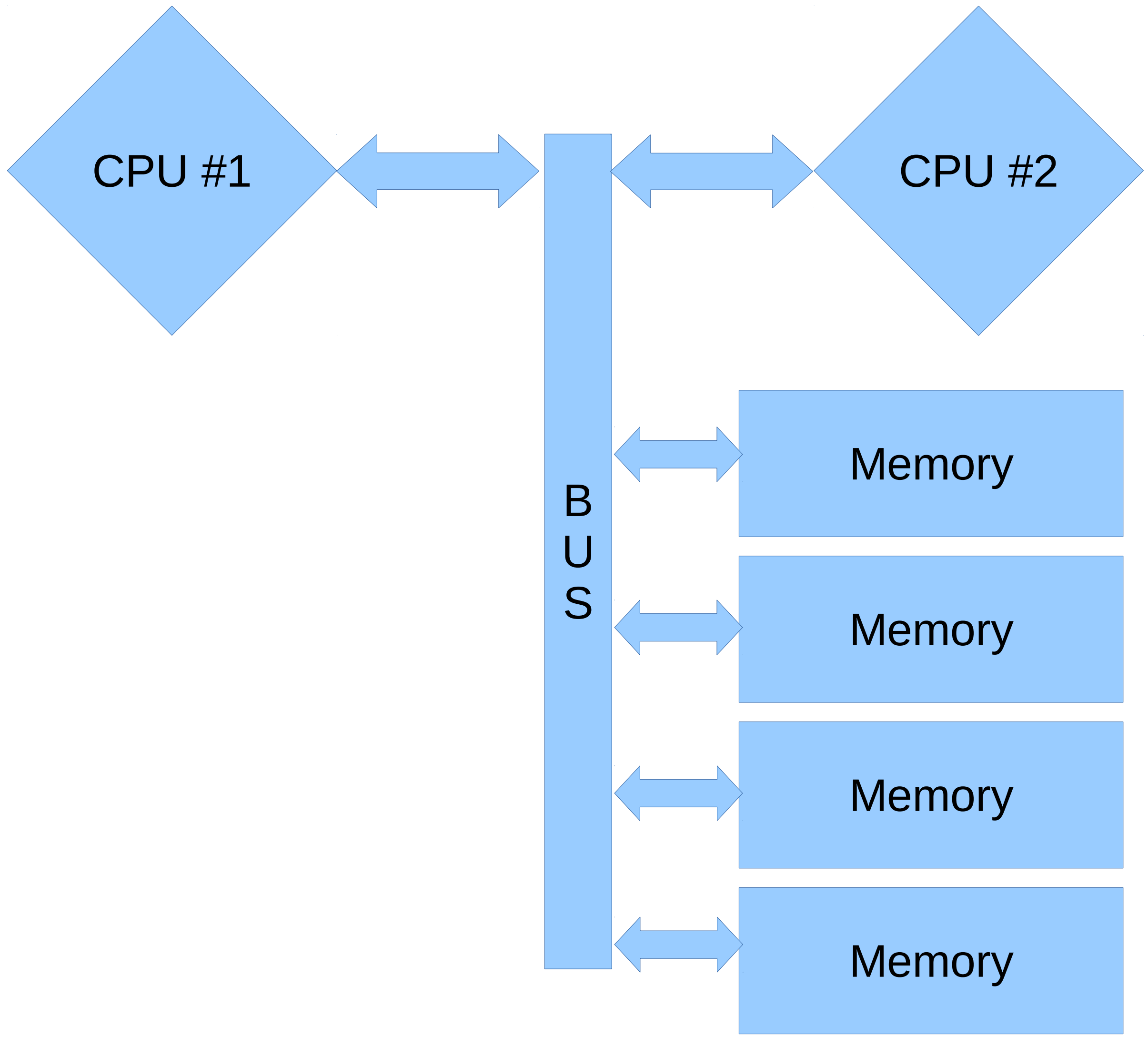
Improving Postgres' Concurrency

Andres Freund
PostgreSQL Developer & Committer
Citus Data – citusdata.com - @citusdata

