PostgreSQL Entangled in Locks:
Attempts to free it

PGCon 2017
26.05.2017

- ***Amit Kapila***
- ***Dilip Kumar***

Overview

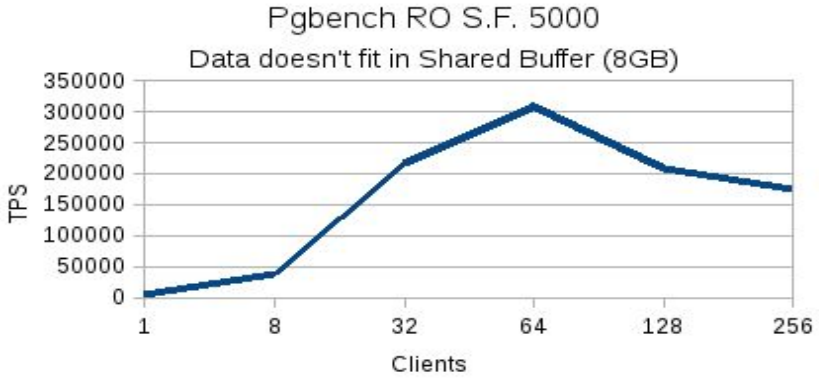
- Background
  - Effects of locking on scalability
  - Past approaches
- Existing issues in lock contention
  - Read-only bottlenecks
    - C-Hash for Buffer mapping lock
    - Snapshot caching for ProcArrayLock
  - Read-write Bottlenecks
    - WAL write lock
    - Clog control lock

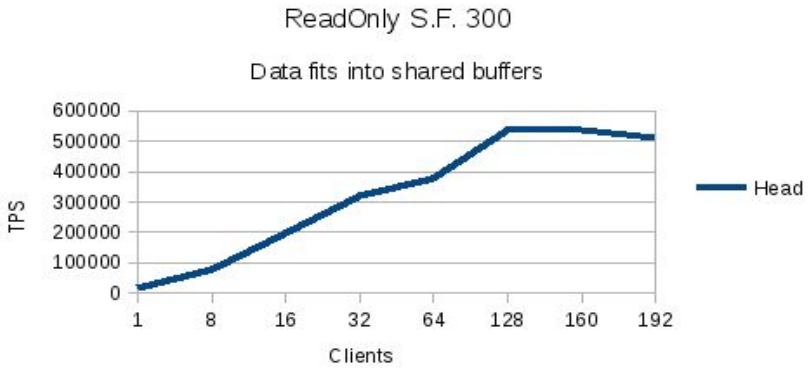
Effects of locking

- Locks have been a major scalability issue in PostgreSQL.
- In the past a lot of work has been done to remove these bottlenecks.
  - Fastpath relation lock.
  - Change locking regimen around buffer replacement.
  - Lockless StrategyGetBuffer in clock sweep.
  - ProcArrayLock contention removal by group commit.
  - Buffer header spin lock to atomic operation.
  - Hash Header Lock Contention.

Empirical evidence for read-only bottlenecks(1/3)

- Performance on Intel 8 socket machine (128 core)
  - Performance Data on Commit f58b664393dcfd02c2f57b3ff20fc0aee6dfebf1.
  - If data doesn't fit in shared buffers, performance is falling after 64 clients.
  - When data fits in shared buffers, it's falling after 128 clients.





