROP OPERATOR CLASS
DROP OPERATOR FAMILY
DROP OWNED
DROP POLICY
DROP PROCEDURE
DROP ROUTINE
DROP RULE
DROP SCHEMA
DROP SEQUENCE
DROP TABLE
DROP TABLESPACE
DROP TEXT SEARCH CONFIGURATION
DROP TEXT SEARCH DICTIONARY
DROP TRIGGER
DROP TYPE
DROP USER MAPPING
DROP VIEW
GRANT
IMPORT FOREIGN SCHEMA
REFRESH MATERIALIZED VIEW
REINDEX
REVOKE
SECURITY LABEL
TRUNCATE
VACUUM
```

Source Code for Share Update Exclusive

Row Locks

Now that we talked about table locks time to go one level deeper to row locks. Row locks are critical to detect changes to row objects as we proceed two transactions changing the same row which results in lost updates.

Worth noting that SERIALIZABLE Isolation level requires row locks to prevent lost updates and writes in the transaction that contains those locks. Meaning, you may Postgres doesn't support read uncommitted isolation level.

The methods that lock rows are limited to SELECT, UPDATE (FOR KEY), UPDATE (key), and all the SELECT FOR.

Source Code for Row Locks

UPDATE (FOR KEY) is an update to a column that doesn't have a unique index while UPDATE (KEY) is an update to a column that does have unique index. Those two acquire different locks that is why they are spelled out.

Here are four row locks in Postgres we discuss them here.

FOR UPDATE

FOR UPDATE is the highest row lock, when a row is locked FOR UPDATE you cannot delete or update it or do a SELECT FOR UPDATE on it. However you can still read it through a normal SELECT, if you want your selects to be blocked you can specify a lock type may use SELECT FOR UPDATE NOWAIT instead which conflicts.

Source Code for Row Locks

The following commands acquire a FOR UPDATE row lock.

```
LOCK [ TABLE ]
DELETE
ALTER TABLE [ NO ] INHERIT
SELECT FOR UPDATE
```

FOR NO KEY UPDATE

This lock is acquired by UPDATES to columns without unique index, so it is weaker than FOR UPDATE as it allows SELECT FOR KEY SHARE.

```
LOCK [ TABLE ]
UPDATE [ WHERE ... ] RETURNING * | output_expression AS output_name [, ...]
SELECT FOR NO KEY UPDATE
```

FOR SHARE

This is the most shared lock. transactions can acquire multiple FOR SHARE locks on a row. When a row is FOR SHARE no DML can modify it.

```
LOCK [ TABLE ]
SELECT FOR SHARE
```

FOR KEY SHARE

The weakest row lock, between like FOR SHARE but allows updates to columns without unique indices.

```
LOCK [ TABLE ]
SELECT FOR KEY SHARE
```

Row Lock Matrix

This matrix shows the 4 row locks and how they conflict with each other. The test I wrote goes over visibility to the commands that block each other.

	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

Postgres table locks are in memory, row locks are stored in the tuple (xmax system field), which saves memory at a cost of potential disk writes. This is not bad for dishes or updates because we are technically tracking the row but also for updates for instance, those read operations can cause some pages to get dirty which will trigger the background writer to flush them to disk.

Row locks are memory intensive in other database, I work with SQL Server almost on daily basis and if row locks are enabled SQL Server can easily run out of memory and fail the transaction when row locks will be acquired. That is why I like the approach the Postgres row lock design. Nothing is free though, disk writes.

Source

Page locks

A postgres page is 8Kb and stores tuples for table and indexes. Because the page is an in memory data structure, it needs to be protected from concurrent process accesses. You see, Postgres design is process based unlike other database, each process (client) connects to the database via a process based process, and those backend processes compete to access pages stored in shared buffer pool.

Source

It is not as bad if multiple processes reading the same page, but it is a problem if one process is modifying a page while others are in the same page. You want to serialize those accesses so you don't corrupt data. This is classic operating system concepts with mutex and semaphores.

Dead Locks

Dead locks happen when two transactions each holding different locks attempting to access each other's resources and end up in an indefinite wait. Postgres detects dead locks and kills one of the transactions for more.

Here is an example (while unlikely it can happen)

Source: Postgres

```
Tx1
BEGIN;
-- ACQUIRES ExclusiveRowLock ON
UPDATE t SET foo=1 WHERE id=1;

Tx2
BEGIN;
-- ACQUIRES ExclusiveRowLock ON
UPDATE t SET foo=2 WHERE id=2;
-- ACQUIRES ON id=1
UPDATE t SET foo=2 WHERE id=1;
-- BLOCKED BY Tx1 FOR ROW LOCK ON id=1 WAITING TO Tx1
-- TO COMMIT OR ROLLBACK