Vertical Scalability

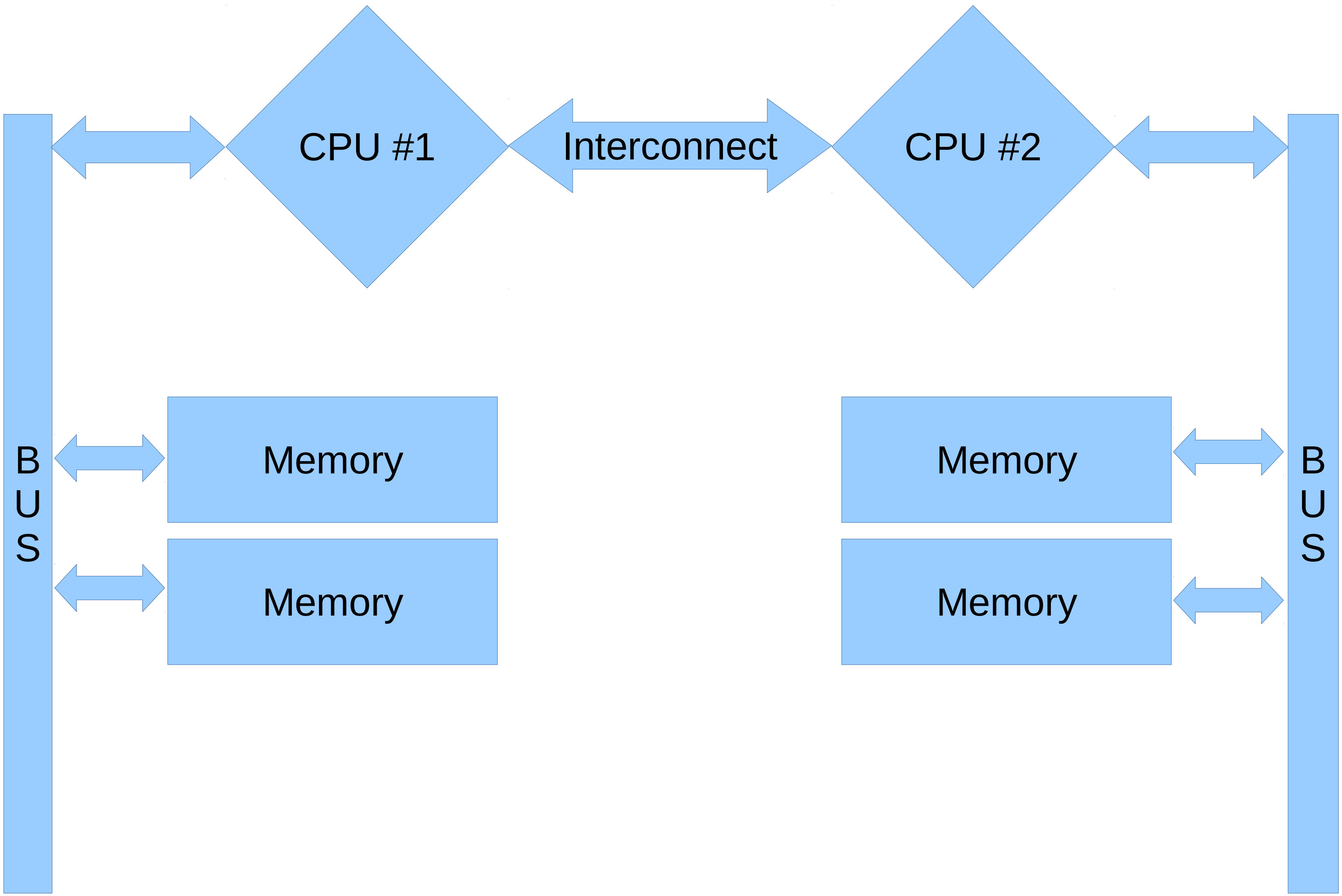
- Bigger Machines → Higher Throughput
- Multi-Core CPUs
  - 2005 - 2 cores
  - 2015 – 18 cores
- Multi-Socket Servers
- NUMA
- Cache Coherency
- Often cheaper to develop for
- Lower Latency / Higher Consistency

Uniform Memory Access



citusdata

Non-Uniform Memory Access



Postgres Locking Primer

- Spinlocks
  - fast (very short locks)
  - exclusive only
  - no queuing (super expensive if locks held too long)
  - no error recovery
  - no deadlock checks
  - fixed number

Postgres Locking Primer

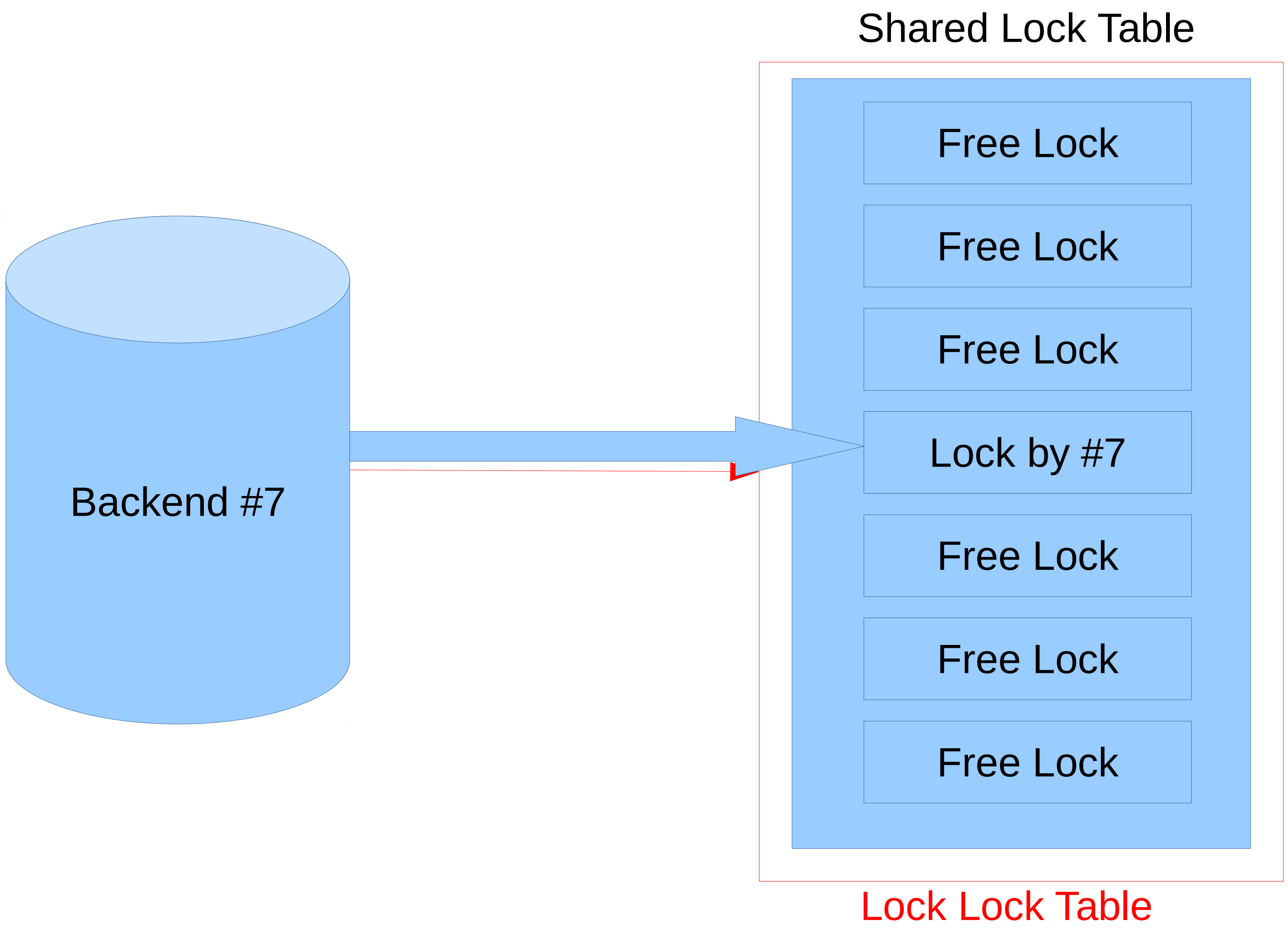
- LWLock
  - fast
  - reader/writer lock
  - error recovery
  - no deadlock checks
  - fixed number
  - uses spinlocks