Empirical evidence for read-only bottlenecks(2/3)

- Experimental Setup:
  - Wait event test
  - Workload: pgbench readonly
  - Hardware: Intel 8 socket machine (128 core with HT)
  - Scale Factor : 5000
  - Run duration : 120 s

Wait Event Count	Wait Event Type	Wait Event Name
39919	*LWLock*	*buffer_mapping*
5119	*Client*	*ClientRead*
3116	IO	*DataFileRead*
558	*Activity*	*WalWriterMain*

Empirical evidence for read-only bottlenecks(3/3)

- Experimental setup:
  - Test with perf
  - Workload: pgbench readonly
  - Hardware: Intel 8 socket machine (128 core with HT)
  - Scale Factor : 300
  - Run duration : 120 s

```
+   18.08%     0.00%  postgres  [unknown]            [.] 0x0000023c0000023c
+   18.08%     0.00%  postgres  [unknown]            [.] 0x08ec834853f58948
+   14.88%    14.55%  postgres  postgres             [.] GetSnapshotData
+    8.91%     0.00%  postgres  [unknown]            [.] 0x7fffffff7fffffff
+    8.84%     8.60%  postgres  postgres             [.] LWLockAttemptLock
+    7.77%     0.00%  postgres  [unknown]            [.] 0000000000000000
+    6.35%     0.00%  postgres  [unknown]            [.] 0x0000000000004011
+    6.35%     6.16%  postgres  postgres             [.] LWLockRelease
+    6.21%     0.00%  postgres  [unknown]            [k] 0x0000000000030000
+    5.96%     0.23%  postgres  libpthread-2.17.so   [.] __libc_send
+    5.17%     0.02%  postgres  [kernel.kallsyms]    [k] system_call_fastpath
+    4.86%     4.69%  postgres  postgres             [.] PinBuffer
+    4.05%     0.02%  postgres  [kernel.kallsyms]    [k] sys_sendto
+    4.02%     0.18%  postgres  [kernel.kallsyms]    [k] SYSC_sendto
+    3.64%     0.02%  postgres  [kernel.kallsyms]    [k] sock_sendmsg
+    3.44%     0.25%  postgres  [kernel.kallsyms]    [k] unix_stream_sendmsg
+    2.68%     2.55%  postgres  postgres             [.] _bt_compare
+    2.59%     2.53%  postgres  postgres             [.] hash_search_with_hash_value
+    2.48%     0.02%  postgres  [kernel.kallsyms]    [k] apic_timer_interrupt
```

Analysis for Bottlenecks in read-only(1/3)

- Wait event test shows that there is significant bottleneck on the BufferMappingLock.
  - especially when data doesn't fit into shared buffers.

- There is also significant bottleneck in GetSnapshotData.
  - Perf shows significant time spent in taking the snapshot.

Analysis for Bottlenecks in read-only(2/3)

- BufferMappingLock
  - This is used to protect the shared buffer mapping hash table.
  - We acquire it in exclusive lock mode to insert an entry for the new block during buffer replacement and in shared mode to find the existing buffer entry.
  - This lock is partitioned for concurrency.

Analysis for Bottlenecks in read-only(3/3)

- GetSnapshotData
  - In read-committed transaction isolation, we need to take the snapshot for every query.
  - There is an extra overhead in computing snapshot every time.
  - There is contention on ProcArrayLock as we compute snapshot under that lock.

Experiments for reducing read-only bottleneck.

- C-Hash (Concurrent hash table) for removing buffer mapping lock contention.

