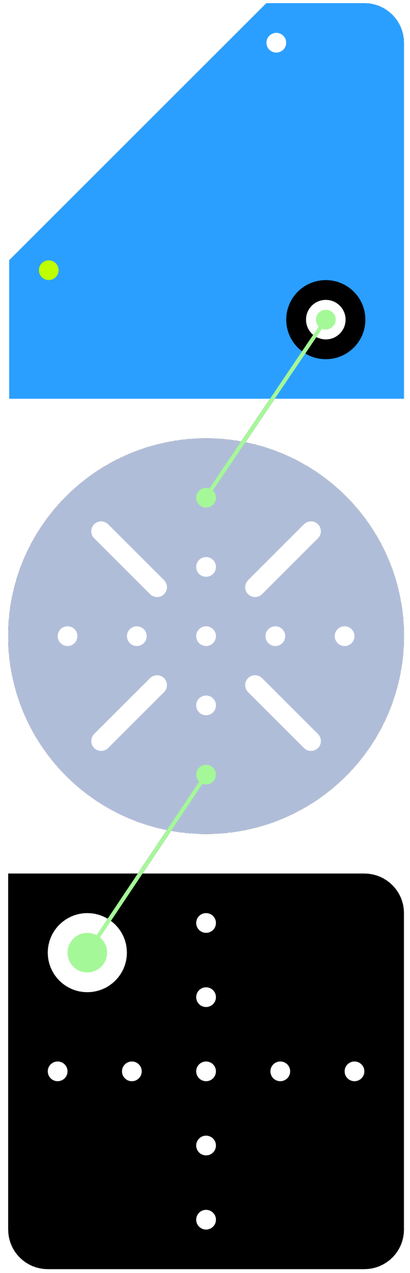
Yandex Cloud

The Edge of ACID with Injection Points



Andrey Borodin

Long waited for *injection_points_wait()*

Complex Systems Are Beyond Human Capacity To Evaluate

John Gall, Theorem of doubtful validity, ”Systemantica”

Fault injection framework

From: Asim R P <apraveen(at)pivotal(dot)io>
To: PostgreSQL mailing lists <pgsql-hackers(at)postgresql(dot)org>
Subject: Fault injection framework
Date: 2019-08-22 14:15:09
Message-ID: CANXE4TdxdESX1jKw48xet-5GvBFVSq=4cgNeioTQff372KO45A@mail.gmail.com
Views: Whole Thread | Raw Message | Download mbox | Resend email
Thread: 2019-08-22 14:15:09 from Asim R P <apraveen(at)pivotal(dot)io>
Lists: pgsql-hackers

```
Hello

Fault injection was discussed a few months ago at PGCon in Ottawa.  At
least a few folks showed interest and so I would like to present what we
have been using in Greenplum.

The attached patch set contains the fault injector framework ported to
PostgreSQL master.  It  provides ability to define points of interest in
backend code and then inject faults at those points from SQL.  Also
included is an isolation test to simulate a speculative insert conflict
scenario that was found to be rather cumbersome to implement using advisory
locks, see [1].  The alternative isolation spec using fault injectors seems
much simpler to understand.

Asim
```

POC: Better infrastructure for automated testing of concurrency issues

Lists:pgsql-hackers

From: Alexander Korotkov <aekorotkov(at)gmail(dot)com>
To: pgsql-hackers <pgsql-hackers(at)postgresql(dot)org>
Subject: POC: Better infrastructure for automated testing of concurrency issues
Date: 2020-11-25 14:10:54
Message-ID:CAPpHfdtSEOHX8dSk9Qp+Z++i4BGQoffKip6JDWngEA+g7Z-XmQ@mail.gmail.com
Views: Whole Thread | Raw Message | Download mbox | Resend email
Lists: pgsql-hackers

```
Hackers,

PostgreSQL is a complex multi-process system, and we are periodically faced
with complicated concurrency issues. While the postgres community does a
great job on investigating and fixing the problems, our ability to
reproduce concurrency issues in the source code test suites is limited.

I think we currently have two general ways to reproduce the concurrency
issues.
1. A text scenario for manual reproduction of the issue, which could
involve psql sessions, gdb sessions etc. Couple of examples are [1] and
[2]. This method provides reliable reproduction of concurrency issues. But
it's  hard to automate, because it requires external instrumentation
(debugger) and it's not stable in terms of postgres code changes (that is
particular line numbers for breakpoints could be changed). I think this is
why we currently don't have such scenarios among postgres test suites.
2. Another way is to reproduce the concurrency issue without directly
touching the database internals using pgbench or other way to simulate the
workload (see [3] for example). This way is easier to automate, because it
doesn't need external instrumentation and it's not so sensitive to source
code changes. But at the same time this way is not reliable and is
resource-consuming.
```

Injection points: some tools to wait and wake

Lists:pgsql-hackers

From: Michael Paquier <michael(at)paquier(dot)xyz>
To: Postgres hackers <pgsql-hackers(at)lists(dot)postgresql(dot)org>
Cc: Ashutosh Bapat <ashutosh(dot)bapat(at)gmail(dot)com>
Subject: Injection points: some tools to wait and wake
Date: 2024-02-19 06:01:40
Message-ID:ZdLuxBk5hGpol91B@paquier.xyz
Views: Whole Thread | Raw Message | Download mbox | Resend email
Lists: pgsql-hackers

```
Hi all,
(Ashutosh in CC as he was involved in the discussion last time.)

I have proposed on the original thread related to injection points to
have more stuff to be able to wait at an arbtrary point and wake at
will the process waiting so as it is possible to control the order of
actions taken in a test:
https://www.postgresql.org/message-id/ZTiV8tn_MIb_H2rE%40paquier.xyz

I didn't do that in the other thread out of time, but here is a patch
set to complete what I wanted, using a condition variable to wait and
wake processes:
- State is in shared memory, using a DSM tracked by the registry and
an integer counter.
- Callback to wait on a condition variable.
- SQL function to update the shared state and broadcast the update to
the condition variable.
- Use a custom wait event to track the wait in pg_stat_activity.
```

Other systems: MySQL \ InnoDB

open_tables(...)

DEBUG_SYNC(thd, "after_open_tables");