citusdata

Postgres Locking Primer

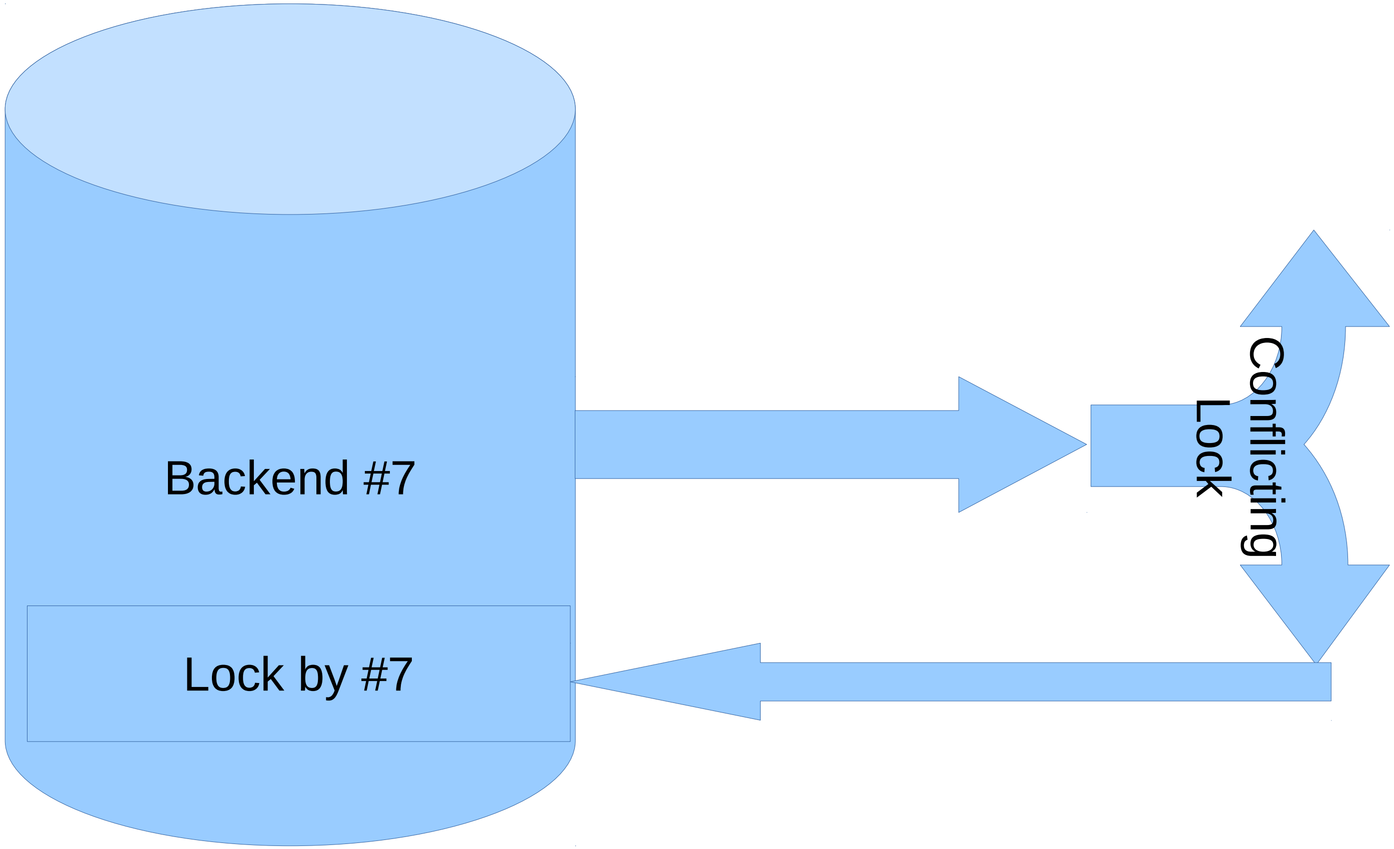
- Heavyweight Locks
  - complex locking modes
  - error recovery
  - deadlock checks
  - “dynamic” identities
  - uses LWLocks & spinlocks