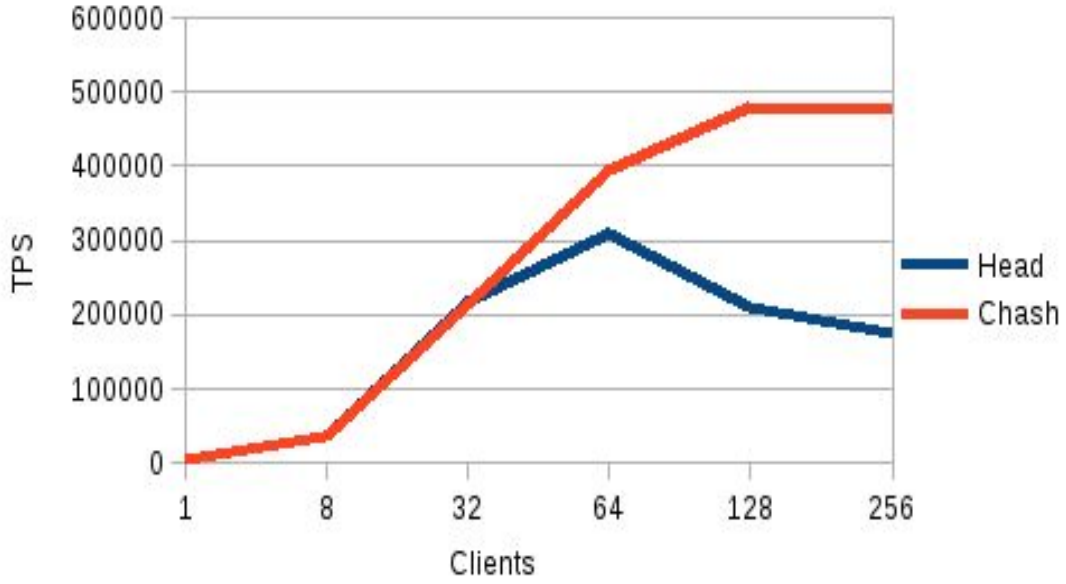
- Snapshot caching for reducing the overhead of GetSnapshotData.

C-Hash

- Lock-free hash table which works using memory barriers and atomic operations.
- Lookups are done without any locks, only memory barriers.
- Inserts and deletes are done using atomic ops.
- Delete just mark the node as deleted but doesn't make it reusable immediately.
- When a backend wishes to move entries from a garbage list to a free list, it must first wait for any backend scanning that garbage list to complete their scans.

Experimental evaluation: results(1/2)

- Experimental setup
  - Pgbench read only test
  - Scale factor 5000
  - Shared buffers 8GB
  - Intel 8 socket machine



Experimental evaluation: results(2/2)

- Experimental setup
  - Wait Event Test on Intel 8 machine Socket.
  - Pgbench readonly test.
  - Scale Factor: 5000
  - Run duration: 120s

On Head		
Event Count	Event Type	Event Name
39919	*LWLock*	*buffer_mapping*
5119	*Client*	*ClientRead*
3116	*IO*	*DataFileRead*

With C-Hash		
Event Count	Event Type	Event Name
3102	*Client*	*ClientRead*
633	*IO*	*DataFileRead*
199	*Activity*	*WalWriterMain*

Experimental evaluation: conclusion

- After C-Hash patch it's scaling up to 128 clients and maximum gain of > 150% at higher clients.
- Wait event test shows that the contention on the buffer mapping lock is completely gone.
- There is 8-10% regression at one client.

Cache the snapshot (1/2)

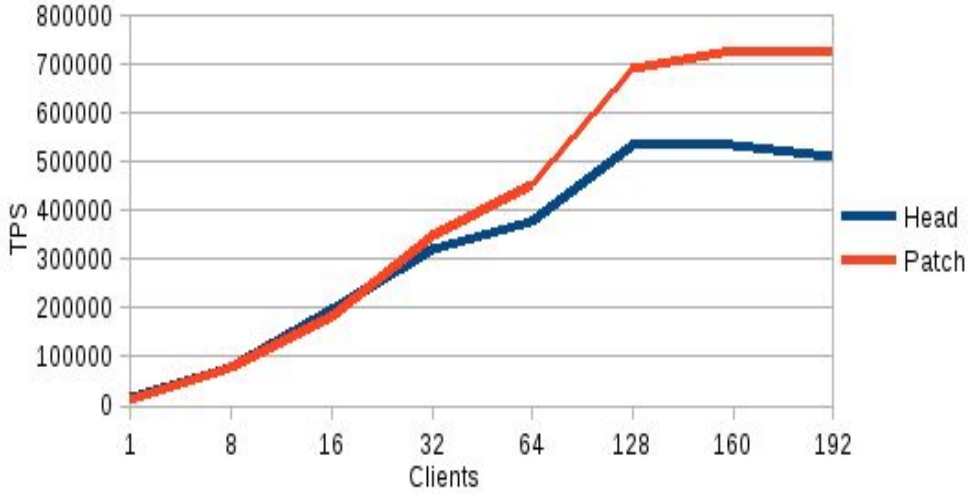
- Calculate the snapshot once and reuse it across the backends till it's valid.
- Once the snapshot is calculated cache it into the shared memory.
- Next time any backend tries to calculate the snapshot, first check the snapshot cache.