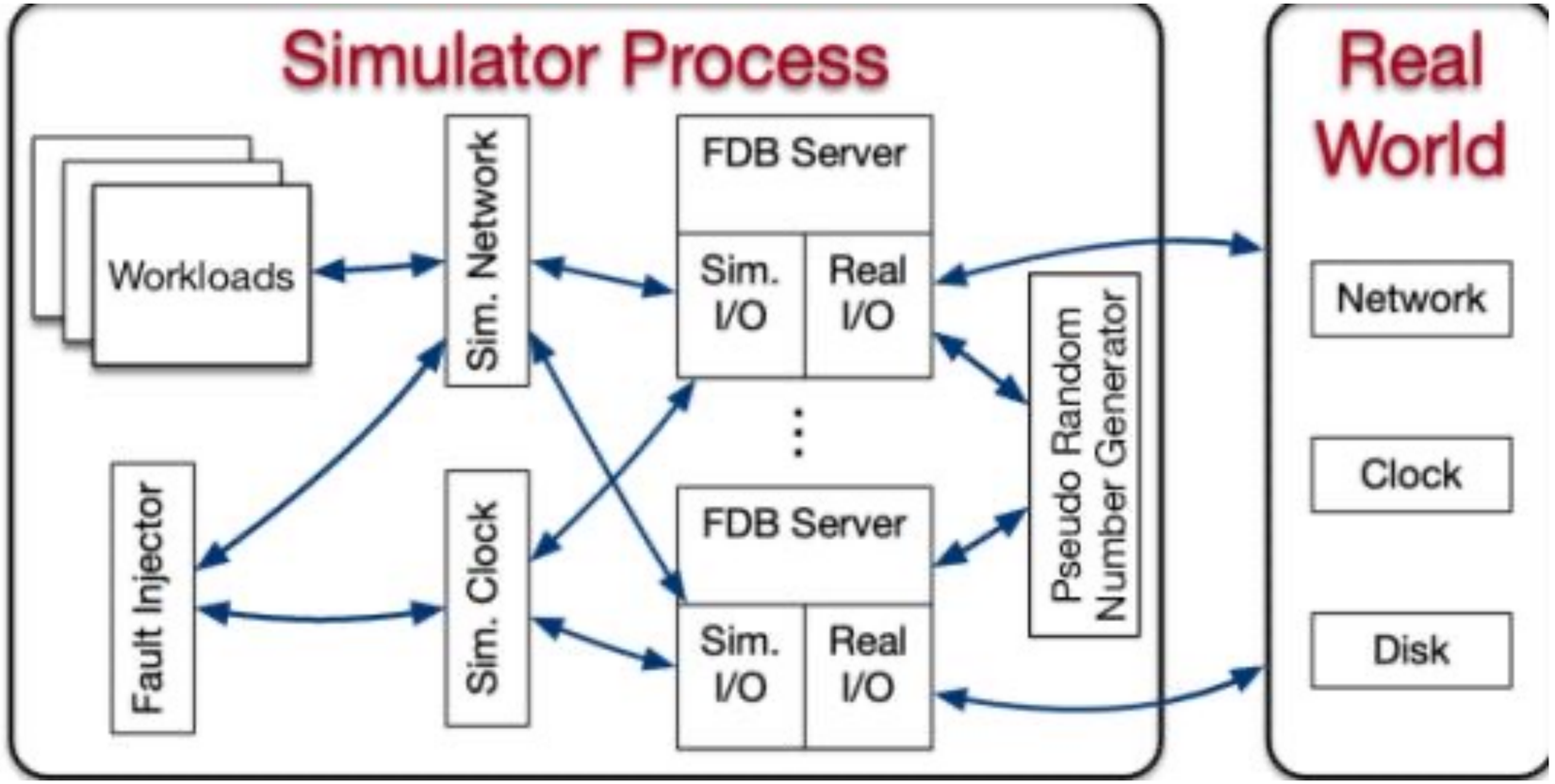
lock_tables(...)

DEBUG_SYNC allows named sync points with
SIGNAL / WAIT_FOR ;
DBUG (trace tool) can inject errors, sleep, print.
Widely used in MTR tests.
Very similar to PG injection points + wait/wake combined.

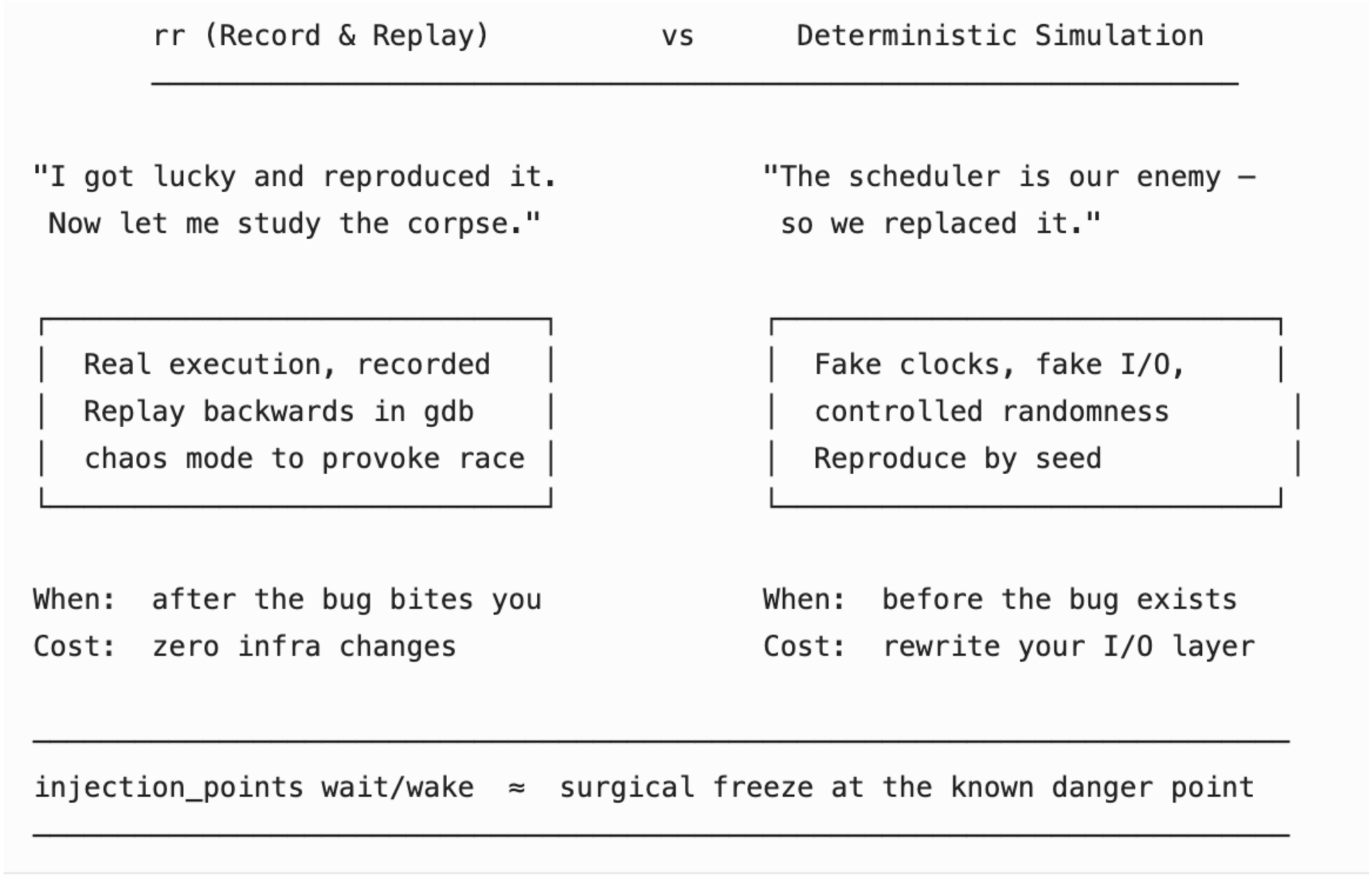
https://dev.mysql.com/doc/dev/mysql-server/9.6.0/PAGE_DEBUG_SYNC.html

Other systems: FoundationDB

Deterministic approach: replay using PRNG seed



https://pierrezemb.fr/posts/diving-into-foundationdb-simulation/


rr (Record & Replay)
vs
Deterministic Simulation
"I got lucky and reproduced it.
Now let me study the corpse."
"The scheduler is our enemy —
so we replaced it."
Real execution, recorded
Replay backwards in gdb
chaos mode to provoke race
Fake clocks, fake I/O,
controlled randomness
Reproduce by seed
When: after the bug bites you
Cost: zero infra changes
When: before the bug exists
Cost: rewrite your I/O layer
injection_points wait/wake ≈ surgical freeze at the known danger point

```
# NOT: "run load and watch for races"
# BUT: declare invariants, let the simulator find all schedules

def test_write_durability(sim):
    sim.run_workload([
        write_wal_record(xid=1),
        crash_at_random_point(),   # simulator picks the moment, not you
        recover(),
        assert_xid_committed(1),   # must hold under ANY schedule
    ])

# Run with every possible seed:
#   seed=1      → crash before fsync        → passes?
#   seed=2      → crash mid-write           → passes?
#   seed=3      → network partition + crash → passes?
#   ...
#   seed=1000000 → ...
```

Other systems

CockroachDB	`TestingKnobs` (Go interfaces)	Per-subsystem hook structs passed through code; tests inject callbacks before compile. Very powerful but tightly coupled to test code — not available in prod builds. Closer to Alexander Korotkov's approach in spirit.
TiDB	`pingcap/failpoint`	Port of FreeBSD kernel failpoints to Go. Failpoints are enabled/disabled at runtime via HTTP or env vars. Supports sleep, panic, return, print. Similar to MySQL DBUG but for Go.
FreeBSD kernel	`fail(9)` / `fail_point`	Original inspiration for most database fault injection. Named points with `fail_point_eval()` ; controlled via `sysctl` . Supports sleep, return, panic, break, print.
Linux kernel	fault-injection framework (`CONFIG_FAULT_INJECTION`)	Injects failures into `alloc_pages` , `kmalloc` , disk I/O. Probabilistic or nth-call triggering. Controlled via debugfs. No synchronization primitive — sleep-based only.
Cassandra / Java DBs	Byteman	JVM bytecode injection at runtime — can insert arbitrary code at any method entry/exit/line. Very powerful for JVM-based systems; no equivalent for C.