Acquiring a Heavyweight Lock



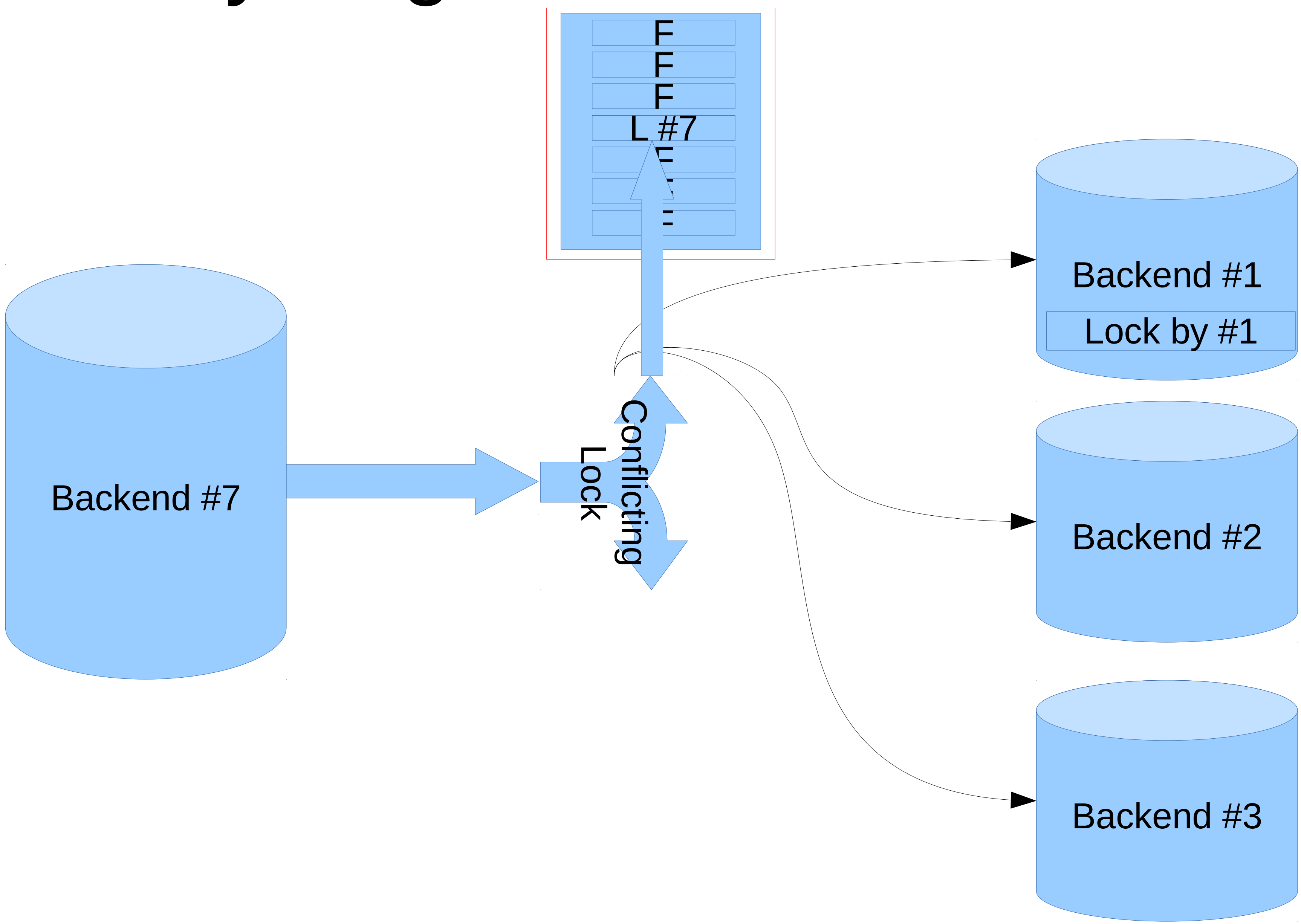
citusdata

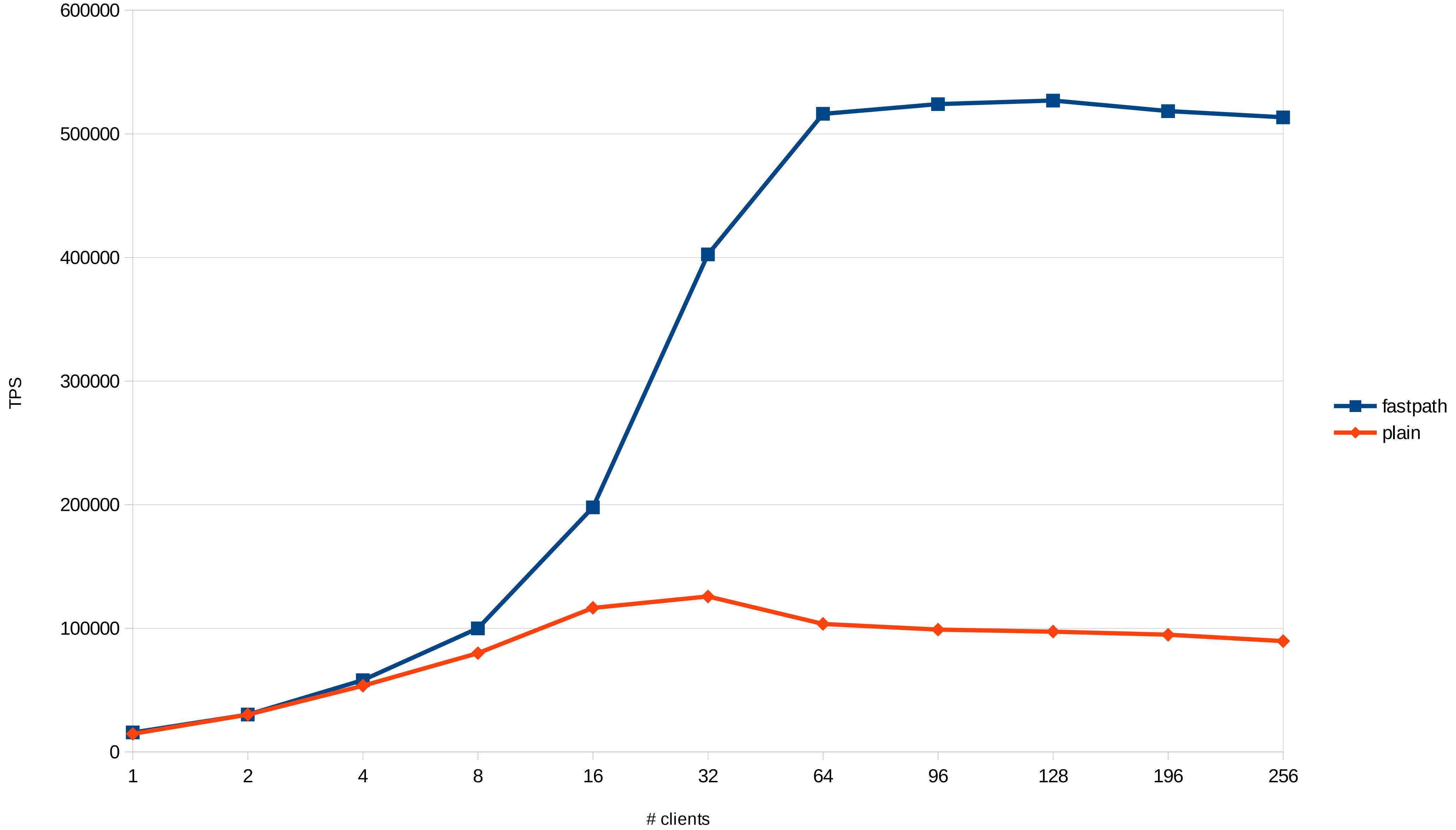
Heavyweight Lock - Fastpath



citusdata

Heavyweight Lock – Slow Path





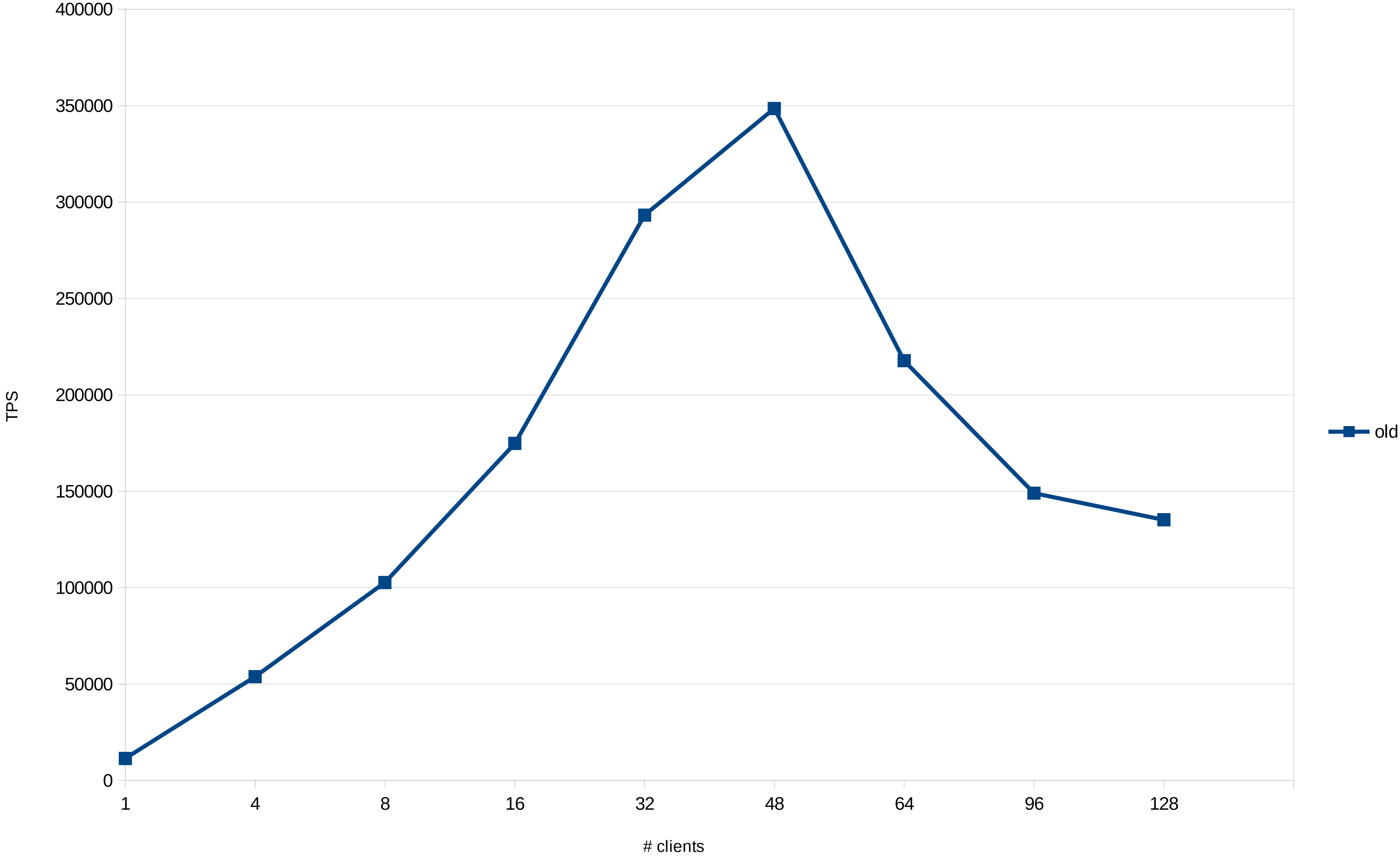
- readonly pgbench scale 300
- EC2 m4.8xlarge - 2 x E5-2676
- master @ aa6b2e6
- fastpath disabled in code

LWLock scalability

```
# perf top -az
     89.53%  postgres  postgres  [.] s_lock
      2.53%  postgres  postgres  [.] LWLockAcquire
      1.79%  postgres  postgres  [.] LWLockRelease
      0.63%  postgres  postgres  [.] hash_search_..._value
```

```
LWLockAcquire(LWLock *l, LWLockMode mode)
{
  retry:
    SpinLockAcquire(&lock->mutex);

    if (mode == LW_SHARED)
    {
        if (!lock->exclusive)
        {
            lock->shared++;
        }
        else
        {
            QueueSelf(l);
            SpinLockRelease(&lock->mutex);
            WaitForRelease(l);
            goto retry;
        }
    }
    ...
    SpinLockRelease(&lock->mutex);
}
```