-- ACQUIRES ON id=2 WAITING
UPDATE t SET foo=1 WHERE id=2;
-- BLOCKED BY Tx2 FOR ROW LOCK ON id=2 WAITING
-- TO COMMIT OR ROLLBACK
-- Tx1 ROLLBACK
-- Tx2 ROLLBACK
-- ERROR: ERROR: deadlock detected
-- DETAIL: Process 16384 waits for ShareLock on transaction 1; blocked by
-- Process 16385.
-- Process 16385 waits for ShareLock on transaction 2; blocked by
-- Process 16384.
-- HINT: See server log for query details.
-- CONTEXT: while updating tuple (0,1) in relation "t"
TRUNCATE TABLE TEST;
```

Page Locks

A page lock is a lock on a page, it locks the page and allows other processes to obtain other unique tuples.

Source

Tuple Locks

Tuple locks are used to lock a tuple (row) and allow other processes to obtain other tuples in the same page. Tuple locks are stored in the tuple header (xmax system field) and are persisted to disk. Tuple locks are used to serialize access to a tuple, for example if two transactions are updating the same tuple the second one will wait for the first one to finish.

Source

Lock Monitoring

Postgres has a system table called pg_locks that shows all locks held by all processes. Postgres also has a function called pg_blocking_pids() that shows which processes are blocking a given process. Combine these two with pg_stat_activity and you can see what queries are causing locks and what queries are being affected by those locks.

Source

Conclusion

Locks are important in any database, I work SQL Server almost on daily basis and if row locks are enabled SQL Server can easily run out of memory and fail the transaction when row locks will be acquired. That is why I like the approach the Postgres row lock design. Nothing is free though, disk writes.

Page locks

A postgres page is 8Kb and stores tuples for table and indexes. Because the page is an in memory data structure, it needs to be protected from concurrent process accesses. You see, Postgres design is process based unlike other database, each process (client) connects to the database via a process based process, and those backend processes compete to access pages stored in shared buffer pool.

It is not as bad if multiple processes reading the same page, but it is a problem if one process is modifying a page while others are in the same page. You want to serialize those accesses so you don't corrupt data. This is classic operating system concepts with mutex and semaphores.

Dead Locks

Dead locks happen when two transactions each holding different locks attempting to access each other's resources and end up in an indefinite wait. Postgres detects dead locks and kills one of the transactions for more.

Here is an example (while unlikely it can happen)

Source: Postgres

```
Tx1
BEGIN;
-- ACQUIRES ExclusiveRowLock ON
UPDATE t SET foo=1 WHERE id=1;

Tx2
BEGIN;
-- ACQUIRES ExclusiveRowLock ON
UPDATE t SET foo=2 WHERE id=2;
-- ACQUIRES ON id=1
UPDATE t SET foo=2 WHERE id=1;
-- BLOCKED BY Tx1 FOR ROW LOCK ON id=1 WAITING TO Tx1
-- TO COMMIT OR ROLLBACK

-- ACQUIRES ON id=2 WAITING
UPDATE t SET foo=1 WHERE id=2;
-- BLOCKED BY Tx2 FOR ROW LOCK ON id=2 WAITING
-- TO COMMIT OR ROLLBACK
-- Tx1 ROLLBACK
-- Tx2 ROLLBACK
-- ERROR: ERROR: deadlock detected
-- DETAIL: Process 16384 waits for ShareLock on transaction 1; blocked by
-- Process 16385.
-- Process 16385 waits for ShareLock on transaction 2; blocked by
-- Process 16384.
-- HINT: See server log for query details.
-- CONTEXT: while updating tuple (0,1) in relation "t"
TRUNCATE TABLE TEST;
```

Page Locks

A page lock is a lock on a page, it locks the page and allows other processes to obtain other unique tuples.

Source

Tuple Locks

Tuple locks are used to lock a tuple (row) and allow other processes to obtain other tuples in the same page. Tuple locks are stored in the tuple header (xmax system field) and are persisted to disk. Tuple locks are used to serialize access to a tuple, for example if two transactions are updating the same tuple the second one will wait for the first one to finish.

Source

Lock Monitoring

Postgres has a system table called pg_locks that shows all locks held by all processes. Postgres also has a function called pg_blocking_pids() that shows which processes are blocking a given process. Combine these two with pg_stat_activity and you can see what queries are causing locks and what queries are being affected by those locks.

Source

Conclusion

Locks are important in any database, I work SQL Server almost on daily basis and if row locks are enabled SQL Server can easily run out of memory and fail the transaction when row locks will be acquired. That is why I like the approach the Postgres row lock design. Nothing is free though, disk writes.

Another example with row locks

```
Tx1
BEGIN;
-- ACQUIRES ShareRowLock (OK)
SELECT * FROM TEST
WHERE ID = 1
FOR SHARE
                                    Tx2
                                     BEGIN;
                                    -- ACQUIRES ShareRowLock (OK)
                                    SELECT * FROM TEST
                                    WHERE ID = 1
                                    FOR SHARE

                                    --Attempts to update the row (X)
                                    --blocked by Tx1 share lock
                                    UPDATE TEST SET V = 1
                                    WHERE ID = 1;

--Attempts to delete the row
--Blocked by Tx2 share lock
DELETE FROM TEST
WHERE ID = 1;

---DEAD LOCK X_X
```

Another dead lock example from the doc.