Tool of the ancient wizards

```
void
pg_usleep(long microsec)
{
    if (microsec > 0)
    {
#ifndef WIN32
        struct timespec delay;

        delay.tv_sec = microsec / 1000000L;
        delay.tv_nsec = (microsec % 1000000L) * 1000;
        (void) nanosleep(&delay, NULL);
#else
        SleepEx((microsec < 500 ? 1 : (microsec + 500) / 1000), FALSE);
#endif
    }
}
```

The transition

From:	Heikki Linnakangas <hlinnaka(at)iki(dot)fi>
To:	Tom Lane <tgl(at)sss(dot)pgh(dot)pa(dot)us>
Cc:	Noah Misch <noah(at)leadboat(dot)com>, Xiaoran Wang <fanfuxiaoran(at)gmail(dot)com>,
Subject:	Re: Recovering from detoast-related catcache invalidations
Date:	2024-12-14 00:06:53
Message-ID:	2234dc98-06fe-42ed-b5db-ac17384dc880@iki.fi
Views:	Whole Thread | Raw Message | Download mbox | Resend email
Lists:	pgsql-hackers

```
I was able to reproduce that, by pausing a process with gdb while it's
building the list in SearchCatCacheList():

1. Create a function called foofunc(integer). It must be large so that
its pg_proc tuple is toasted.

2. In one backend, run "SELECT foofunc(1)". It calls
FuncnameGetCandidates() which calls
"SearchSysCacheList1(PROCNAMEARGSNSP, CStringGetDatum(funcname));". Put
a break point in SearchCatCacheList() just after the systable_beginscan().

3. In another backend, create function foofunc() with no args.

4. continue execution from the breakpoint.

5. Run "SELECT foofunc()" in the first session. It fails to find the
function. The error persists, it will fail to find that function if you
try again, until the syscache is invalidated again for some reason.
```

The transition

From:	Heikki Linnakangas <hlinnaka(at)iki(dot)fi>
To:	Tom Lane <tgl(at)sss(dot)pgh(dot)pa(dot)us>
Cc:	Noah Misch <noah(at)leadboat(dot)com>, Xiaoran Wang <fanfuxiaoran(at)gmail(dot)com>,
Subject:	Re: Recovering from detoast-related catcache invalidations
Date:	2024-12-14 00:06:53
Message-ID:	2234dc98-06fe-42ed-b5db-ac17384dc880@iki.fi
Views:	Whole Thread | Raw Message | Download mbox | Resend email
Lists:	pgsql-hackers

I was able to reproduce that, by pausing a process with gdb while it's
building the list in SearchCatCacheList():

1. Create a function called foofunc(integer). It must be large so that
its pg_proc tuple is toasted.

2. In one backend, run "SELECT foofunc(1)". It calls
FuncnameGetCandidates() which calls
"SearchSysCacheList1(PROCNAMEARGSNSP, CStringGetDatum(funcname));". Put
a break point in SearchCatCacheList() just after the systable_beginscan().

3. In another backend, create function foofunc() with no args.

4. continue execution from the breakpoint.

5. Run "SELECT foofunc()" in the first session. It fails to find the
function. The error persists, it will fail to find that function if you
try again, until the syscache is invalidated again for some reason.

Attached is an injection point test case to reproduce that. If you
change the test so that the function's body is shorter, so that it's not
toasted, the test passes.

Test tools

Regression tests

Isolation tests

TAP tests

Injection points tests (with wait and wake)

```c
/* ── PLACING A POINT IN PRODUCTION CODE ──────────────────────────────────
 *
 * One macro call — zero cost when built without --enable-injection-points.
 * Compiles to ((void) name) in production builds.
 */
INJECTION_POINT("create-restart-point", NULL);

/* The optional arg can carry runtime context to the callback: */
INJECTION_POINT("my-point", (void *) &my_struct);
```

```c
/* ── PRELOAD + CACHED: avoid shmem lookup inside a critical section ──────────
 *
 * INJECTION_POINT_LOAD() caches the callback in process-local memory.
 * INJECTION_POINT_CACHED() uses that cache — no shared memory access,
 * safe to call even close to a critical section (but still not inside one).
 */

/* Load the injection point BEFORE entering the critical section */
INJECTION_POINT_LOAD("multixact-create-from-members");

multi = GetNewMultiXactId(nmembers, &offset);  /* starts START_CRIT_SECTION */

/* Fire from local cache — no palloc, no shmem lookup */
INJECTION_POINT_CACHED("multixact-create-from-members", NULL);
```

```
/* ── GUARD: skip work when nothing is attached ──────────────────────────
 *
 * Useful when the injection point body is expensive to prepare.
 */
if (IS_INJECTION_POINT_ATTACHED("my-expensive-point"))
{
    PrepareExpensiveDebugState();
    INJECTION_POINT("my-expensive-point", NULL);
}
```

```
# ── BASIC ACTIONS ──────────────────────────────────────────────────────

# 'wait'   – suspend the process with a condition variable
# 'error'  – raise ERROR at the point
# 'panic'  – raise PANIC at the point (force crash recovery)
SELECT injection_points_attach('my-point', 'wait');
SELECT injection_points_attach('my-point', 'error');
SELECT injection_points_attach('my-point', 'panic');

# ── SCOPING: process-local attachment ──────────────────────────────────

# Without this, injection points are global (visible to all backends).
# With this, the point is owned by the current backend and auto-detaches
# on disconnect — safe for concurrent TAP tests.
SELECT injection_points_set_local();
SELECT injection_points_attach('my-point', 'wait');

# ── OBSERVE ────────────────────────────────────────────────────────────

# Standard: poll pg_stat_activity
$node->wait_for_event('backend_type', 'my-point');
# which checks:
#   wait_event_type = 'InjectionPoint' AND wait_event = 'my-point'
```

```
# 1. ATTACH — register a named wait point in shared memory
$node_standby->safe_psql('postgres',
    "SELECT injection_points_attach('create-restart-point', 'wait');");

# 2. TRIGGER — start the target operation in the background;
#    it will block when it hits the injection point
my $psql_session =
  $node_standby->background_psql('postgres', on_error_stop => 0);
$psql_session->query_until(qr/starting_checkpoint/, q(
   \echo starting_checkpoint
   CHECKPOINT;
));

# 3. OBSERVE — wait until the process is suspended at the point
#    (pg_stat_activity: wait_event_type='InjectionPoint')
$node_standby->wait_for_event('checkpointer', 'create-restart-point');

# 4. ACT — do whatever needs to happen while the process is frozen
$node_primary->stop;
$node_standby->promote;

# 5. WAKE — release the suspended process to continue
$node_standby->safe_psql('postgres',
    "SELECT injection_points_wakeup('create-restart-point');");
```