- readonly pgbench
- 4xE5-4620
- scale 100

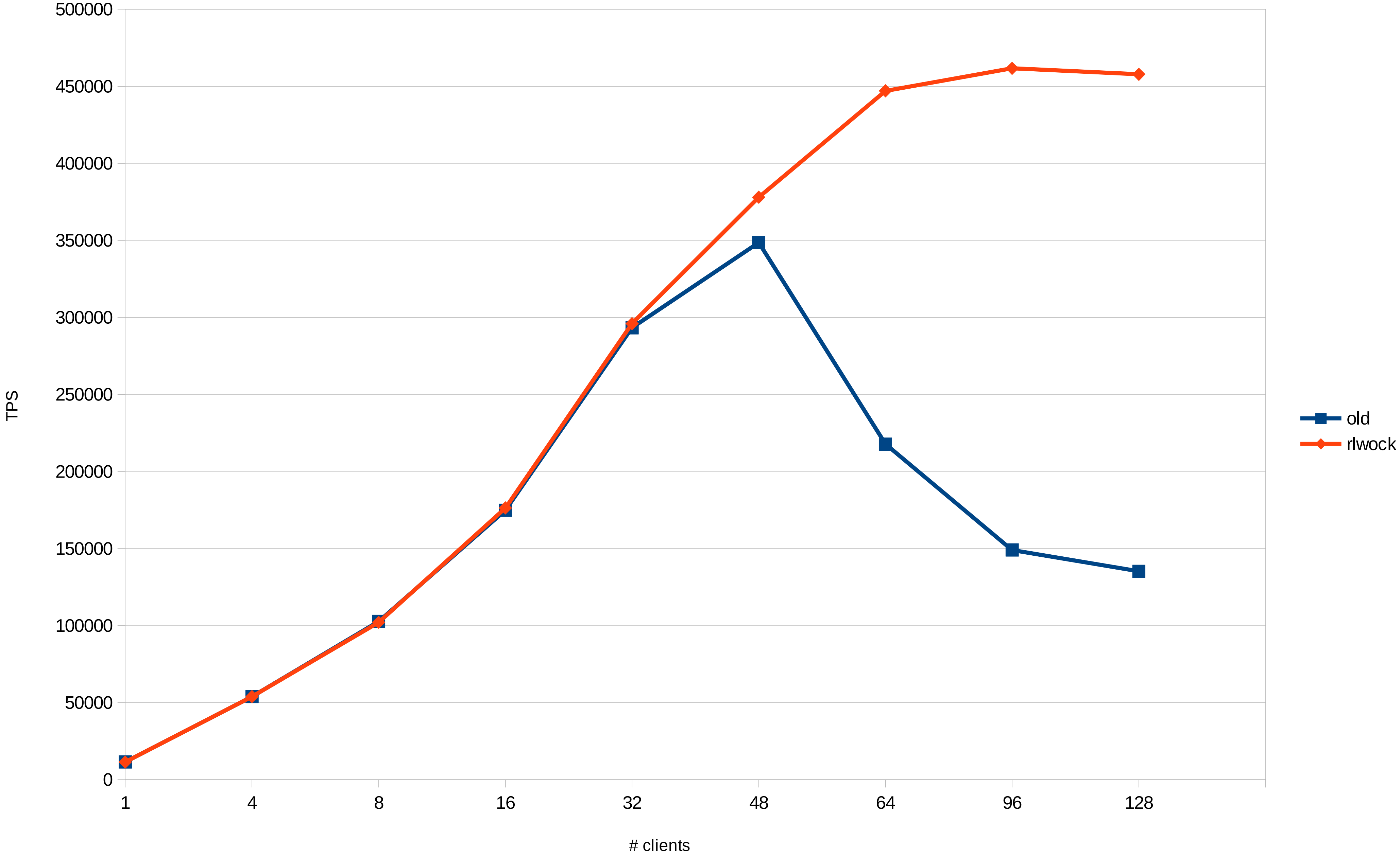
citusdata

Fix this!

- Use atomic operations
  - atomic add & subtract, compare exchange
- Complex, due to queuing

```
if (!atomic_try_acquire(lock, mode))
{
    QueueSelf();
    if (!atomic_try_acquire(lock, mode))
        WaitForRelease();
        goto retry;
    else
        UnQueueSelf();
}
```

citusdata



- readonly pgbench
- 4xE5-4620
- scale 100

citusdata

Buffer Descriptors & Buffers

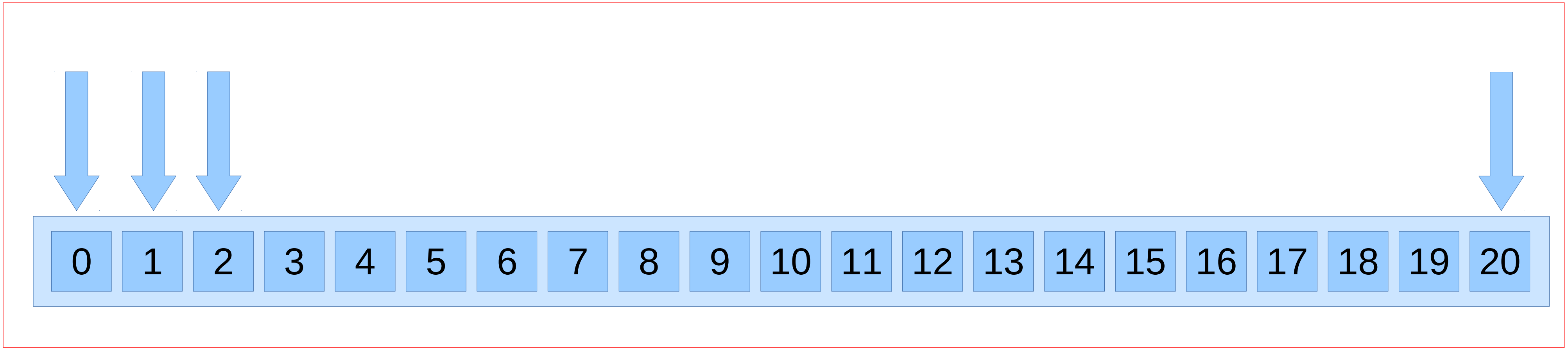
```
struct BufferDesc
{
    BufferTag   tag;            /* ID of page contained in buffer */

    uint16      usage_count;  /* usage counter for clock sweep */
    unsigned    refcount;     /* # of backends holding pins */

    slock_t     buf_hdr_lock; /* protects the above fields */
} BufferDesc;
```