Cache the snapshot(2/2)

- If a valid snapshot is available, reuse it.
- Otherwise, calculate the new snapshot and store it into the cache.
- ProcArrayEndTransaction will invalidate the cached snapshot.

Experimental evaluation: results

- Experimental setup
  - Pgbench read only test
  - Scale factor 300
  - Shared buffers 8GB
  - Intel 8 socket machine

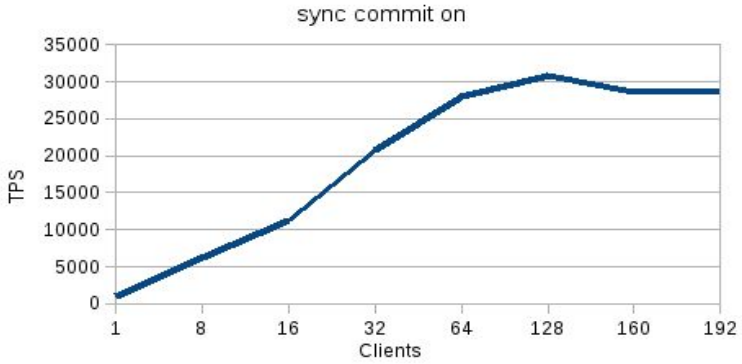


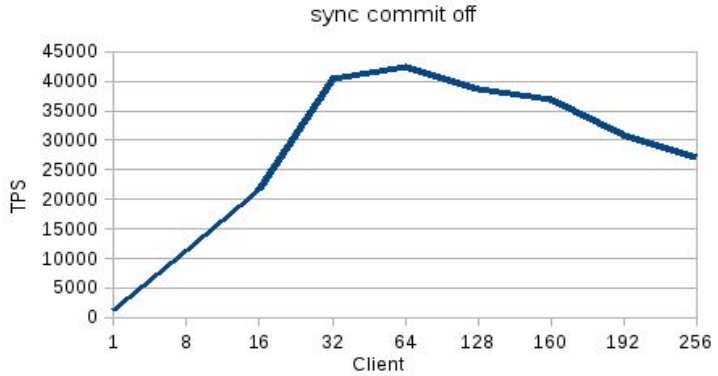
Experimental evaluation: conclusion

- After the patch, it can scale up to 128 clients.
- Observed >40% gain at higher client counts.
- Perf shows significant reduction in GetSnapshotData .

Empirical evidence for read-write bottlenecks(1/2)

- Experimental setup
  - Test on Intel 8 socket machine
  - Scale factor 300
  - Shared buffers 8GB
  - sync_commit=on and sync_commit=off





Empirical evidence for read-write bottlenecks(2/2)

- Wait event test at 128 clients.

sync_commit on		
11607	*IPC*	*ProcArrayGroupUpdate*
8477	*LWLock*	*WALWriteLock*
7996	*Client*	*ClientRead*
2671	*Lock*	*transactionid*
557	*LWLock*	*wal_insert*

sync_commit off		
10005	*IPC*	*ProcArrayGroupUpdate*
9430	*LWLock*	*CLogControlLock*
6352	*Client*	*ClientRead*
5480	*Lock*	*transactionid*
1368	*LWLock*	*wal_insert*

Analysis for Bottlenecks in read-write

- Wait event shows that ProcArrayGroupUpdate is on the top.
- And also shows significant contentions on WALWriteLock.
- With sync commit off, WALWriteLock is reduced and it shows the next contention on ClogControlLock.