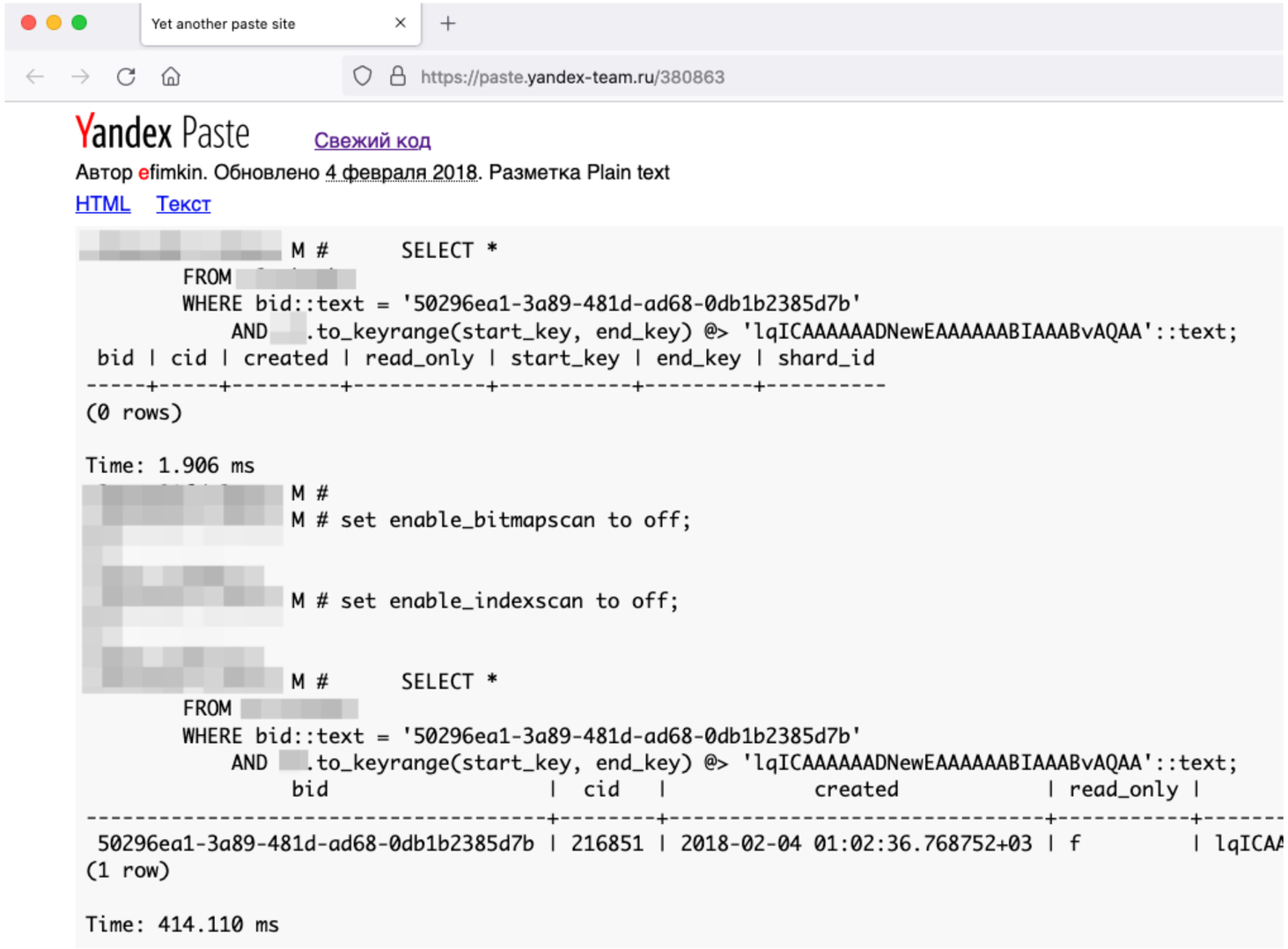
```
# ── WAKE ──────────────────────────────────────────────────────────────────

# Wake all processes waiting at the point (CV broadcast)
SELECT injection_points_wakeup('my-point');

# ── DETACH ────────────────────────────────────────────────────────────────

# Remove the callback — processes already waiting are still released by wakeup.
# Best practice: detach before wakeup so no new arrivals get caught.
SELECT injection_points_detach('my-point');
SELECT injection_points_wakeup('my-point');

# ── INSPECT ───────────────────────────────────────────────────────────────

# List all currently attached points
SELECT * FROM injection_points_list();
--  point_name       | library          | function
--  my-point         | injection_points | injection_wait
```

CASE STUDY 1:

before injection points

case 2PC vs CIC

February 2018: first encounter

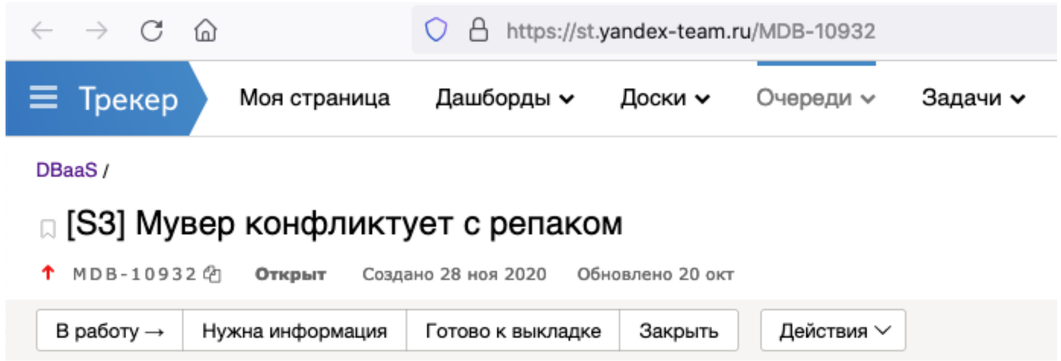


November 2020: S3 metadata repack at risk

pg_repack --index is essentially REINDEX

We were doing it cron job every week

> **Saved up to 20% of disk space**



December 2020: reproduction

Андрей Бородин 16 дек 2020

Наблюадю воспроизведение
С таблицей

```
postgres=# \d t1
                Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 i      | integer |           |          |
Indexes:
    "i1" btree (i)
```

Запускаю заполнение из Go

```
func doInserts(start int) {
        ctx := context.Background()
        conn, err := pgx.Connect(ctx, "host=127.0.0.1 port=5432 user=x4mmm database=postgres")
        pgConn := conn.PgConn()

        for i := start; ; i++ {
                is := fmt.Sprintf("%d", i)
                err = pgConn.Exec(ctx, fmt.Sprintf("begin;"+
                        "insert into t1 values(%d);"+
                        " PREPARE TRANSACTION 'x"+is+"'; ",
                        rand.Int31())).Close()
                exec := pgConn.Exec(ctx, " COMMIT PREPARED 'x"+is+"';")
        }
```

Запускаю бенч с CREATE INDEX CONCURRENTLY

```
drop index if exists i1; create index concurrently i1 on t1(i); select bt_index_check('i1',true);
```

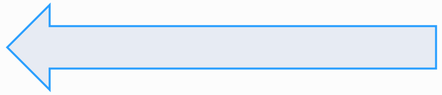
Через несколько секунд появляется индекс в котором не хватает данных.