```
Tx1
BEGIN;
-- ACQUIRES FOR UPDATE lock
-- ON row 11111 (OK)
UPDATE accounts
SET balance = balance + 100.00
WHERE acctnum = 11111;
                                        Tx2
                                        BEGIN;
                                        -- ACQUIRES FOR UPDATE lock (OK)
                                        -- ON row 22222 (OK)
                                        UPDATE accounts
                                        SET balance = balance + 100.00
                                        WHERE acctnum = 22222;

                                        -- Attempts to acquire FOR UPDATE
                                        -- ON row 11111, blocked by tx1 (X)
                                        UPDATE accounts
                                        SET balance = balance - 100.00
                                        WHERE acctnum = 11111;

--Attempts to acquire FOR UPDATE
--On row 22222, blocked by tx2 (X)
UPDATE accounts
SET balance = balance - 100.00
WHERE acctnum = 22222;

---DEAD LOCK X_X
```

Advisory Locks

Sometimes the application requirement makes the native MVCC locks insufficent. That is why most databases provide application-level locks that are controlled by the application to be held and released. While those locks still live in the database they are acquired and released by the application.

You might say why not just use FOR SHARE and FOR UPDATE to simulate that. The problem is you have to have a row to lock in those cases and you might be unnecessarily blocking other transactions from doing legitment modifications to that row. Advisory locks are obtained on integer values not rows or tables. Those numbers can come from columns of rows they don't have to be.

Another reason why row locks don't work is they are always tied to a transacation, if the transcation commits or rollsback the locks are gone and this is something your application might not want. Take for example a long — running operation in the app that does multiple database transactions, you want to prevent multiple users from running this long running operation concurrently. It is almost very hard to control that with just normal locks so what you can do is obtain a session level adversary lock when the operation starts so other users will attempt to also to obtain the same Advisory lock and get blocked.

There are two types of advisory locks, session and transcation. Session locks obtained with pg_advisory_lock () are kept for the length of the session (connection), while transaction advisory locks obtained with pg_advisory_xact_lock () are kept for the length of the current running transaction.

Here is an example.

```
-- Start Applicaiton Operation
-- Acquires a session lock
SELECT pg_advisory_lock(100);
--TxA1
BEGIN;
--DO WORK
COMMIT;
                                    -- Start Applicaiton Operation
                                    -- Attemps Acquires a session lock
                                    -- BLOCKS (X)
                                    SELECT pg_advisory_lock(100);


--TxA2
BEGIN;
--DO MORE WORK
COMMIT;
--Release session lock 100
SELECT pg_advisory_unlock(100);
-- End Applicaiton Operation
                                    -- TxB1, lock unblocks
                                    BEGIN;
                                    --DO MORE WORK
                                    COMMIT;
                                    -- TxB2

                                     BEGIN;
                                    --DO MORE WORK
                                    COMMIT;

                                    --Release session lock 100
                                    SELECT pg_advisory_unlock(100);
```

Weak Locks

6/21/2023 — I added this additional paragraph to mention weak locks, something I recently learned in Postgres. Postgres has weak locks, those are table locks that rarely conflicts , acquired by DMLs, they are mainly AccessShareLock, RowShareLock, RowExclusiveLock.

Because they are common, and weak, Postgres manages them through a fast path a data structure in the process as oppose through the normal lock manager. But it can't just use that data structure with no limit, it has a limit and that limit is 16 weak locks per backend process according to the constant FP_LOCK_SLOTS_PER_BACKEND which can't be changed unless you recompile postgres alas. If you don't know backend process == connection in postgres. So if your data model is heavily normalized OR you are using partitioning and your queries are scanning multiple partitioning in a long transaction watch out not to exceed that. otherwise you hit the lock manager and contention is created.

Postgres Weak Locks

```
/*
 * We allow a small number of "weak" relation locks (AccessShareLock,
 * RowShareLock, RowExclusiveLock) to be recorded in the PGPROC structure
 * rather than the main lock table.  This eases contention on the lock
 * manager LWLocks.  See storage/lmgr/README for additional details.
 */
#define     FP_LOCK_SLOTS_PER_BACKEND 16
```

Summary

Understanding Postgres locking will give you an edge to making better application design choices and better luck at trouble-shooting lock waiting, dead-locks and general latency when executing queries. Advisory locks can also be used to build interesting use cases that don't work with normal MVCC locks. Hope you enjoyed this post.

If you like this content, consider checking out my database course.