WAL Write Lock(1/3)

- This lock is acquired to write and flush the WAL buffer data to disk.
  - during commit.
  - during writing dirty buffers, if WAL is already not flushed.
  - periodically by WALWriter.

- Generally, we observe very high contention around this lock during read-write workload.

WAL Write Lock(2/3)

- Experiments to find overhead
  - `Removed WAL write and flush calls.`
    - The TPS for read-write pgbench test is increased from 27871 to 45068 (at 300 scale factor with 64 clients).
  - `Tested with fsync off.`
    - The TPS for read-write pgbench test is increased from 27871 to 41835 (at 300 scale factor with 64 clients).

WAL Write Lock(3/3)

- Split WAL write and flush operations
  - Take WAL flush calls out of WALWriteLock and perform them under a new lock ("WALFlushLock").
  - This should allow simultaneous os writes when a fsync is in progress.
  - LWLockAcquireOrWait is used for the newly introduced WAL Flush Lock to accumulate flush calls.

Group flush the WAL(1/2)

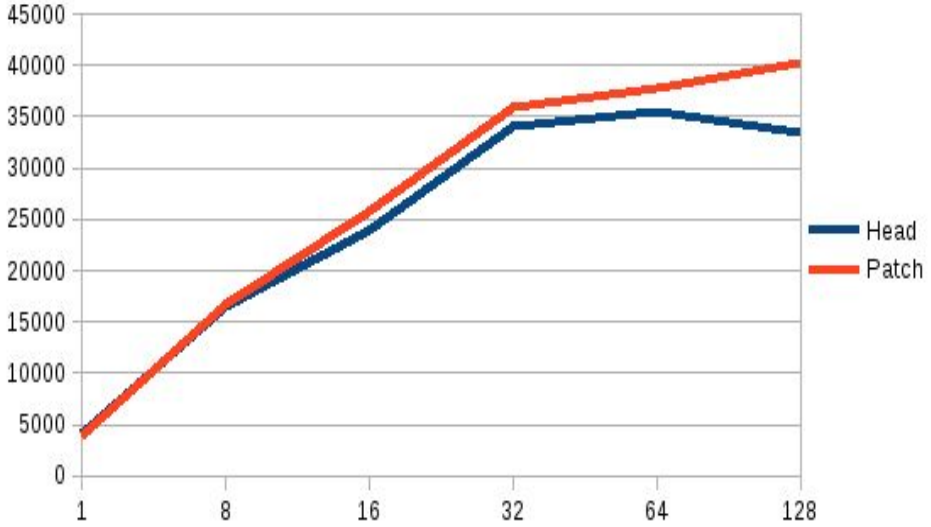
- Each proc will advertise its write location and add itself to the pending_flush_list.
- The first backend that sees the list as empty (leader), will proceed to flush the WAL for all the procs in the pending_flush_list.
- The leader backend will acquire the LWLock and traverse the pending_flush_list to find the highest write location to flush.

Group flush the WAL(2/2)

- The leader backend will flush the WAL up to highest noted write location.
- After flush, it wakes up all the procs for which it has flushed the WAL.

Experimental evaluation: results

- Experimental setup
  - Intel 2 socket machine (56 cores)
  - Pgbench Read-write test
  - Scale factor 1000
  - Sync_commit off
  - Shared buffer 14GB



Experimental evaluation: conclusion

- Combining both the approaches group flushing and separating lock shows some improvement (~15-20%) at higher client counts.
- Haven’t noticed a very big improvement with any of the approaches independently.
- Some more tests can be performed for larger WAL records to see the impact of combined writes.

Clog Control Lock(1/2)

- Wait event test shows huge contention around this lock at high client count (>64) especially with mixed workload.

- This lock is acquired
  - In exclusive mode to update the transaction status into the CLOG.
  - In exclusive mode to load the CLOG page into memory.
  - In shared mode to read the transaction status from the CLOG.