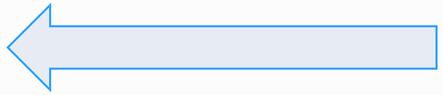
```
postgres=# select bt_index_check('i1',true);
NOTICE:  heap tuple (277,212) from table "t1" lacks matching index tuple within index "i1"
HINT:  Retrying verification using the function bt_index_parent_check() might provide a more specific error.
```

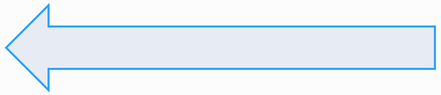
Go program simulates workload

AM check verifies invariants

```
BEGIN;
INSERT INTO t1 VALUES (1);
PREPARE TRANSACTION 'x';
-- XID moves to 2PC dummy PGPROC.
-- That PGPROC had backendId = InvalidBackendId, lxid = (LocalTransactionId) xid
-- so CIC's GetLockConflicts() returned a VXID that no longer existed
-- → CIC didn't wait, proceeded without the 2PC xact in its snapshot

CREATE INDEX CONCURRENTLY i1 ON t1(i);
-- missed the tuple inserted by 'x'

COMMIT PREPARED 'x';

SELECT bt_index_check('i1', true);
-- NOTICE: heap tuple lacks matching index tuple
```







January 2021: Fix for 2PC

Home / Commitfest 2021-01 / CREATE INDEX CONCURRENTLY does not index prepared xact's data

CREATE INDEX CONCURRENTLY does not index prepared xact's data

Edit | Comment/Review ▾ | Change Status ▾

Title	CREATE INDEX CONCURRENTLY does not index prepared xact's data
Topic	Bug Fixes
Created	2020-12-20 18:13:07
Last modified	2021-01-30 16:04:49 (8 months, 3 weeks ago)
Latest email	2021-01-30 16:06:56 (8 months, 3 weeks ago)
Status	2021-01: Committed
Target version	
Authors	Andrey Borodin (x4m)
Reviewers	Noah Misch (nmisch)
Committer	Noah Misch (nmisch)
Links	

April 2021: Bug resurrection

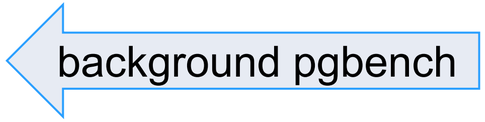
From:	Andrey Borodin <x4mmm(at)yandex-team(dot)ru>
To:	Noah Misch <noah(at)leadboat(dot)com>
Cc:	Michael Paquier <michael(at)paquier(dot)xyz>, Tom Lane <tgl(at)sss(dot)pgh(dot)pa(dot)us>, pgsql-bugs(at)lists(dot)postgresql(dot)org
Subject:	Re: CREATE INDEX CONCURRENTLY does not index prepared xact's data
Date:	2021-05-01 12:42:25
Message-ID:	9C782C9D-D53E-4154-B910-46F062A9AE9B@yandex-team.ru
Views:	Raw Message | Whole Thread | Download mbox | Resend email
Thread:	2021-05-01 12:42:25 from Andrey Borodin <x4mmm(at)yandex-team(dot)ru>
Lists:	pgsql-bugs

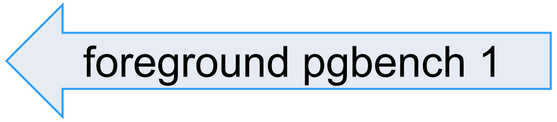
```
> 30 янв. 2021 г., в 21:06, Andrey Borodin <x4mmm(at)yandex-team(dot)ru> написал(а):
>
>
>
>> 24 янв. 2021 г., в 07:27, Noah Misch <noah(at)leadboat(dot)com> написал(а):
>>
>> I changed that, updated comments, and fixed pgindent damage.  I plan to push
>> the attached version.
>
> I see that patch was pushed. I'll flip status of CF entry. Many thanks!

FWIW I have 2 new reported cases on 12.6. I've double-checked that at the moment of corruption installation run version with the patch.
To the date I could not reproduce the problem myself, but I'll continue working on this.

Thanks!

Best regards, Andrey Borodin.
```

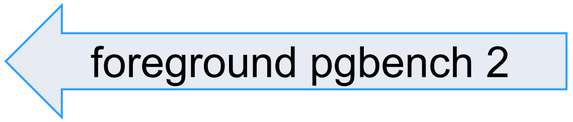
```
/* PrepareTransaction() — original order: */

AtPrepare_Locks();              /* (1) fast-path → regular locks, XID still in PGPROC */

ProcArrayClearTransaction(); /* (2) XID zeroed out in PGPROC
                              *           ↑
                              *    RACE WINDOW:
                              *    CIC's GetOldestXmin() runs here.
                              *    Our XID is gone → tuple looks committed.
                              *    CIC decides to index it — but hasn't yet. */

PostPrepare_Locks();            /* (3) locks moved to 2PC structure — too late */
```

https://github.com/postgres/postgres/commit/c776952049b

```
# Run background pgbench with CIC. We cannot mix-in this script into single pgbench:
# CIC will deadlock with itself occasionally.
my $pgbench_in  = '';
my $pgbench_out = '';
my $pgbench_timer = IPC::Run::timeout(180);
my $pgbench_h = $node->background_pgbench('postgres', \$pgbench_in, \$pgbench_out, $pgbench_timer,
    {
        '002_pgbench_concurrent_cic' =>
          q(
            REINDEX INDEX CONCURRENTLY idx;
            SELECT bt_index_check('idx',true);
           )
    },
    '--no-vacuum --client=1 --transactions=200');

# Run pgbench.
$node->pgbench(
    '--no-vacuum --client=5 --transactions=200',
    0,
    [qr{actually processed}],
    [qr{^$}],
    'concurrent transactions',
    {
        '002_pgbench_concurrent_transaction' =>
          q(
            BEGIN;
            SELECT pg_sleep(0.001);
            INSERT INTO tbl VALUES(0);
            COMMIT;
          ),
        '002_pgbench_concurrent_transaction_savepoints' =>
          q(
            BEGIN;
            SELECT pg_sleep(0.001);
            SAVEPOINT s1;
            SELECT pg_sleep(0.001);
            INSERT INTO tbl VALUES(0);
            COMMIT;
          )
    });
```







https://github.com/postgres/postgres/commit/c776952049b

The most complex bug I ever worked on

- Reproduced in production once in 3 month with 1 MRPS avg workload
- 3 Yandex services reported suspicious behavior from Feb 2018
- 9 month of active investigation knowing where to search and what to fix
- Parts of the fix committed 4 times, 6 reviewers
- Tests triggered kernel bugs in 2 OSes

February 2022: To be continued on Standby

PG Bug reporting form 9 February 2022, 02:01

BUG #17401: REINDEX TABLE CONCURRENTLY creates a race condition on a streaming replica

To: PostgreSQL mailing lists, Cc: Ben Chobot Details

The following bug has been logged on the website:

Bug reference: 17401
Logged by: Ben Chobot
Email address: bench@silentmedia.com
PostgreSQL version: 12.9
Operating system: Linux (Ubuntu)
Description:

This bug is is almost identical to BUG #17389, which I filed blaming pg_repack; however, further testing shows the same symptoms using vanilla REINDEX TABLE CONCURRENTLY.