citusdata

Inefficient Buffer Replacement

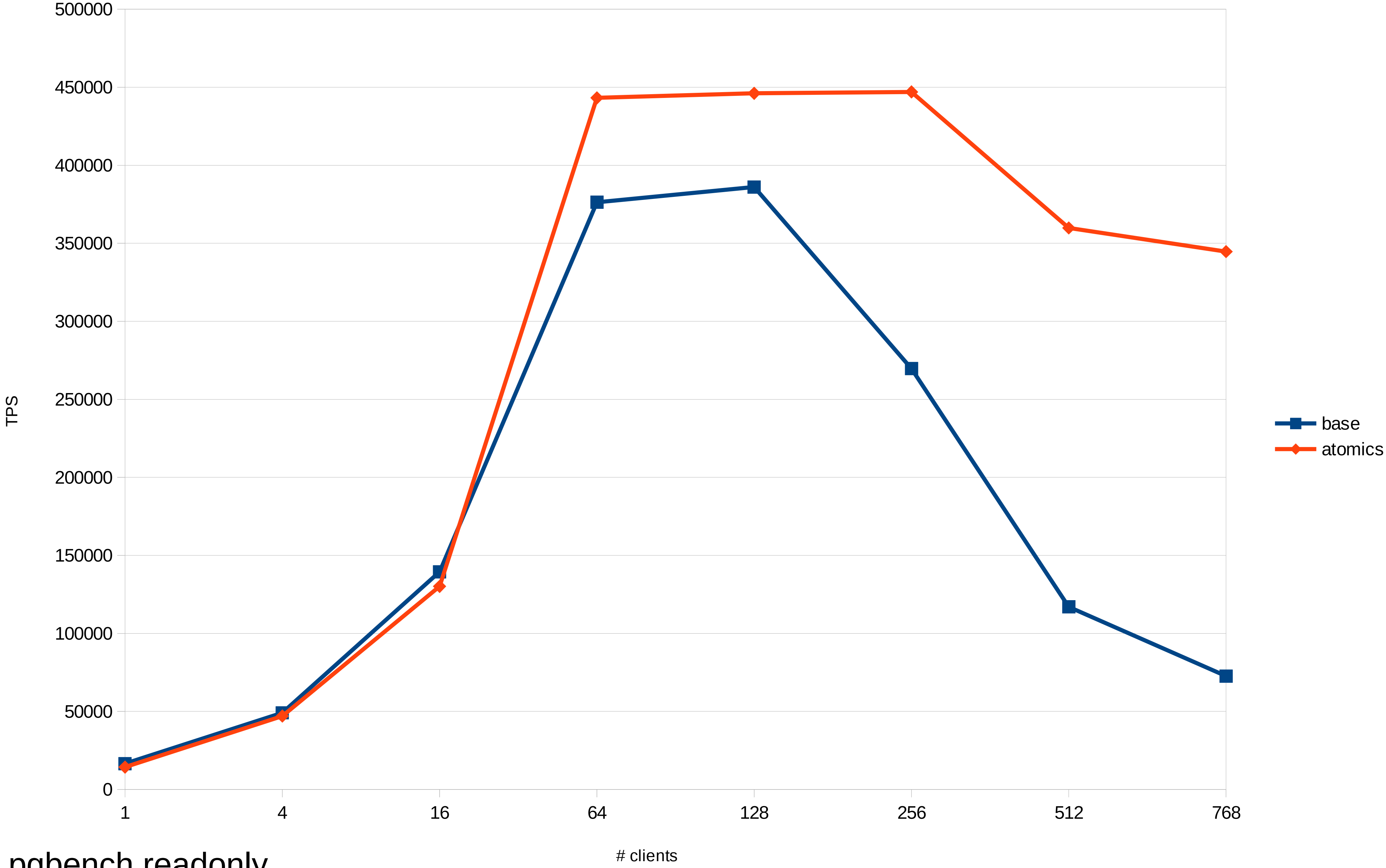


citusdata

Fix It

- granular locking: spinlock per clock tick
- atomics:

```
victim = pg_atomic_fetch_add_u32(
    &StrategyControl->nextVictimBuffer, 1);
victim = victim % NBuffers;
```



- pgbench readonly
- scale 1000 (~14GB)
- 4GB shared buffers
- EC2 m4.8xlarge - 2 x E5-2676
- master @ aa6b2e6