Clog Control Lock(2/2)

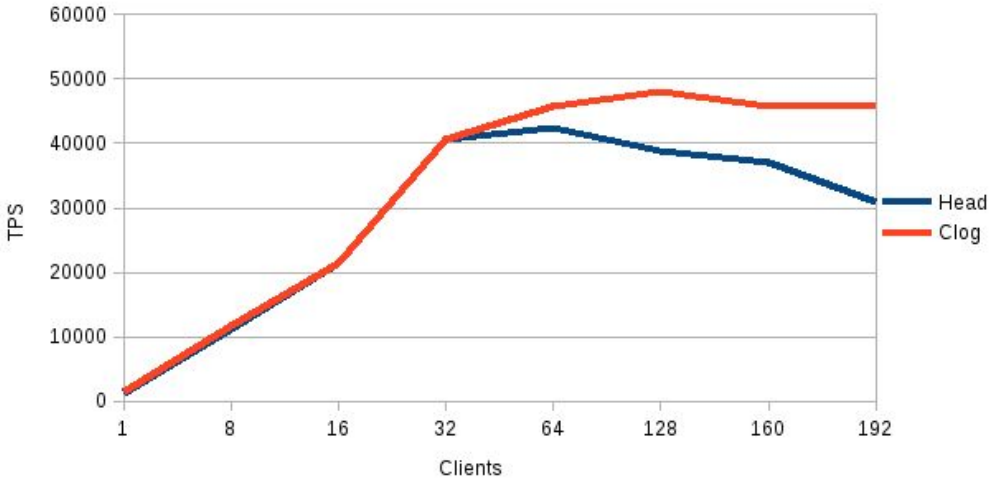
- The contention around this lock happens due to
  - Multiple processes contends with each other when simultaneously writing the transaction status in the CLOG.
  - Processes writing the transaction status in the CLOG contends with the processes reading the transaction status from the CLOG.
  - Both the above contentions together lead to high contention around CLOGControlLock in read-write workloads.

Group update the transaction status in Clog

- Solution used for ClogControlLock is similar to the ProcArray Group Clear XID.
- One backend will become the group leader and that process will be responsible for updating the transaction status for rest of the group members.
- This reduces the contention on the lock significantly.

Experimental evaluation: results(1/2)

- Experimental setup
  - Pgbench Read-write test
  - Scale factor 300
  - Sync_commit off
  - Shared buffer 8GB



Experimental evaluation: results(2/2)

- Wait event test at 128 clients sync commit off.

On Head		
Event Count	Event Type	Event Name
10005	*IPC*	*ProcArrayGroupUpdate*
9430	*LWLock*	*CLogControlLock*
6352	*Client*	*ClientRead*
5480	*Lock*	*transactionid*
1368	*LWLock*	*wal_insert*

With Group update Clog		
Event Count	Event Type	Event Name
18671	*IPC*	*ProcArrayGroupUpdate*
10014	*LWLock*	*ClientRead*
6115	*Client*	*transactionid*
2552	*Lock*	*wal_insert*
687	*LWLock*	*ProcArrayLock*

Experimental evaluation: conclusion

- On the head, performance is falling after 64 clients whereas, with the patch, it can scale up to 128 clients.
- ~50% performance gain at higher clients.
- Wait event test shows significant reduction in contention on ClogControlLock.

Conclusion

- Pgbench test shows significant bottleneck exist in BuffermappingLock at higher scale factor for read-only workload.
  - C-Hash shows >150% gain at higher clients.
  - Snapshot caching shows >40% gain when data fits into shared buffers.
- Read-write test show bottleneck on *ProcArrayGroupUpdate, WALWriteLock and ClogControlLock.*
  - Group flush along with taking the WAL flush calls out of WALWriteLock shows ~20% gain.
  - Clog group update shows ~50% gain.

Thank You