CASE STUDY 2:

MultiXact "edge case 2"

```
WAL stream (in order of logging, not assignment):

  ┌──────────────────────────────────────────────┐
  │ CREATE_ID  mxid=101  offset=500  nmembers=3  │
  └──────────────────────────────────────────────┘
      offsets[101] = 500
      offsets[102] = 0

      mxid=101 readable:  members[500 .. offsets[102]-1]
        offsets[102]=0 means "I am the latest" → corner case 1
        read until latestOffset                              ✓

  ┌──────────────────────────────────────────────┐
  │ CREATE_ID  mxid=103  offset=509  nmembers=2  │
  └──────────────────────────────────────────────┘
      offsets[103] = 509
      offsets[102] = 0   ← still unknown

      mxid=101 UNREADABLE now:
        offsets[102]=0, but mxid=103 already exists
        → 101 is NOT the latest anymore        ← corner case 2 UNREADABLE until mxid=103 do pg_usleep(1000)
        → end boundary unknown
        → reader must wait for CREATE_ID mxid=102

  ┌──────────────────────────────────────────────┐
  │ CREATE_ID  mxid=102  offset=503  nmembers=6  │
  └──────────────────────────────────────────────┘
      offsets[102] = 503  ✓

      mxid=101 readable again:  members[500..502]          ✓
```

Sleep problem and the fix attempt

```
1420  1436          if (nextMXOffset == 0)
1421  1437          {
1422  1438              /* Corner case 2: next multixact is still being filled in */
1423  1439              LWLockRelease(lock);
1424  1440              CHECK_FOR_INTERRUPTS();
1425        -           pg_usleep(1000L);
      1441  +
      1442  +         ConditionVariableSleep(&MultiXactState->nextoff_cv,
      1443  +                                WAIT_EVENT_MULTIXACT_CREATION);
      1444  +         slept = true;
1426  1445              goto retry;
```

https://github.com/postgres/postgres/commit/a0e0fb1

Commit 768a9fd

 **alvherre** committed on Aug 20, 2024 · ✕ 0 / 8 · Verified

```
Add injection-point test for new multixact CV usage

Before commit a0e0fb1, multixact.c contained a case in the
multixact-read path where it would loop sleeping 1ms each time until
another multixact-create path completed, which was uncovered by any
tests.  That commit changed the code to rely on a condition variable
instead.  Add a test now, which relies on injection points and "loading"
thereof (because of it being in a critical section), per commit
4b21100.

Author: Andrey Borodin <x4mmm@yandex-team.ru>
Reviewed-by: Michaël Paquier <michael@paquier.xyz>
Discussion: https://postgr.es/m/0925F9A9-4D53-4B27-A87E-3D83A757B0E0@yandex-team.ru
```

master · REL_18_0 ••• REL_18_BETA1

Re: MultiXact\SLRU buffers configuration

From:	Thom Brown <thom(at)linux(dot)com>
To:	Michael Paquier <michael(at)paquier(dot)xyz>
	Alvaro Herrera <alvherre(at)alvh(dot)no-ip(dot)org>,
Cc:	Freund <andres(at)anarazel(dot)de>, Thomas Munro
	<horikyota(dot)ntt(at)gmail(dot)com>, pgsql-hackers
Subject:	Re: MultiXact\SLRU buffers configuration
Date:	2024-10-29 17:45:02

```
I believe I am seeing the problem being discussed occuring on a production
system running 15.6, causing ever-increasing replay lag on the standby,
until I cancel the offending process on the standby and force it to process
its interrupts.

Here's the backtrace before I do that:

#0  0x00007f4503b81876 in select () from /lib64/libc.so.6
#1  0x0000558b0956891a in pg_usleep (microsec=microsec(at)entry=1000) at
pgsleep.c:56
#2  0x0000558b0917e01a in GetMultiXactIdMembers (from_pgupgrade=false,
onlyLock=<optimized out>, members=0x7ffcd2a9f1e0, multi=109187502) at
multixact.c:1392
```

This occurred twice, meaning 2 processes needed terminating.

From:	"Andrey M(dot) Borodin" <x4mmm(at)yandex-team(dot)ru>
To:	Thom Brown <thom(at)linux(dot)com>
Cc:	Michael Paquier <michael(at)paquier(dot)xyz>, Alvaro Herrera <alvherre(at)alvh(dot)no-ip(dot)org>, vignesh C <vignesh21(at)gmail(dot)com>, Andrew Borodin <amborodin86(at)gmail(dot)com>, Yura Sokolov <y(dot)sokolov(at)postgrespro(dot)ru>, Andres Freund <andres(at)anarazel(dot)de>, Thomas Munro <thomas(dot)munro(at)gmail(dot)com>, Gilles Darold <gilles(at)darold(dot)net>, Alexander Korotkov <aekorotkov(at)gmail(dot)com>, Daniel Gustafsson <daniel(at)yesql(dot)se>, Kyotaro Horiguchi <horikyota(dot)ntt(at)gmail(dot)com>, pgsql-hackers mailing list <pgsql-hackers(at)postgresql(dot)org>
Subject:	Re: MultiXact\SLRU buffers configuration
Date:	2024-11-05 14:11:07
Message-ID:	D48EEBE4-0BD8-4503-8738-1E9E069A3970@yandex-team.ru
Views:	Whole Thread | Raw Message | Download mbox | Resend email
Lists:	pgsql-hackers

> On 1 Nov 2024, at 00:25, Thom Brown <thom(at)linux(dot)com> wrote:
>
> Unfortunately I didn't gather much information when it was occuring, and prioritised getting rid of the process blocking replay. I just attached gdb to it, got a backtrace and then "print ProcessInterrupts()".
>

Currently I do not see how this wait can result in a deadlock.
But I did observe standby in a pathological sequential scan encountering recent multixact again and again (new each time).
I hope this situation will be alleviated by recent cahnges - now there is not a millisecond wait, but hopefully smaller amount of time.

In v17 we also added injection point test which reproduces reading very recent multixact. If there is a deadlock I hope buildfarm will show it to us.

Best regards, Andrey Borodin.

Found in pgsql-hackers Mailbox

Dmitry 25 June 2025 at 11:11

IPC/MultixactCreation on the Standby server

To: pgsql-hackers@lists.postgresql.org

Hi, hackers

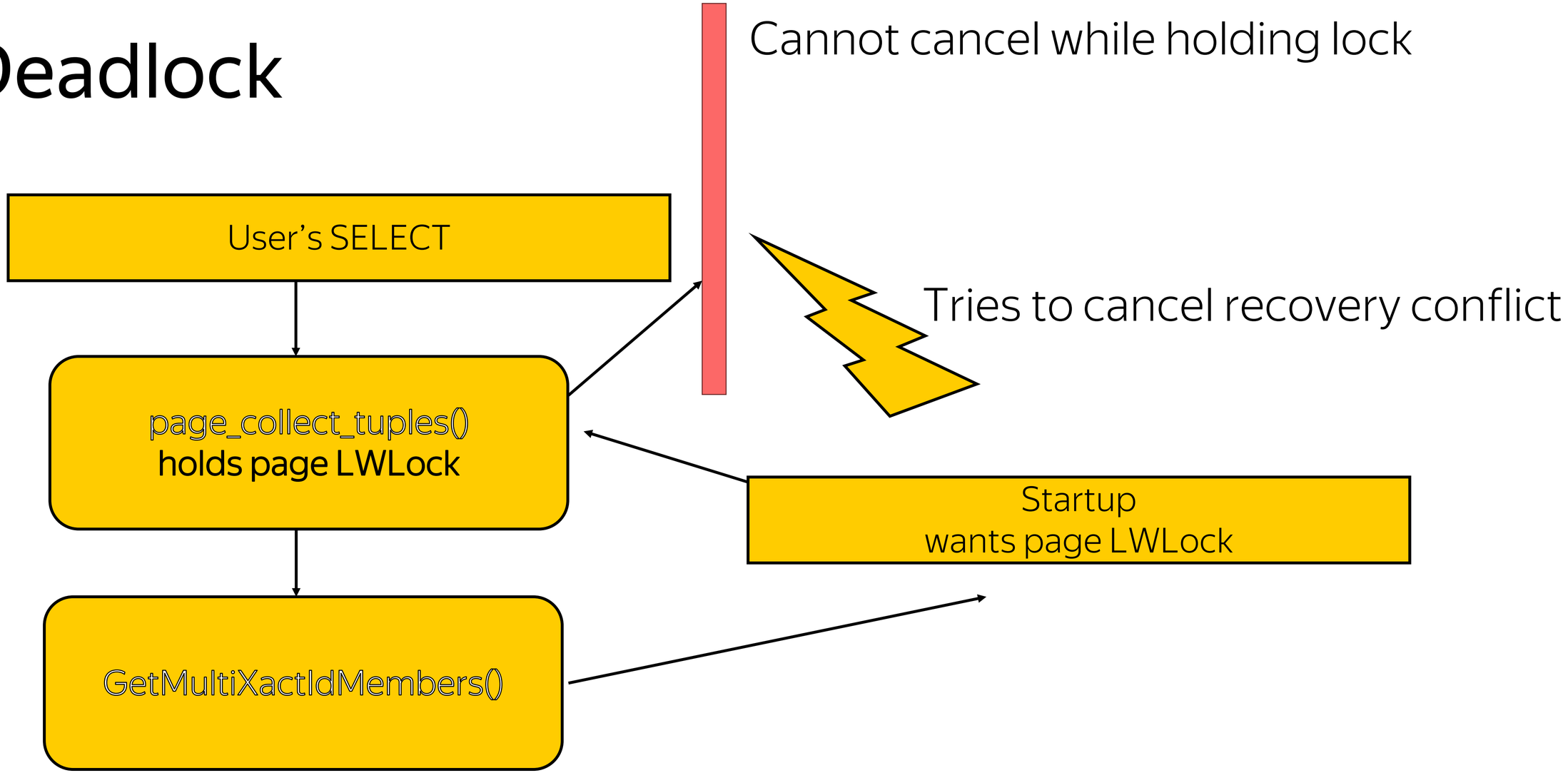
The problem is as follows.
A replication cluster includes a primary server and one hot-standby replica.
The workload on the primary server is represented by multiple requests generating multixact IDs,
while the hot-standby replica performs reading requests.