citusdata

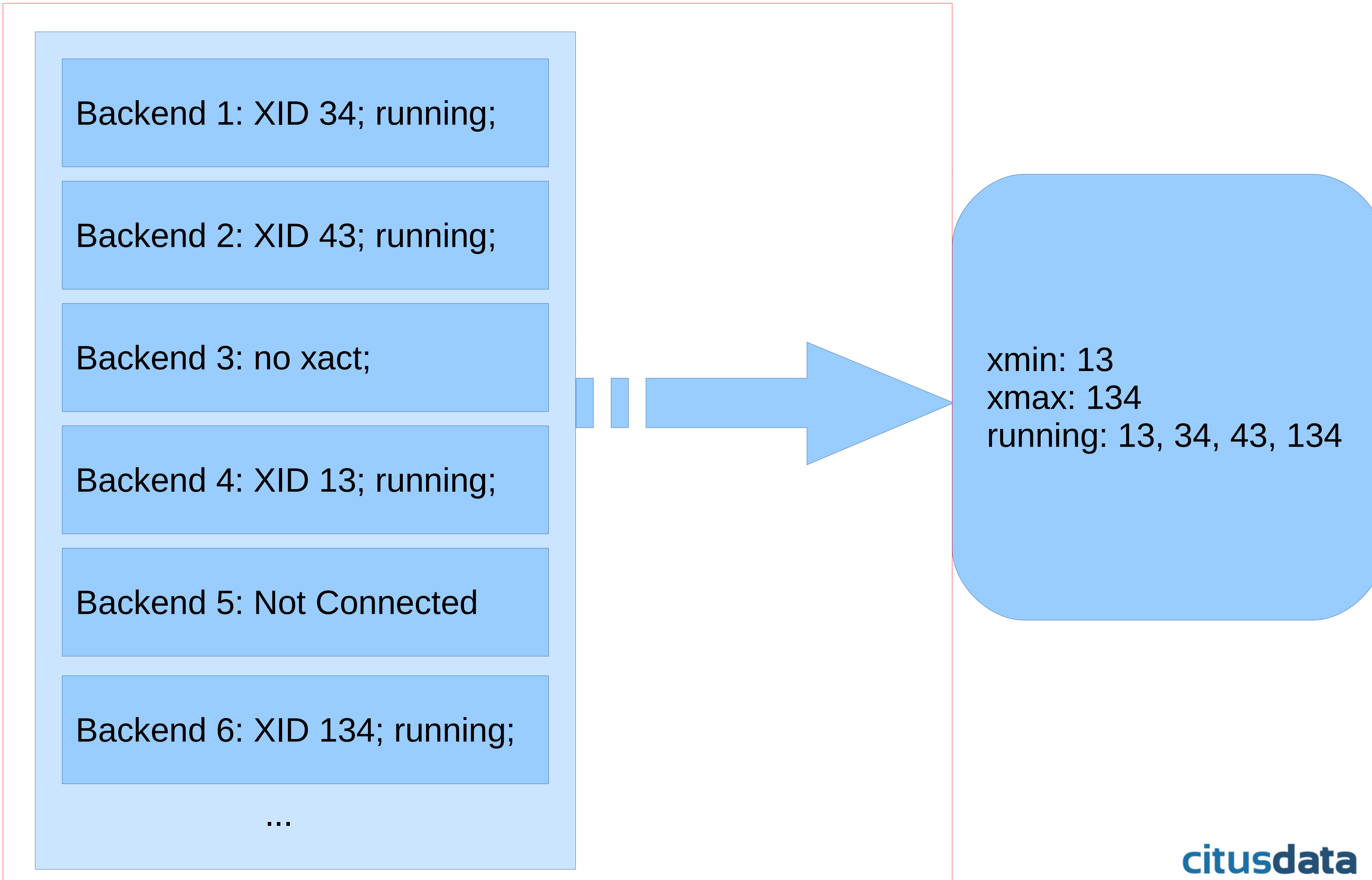
Scalability Approaches

- Avoid locks in common cases
- More efficient locking
- Atomic operations
- More granular locking

Not Yet Fixed Scalability Issues

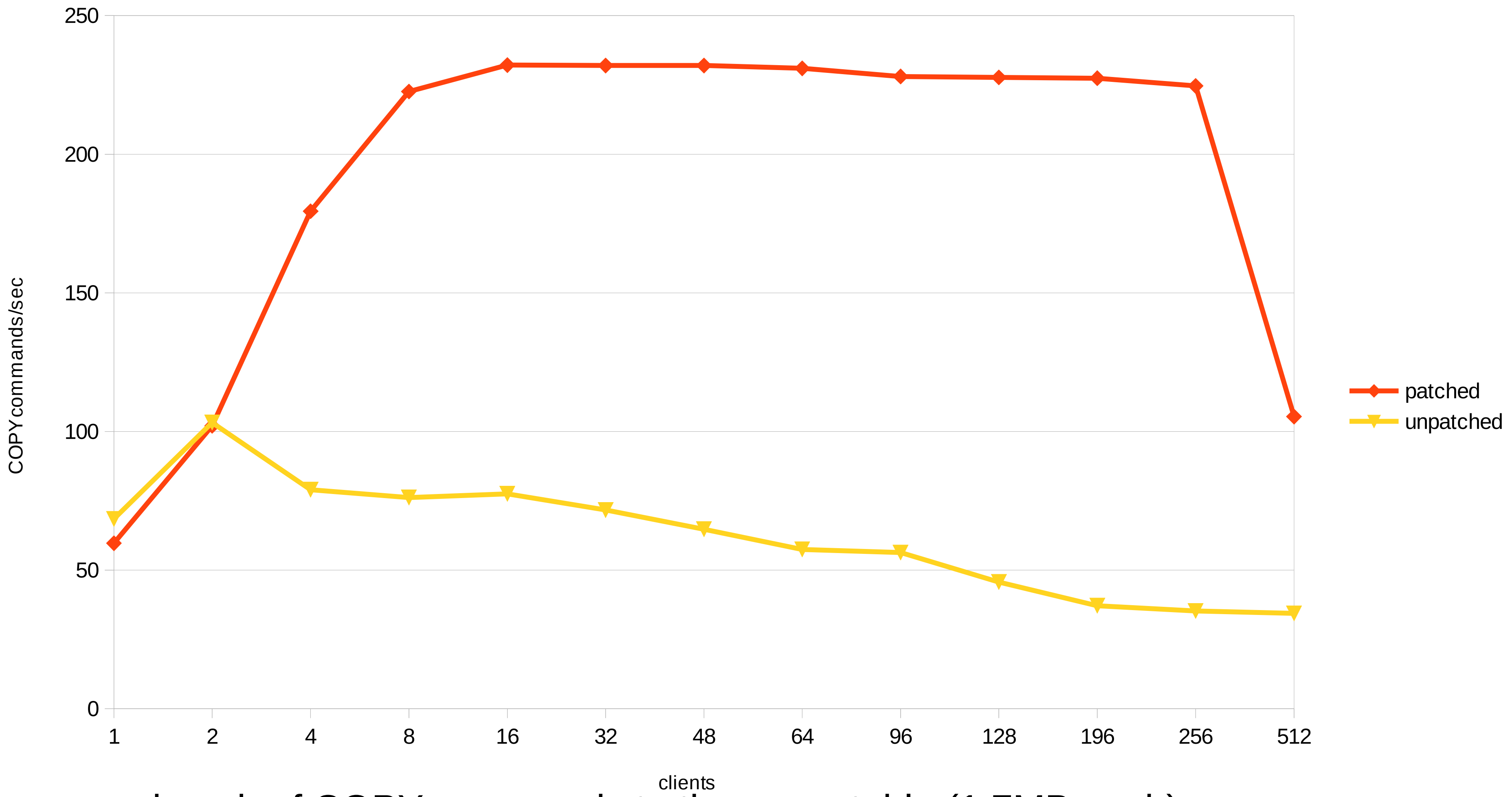
- Extension Lock
  - Problematic: Bulk write workloads
- Buffer Replacement Complexity & Accuracy
  - Problematic: Larger than memory workloads
- Expensive Snapshot Computation
  - Problematic: High QPS (combined read & write) workloads
- Buffer Pins use spinlocks
  - Problematic: Lots of accesses to the same buffer

Expensive Snapshot Computation



citusdata

Extension Lock Scalability



- pgbench of COPY commands to the same table (1.7MB each)
- 4xE5-4620 (32 cores, 64 threads)
- 48 GB shared memory, 256 GB in total

citusdata