After some time, all requests on the hot-standby are stuck and never get finished.

The `pg_stat_activity` view on the replica reports that processes are stuck waiting for
IPC/MultixactCreation,
pg_cancel_backend and pg_terminate_backend cannot cancel the request, SIGQUIT is the only way to
stop it.

Here is an example with a synthetic workload reproducing the problem.

Deadlock


Cannot cancel while holding lock
User’s SELECT
page_collect_tuples()
holds page LWLock
Tries to cancel recovery conflict
Startup
wants page LWLock
GetMultiXactIdMembers()

```
WAL stream (in order of logging, not assignment):

  ┌──────────────────────────────────────────────┐
  │ CREATE_ID  mxid=101  offset=500  nmembers=3  │
  └──────────────────────────────────────────────┘
      offsets[101] = 500
      offsets[102] = 0

      mxid=101 readable:  members[500 .. offsets[102]-1]
        offsets[102]=0 means "I am the latest" → corner case 1
        read until latestOffset                              ✓

  ┌──────────────────────────────────────────────┐
  │ CREATE_ID  mxid=103  offset=509  nmembers=2  │
  └──────────────────────────────────────────────┘
      offsets[103] = 509
      offsets[102] = 0   ← still unknown

      mxid=101 UNREADABLE now:
        offsets[102]=0, but mxid=103 already exists
        → 101 is NOT the latest anymore        ← corner case 2 UNREADABLE until mxid=103 do pg_usleep(1000)
        → end boundary unknown
        → reader must wait for CREATE_ID mxid=102

  ┌──────────────────────────────────────────────┐
  │                                              │
  └──────────────────────────────────────────────┘
      offsets[102] = 503  ✓

      mxid=101 readable again:  members[500..502]          ✓
```

CRASH HERE!

```
1480  1507          if (nextMXOffset == 0)
1481  1508          {
1482        -           /* Corner case 2: next multixact is still being filled in */
1483        -           LWLockRelease(lock);
1484        -           CHECK_FOR_INTERRUPTS();
1485        -
1486        -           INJECTION_POINT("multixact-get-members-cv-sleep", NULL);
1487        -
1488        -           ConditionVariableSleep(&MultiXactState->nextoff_cv,
1489        -                                  WAIT_EVENT_MULTIXACT_CREATION);
1490        -           slept = true;
1491        -           goto retry;
      1509  +           ereport(ERROR,
      1510  +                   (errcode(ERRCODE_DATA_CORRUPTED),
      1511  +                    errmsg("MultiXact %d has invalid next offset",
      1512  +                           multi)));
1492  1513          }
```

CASE STUDY 3:

measurement changes the system

VM corruption on standby

From:	Andrey Borodin <x4mmm(at)yandex-team(dot)ru>
To:	PostgreSQL Hackers <pgsql-hackers(at)lists(dot)postgresql(dot)org>, Melanie Plageman <melanieplageman(at)gmail(dot)com>
Subject:	VM corruption on standby
Date:	2025-08-06 14:59:58
Message-ID:	B3C69B86-7F82-4111-B97F-0005497BB745@yandex-team.ru
Views:	Whole Thread | Raw Message | Download mbox | Resend email
Thread:	2025-08-06 14:59:58 from Andrey Borodin <x4mmm(at)yandex-team(dot)ru>
Lists:	pgsql-hackers

```
Hi hackers!

I was reviewing the patch about removing xl_heap_visible and found the VM\WAL machinery very interesting.
At Yandex we had several incidents with corrupted VM and on pgconf.dev colleagues from AWS confirmed that they
saw something similar too.
So I toyed around and accidentally wrote a test that reproduces $subj.

I think the corruption happens as follows:
0. we create a table with one frozen tuple
1. next heap_insert() clears VM bit and hangs immediately, nothing was logged yet
2. VM buffer is flushed on disk with checkpointer or bgwriter
3. primary is killed with -9
now we have a page that is ALL_VISIBLE\ALL_FORZEN on standby, but clear VM bits on primary
4. subsequent insert does not set XLH_LOCK_ALL_FROZEN_CLEARED in it's WAL record
5. pg_visibility detects corruption
```

```
Idea: search for VM corruption bug
  PageIsAllVisible(page) && visibilitymap_clear() race
Plan:
  Place INJECTION_POINT inside critical section
  to freeze the process at the dangerous moment,
  then kill -9 the postmaster and check VM on recovery
```

```
            INJECTION_POINT("vm-clear-before-wal")
                        ↓
       injection_wait() fires
                        ↓
    ConditionVariableSleep()  ← inside critical section!
                        ↓
       WaitLatch(WL_EXIT_ON_PM_DEATH)
                        ↓
          postmaster killed
                        ↓
      WL_EXIT_ON_PM_DEATH fires
                        ↓
   proc_exit() called mid-critical-section
                        ↓
    shared memory written inconsistently
                        ↓
"VM corruption" found on recovery  ← not a real bug
                                     injection point itself
                                     caused the corruption
```

```c
/* injection_wait() — what actually happens: */

ConditionVariablePrepareToSleep(&inj_state->wait_point);

for (;;) {
    ConditionVariableSleep(          /* calls WaitLatch() internally */
        &inj_state->wait_point,
        injection_wait_event
    );
}

ConditionVariableCancelSleep();
```

Commit `c13070a`

 **akorotkov** committed on Aug 22, 2025 · ✓ 7 / 10

```
Revert "Get rid of WALBufMappingLock"

This reverts commit bc22dc0.
It appears that conditional variables are not suitable for use inside
critical sections.  If WaitLatch()/WaitEventSetWaitBlock() face postmaster
death, they exit, releasing all locks instead of PANIC.  In certain
situations, this leads to data corruption.

Reported-by: Andrey Borodin <x4mmm@yandex-team.ru>
Discussion: https://postgr.es/m/B3C69B86-7F82-4111-B97F-0005497BB745%40yandex-team.ru
Reviewed-by: Andrey Borodin <x4mmm@yandex-team.ru>
Reviewed-by: Aleksander Alekseev <aleksander@tigerdata.com>
Reviewed-by: Kirill Reshke <reshkekirill@gmail.com>
Reviewed-by: Tom Lane <tgl@sss.pgh.pa.us>
Reviewed-by: Thomas Munro <thomas.munro@gmail.com>
Reviewed-by: Tomas Vondra <tomas@vondra.me>
Reviewed-by: Andres Freund <andres@anarazel.de>
Reviewed-by: Yura Sokolov <y.sokolov@postgrespro.ru>
Reviewed-by: Michael Paquier <michael@paquier.xyz>
Backpatch-through: 18
```

`master`

3 files changed

Work in progress

- Build reproducer of curruption due to CondVar in buffer mapping wait

- Make injection point waits that can be used for such tests

CASE STUDY 4:

unobservable injection points

Re: "ERROR: latch already owned" on gharial

Lists:pgsql-hackers

From: Thomas Munro <thomas(dot)munro(at)gmail(dot)com>
To: pgsql-hackers <pgsql-hackers(at)postgresql(dot)org>
Cc: CM Team <cm(at)enterprisedb(dot)com>
Subject: "ERROR: latch already owned" on gharial
Date: 2022-05-25 00:45:21
Message-ID: CA+hUKGJ_0RGcr7oUNzcHdn7zHqHSB_wLSd3JyS2YC_DYB+-V=g@mail.gmail.com
Views: Whole Thread | Raw Message | Download mbox | Resend email
Lists: pgsql-hackers

Hi,

A couple of recent isolation test failures reported $SUBJECT.

It could be a bug in recent-ish latch refactoring work, though I don't
know why it would show up twice just recently.

Just BTW, that animal has shown signs of a flaky toolchain before[1].
I know we have quite a lot of museum exhibits in the 'farm, in terms
of hardare, OS, and tool chain. In some cases, they're probably just
forgotten/not on anyone's upgrade radar. If they've shown signs of
misbehaving, maybe it's time to figure out if they can be upgraded?
For example, it'd be nice to be able to rule out problems in GCC 4.6.0
(that's like running PostgreSQL 9.1.0, in terms of vintage,
unsupported status, and long list of missing bugfixes from the time
when it was supported).

[1] https://www.postgresql.org/message-id/CA+hUKGJK5R0S1LL_W4vEzKxNQGY_xGAQ1XknR-WN9jqQeQtB_w@mail.gmail.com

Re: "ERROR: latch already owned" on gharial

Lists:pgsql-hackers

From: Thomas Munro <thomas(dot)munro(at)gmail(dot)com>
To: pgsql-hackers <pgsql-hackers(at)postgresql(dot)org>
Cc: CM Team <cm(at)enterprisedb(dot)com>
Subject: "ERROR: latch already owned" on gharial
Date: 2022-05-25 00:45:21
Message-ID: CA+hUKGJ_0RGcr7oUNzcHdn7zHqHSB_wLSd3JyS2YC_DYB+-V=g@mail.gmail.com
Views: Whole Thread | Raw Message | Download mbox | Resend email
Lists: pgsql-hackers

```
Hi,

A couple of recent isolation test failures reported $SUBJECT.

It could be a bug in recent-ish latch refactoring work, though I don't
know why it would show up twice just recently.

Just BTW, that animal has shown signs of a flaky toolchain before[1].
I know we have quite a lot of museum exhibits in the 'farm, in terms
of hardare, OS, and tool chain.  In some cases, they're probably just
forgotten/not on anyone's upgrade radar.  If they've shown signs of
misbehaving, maybe it's time to figure out if they can be upgraded?
For example, it'd be nice to be able to rule out problems in GCC 4.6.0
(that's like running PostgreSQL 9.1.0, in terms of vintage,
unsupported status, and long list of missing bugfixes from the time
when it was supported)
```

+1, this is at least the third non-obvious miscompilation from gharial. Installing the latest GCC that builds easily (perhaps GCC 10.3) would make this a good buildfarm member again. If that won't happen, at least add a note to the animal like described in

[PATCH] Fix ProcKill lock-group vs procLatch recycle race

From: Vlad Lesin <vladlesin(at)gmail(dot)com>
To: PostgreSQL Hackers <pgsql-hackers(at)lists(dot)postgresql(dot)org>
Subject: [PATCH] Fix ProcKill lock-group vs procLatch recycle race
Date: 2026-04-27 08:14:59
Message-ID: d2983796-2603-41b7-a66e-fc8489ddb954@gmail.com


```
Hello all,

The following are the patches and demonstration material for a
concurrency bug in ProcKill() where a lock-group leader and a member can
exit in parallel.

------------------------------------------------------------------------
Problem
------------------------------------------------------------------------

If a leader detaches from the lock group under leader_lwlock but
has not yet reached DisownLatch(&MyProc->procLatch), a concurrent
last follower can still put the *leader* PGPROC on a free list, or
the leader and the follower can make inconsistent decisions about
*who* returns which PGPROC, so that a slot is linked into the free
list with procLatch still owned, or is pushed twice.  A new backend
that recycles the slot can then hit:

      PANIC: latch already owned by PID ...

A concrete interleaving (lock group leader vs last member)
is the following(PG15 code).

=== Lock group leader ===
L1: LWLockAcquire(leader_lwlock)
L2: dlist_delete(&MyProc->lockGroupLink)
      The list contains follower.
L3: dlist_is_empty → false (the follower still in)
L4: else if (leader != MyProc) → false, do nothing
L5: LWLockRelease(leader_lwlock)
      *lwlock released*

       >>>WINDOW OPEN HERE<<<

L6: SwitchBackToLocalLatch()
L7: pgstat_reset_wait_event_storage()
L8: proc = MyProc; MyProc = NULL;
L9: DisownLatch(&proc->procLatch)
      *only here owner_pid = 0*
L10: SpinLockAcquire(ProcStructLock)
L11: if (proc->lockGroupLeader == NULL) → false, skip
L12: SpinLockRelease(ProcStructLock)
============================
```

```
ProcKill() — original order:

  SwitchBackToLocalLatch()
  pgstat_reset_wait_event_storage()
  ...lock-group block...

  [follower pushes leader's PGPROC
   onto freelist here]
  ...
  DisownLatch(&proc->procLatch)

                  ↑
        too late: PGPROC already
        on freelist before Disown
```

```
Why it's a problem:

  ┌──────────────────────────────────┐
  │ PGPROC on freelist               │
  │ procLatch.owner_pid != 0         │
  └──────────────────────────────────┘

  ↓
new backend picks up the slot
OwnLatch(&proc->procLatch)

PANIC: latch already owned by PID N
```

```
Standard observability fails:

  -- both return nothing at this point
  SELECT * FROM pg_stat_activity WHERE pid = $leader_pid;
  SELECT pg_terminate_backend($leader_pid);  -- already gone


Solution: read PGPROC->wait_event_info directly from ProcGlobal->allProcs

  -- custom helper, bypasses ProcArray
  SELECT prockill_backend_in_injection($leader_pid,
                                       'prockill-after-lockgroup-leader');
  -- reads allProcs[procno].wait_event_info
  -- remains valid until proc->pid = 0 near the end of ProcKill
```

- **Injection points have SQL interface**

- **Many interesting cases happen when SQL is not available**

My wishlist

Injection point waits suitable for kill -9 towards a postmaster or anything else

- I’m mostly interested in corruption investigations

Injection point waits suitable for kill -9 towards a postmaster or anything else

- I’m mostly interested in corruption investigations

File system controls of injections points

- Or another observability without SQL

Injection point waits suitable for kill -9 towards a postmaster or anything else

- I’m mostly interested in corruption investigations

File system controls of injections points

- Or another observability without SQL

injection_points_wait()

- Polling *pg_stat_activity* takes to much time

Injection point waits are a last resort — use them only when the race window is too difficult to hit any other way

- Test takes tens of milliseconds
- Injection points require *--enable-injection-points*

- Don't codify a race - eliminate it. The best test for a fixed race is one you no longer need.

Questions?

Andrey Borodin

Open-source RDBMS development team lead

Cloud