```
sim_done = 1
sim_done = 4
sim_done = 3
sim_done = 2
sim_done = 5
sim_done = 6
PASS! SimTime = 1.92208e+10
=Worker Thread=
    Throughput:                     148207
    MaxThr:                         148203
    PerThdThr:                      148207
    run_time:                       115.297
(19.2161,19.216,19.2163,19.2167,19.2156,19.2161,0,)
    log_bytes:                      0 (0,0,0,0,0,0,0,)
    log_dep_size:                   0 (0,0,0,0,0,0,0,)
    log_total_size:                 0 (0,0,0,0,0,0,0,)
    latency:                        253.288
(42.2184,42.2165,42.217,42.2109,42.2118,42.2131,0,)

    num_commits:                    2.84797e+06
(500000,474687,499111,401651,498028,474496,0,)
    num_async_commits:              0 (0,0,0,0,0,0,0,)
    num_aborts:                     59 (13,5,24,1,13,3,0,)
    num_aborts_logging:             0 (0,0,0,0,0,0,0,)
    num_log_records:                0 (0,0,0,0,0,0,0,)
    log_data:                       0 (0,0,0,0,0,0,0,)
    num_latency_count:              2.84803e+06
(500013,474692,499135,401652,498041,474499,0,)
    num_raw_edges:                  0 (0,0,0,0,0,0,0,)
    num_waw_edges:                  0 (0,0,0,0,0,0,0,)
    num_war_edges:                  0 (0,0,0,0,0,0,0,)
    int_num_log:                    0 (0,0,0,0,0,0,0,)
    int_debug_get_next:             0 (0,0,0,0,0,0,0,)
    int_debug1:                     0 (0,0,0,0,0,0,0,)
    int_debug2:                     0 (0,0,0,0,0,0,0,)
    int_debug3:                     0 (0,0,0,0,0,0,0,)
    int_debug4:                     0 (0,0,0,0,0,0,0,)
    int_debug5:                     0 (0,0,0,0,0,0,0,)
    int_debug6:                     0 (0,0,0,0,0,0,0,)
    int_debug7:                     0 (0,0,0,0,0,0,0,)
    int_debug8:                     0 (0,0,0,0,0,0,0,)
    int_debug9:                     0 (0,0,0,0,0,0,0,)
    int_debug10:                    0 (0,0,0,0,0,0,0,)
    int_psnflush:                   0 (0,0,0,0,0,0,0,)
    int_flush_time_interval:        0 (0,0,0,0,0,0,0,)
    int_flush_half_full:            0 (0,0,0,0,0,0,0,)
    int_rec_fail_to_insert:         0 (0,0,0,0,0,0,0,)
    int_num_get_row:                0 (0,0,0,0,0,0,0,)
    int_locktable_volume:           0 (0,0,0,0,0,0,0,)
    int_aux_bytes:                  0 (0,0,0,0,0,0,0,)
    int_nonzero:                    0 (0,0,0,0,0,0,0,)
    num_log_entries:                0 (0,0,0,0,0,0,0,)
```

```
    time_ts_alloc:                0.414599 0.0690998 0.791103% 145.577
(0.0780597,0.0640053,0.0760691,0.0557177,0.0774345,0.0633127,0,)
    time_man:                     103.657 17.2762 197.79% 36396.8
(17.4497,16.9893,17.4421,17.3147,17.4403,17.021,0,)
    time_cleanup:                 78.1796 13.0299 149.176% 27451
(13.8762,12.6601,13.8817,11.1327,13.8947,12.7342,0,)
    time_txn:                     0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_index:                   1.28026 0.213376 2.44288% 449.532
(0.206983,0.223236,0.206608,0.202331,0.205339,0.235759,0,)
    time_log:                     0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_io:                      0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_wait_io:                 0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_phase1_add_graph:        0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover_txn:             0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_lv_overhead:             0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_state_malloc:            0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_phase1_1:                3.79497 0.632495 7.24125% 1332.52
(0.61578,0.72249,0.548652,0.700306,0.535224,0.672518,0,)
    time_phase1_2:                2.59121 0.431869 4.94434% 909.844
(0.396998,0.49119,0.399693,0.414567,0.414911,0.473853,0,)
    time_phase2:                  5.94461 0.990769 11.343% 2087.31
(0.887513,1.13468,0.92201,0.949424,0.953313,1.09768,0,)
    time_phase3:                  3.49224 0.58204 6.6636% 1226.22
(0.492516,0.696787,0.497725,0.619835,0.489438,0.695936,0,)
    time_phase1_1_raw:            15.5355 2.58925 29.6436% 5454.93
(2.1276,2.55492,2.12508,4.06357,2.10918,2.55515,0,)
    time_phase1_2_raw:            0.316563 0.0527605 0.60404% 111.154
(0.0536517,0.0547812,0.0530449,0.0466582,0.0533808,0.0550464,0,)
    time_phase2_raw:              0.258381 0.0430635 0.493022% 90.7246
(0.041984,0.0492166,0.0398061,0.0321466,0.041186,0.0540419,0,)
    time_phase3_raw:              0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover_full:            0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover1:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover2:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover3:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover4:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover5:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover6:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover7:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_recover8:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug_get_next:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug0:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug1:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug2:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug3:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug4:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug5:                  2.72727e-08 6.81818e-09 7.80593e-08% 9.57619e-06
(7.27273e-09,0,6.36364e-09,6.36364e-09,7.27273e-09,0,0,)
    time_debug6:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug7:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug8:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug9:                  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug10:                 0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug11:                 19.3175 3.21959 36.8601% 6782.91
```

```
(3.23191,3.2176,3.21231,3.22556,3.1632,3.26696,0,)
    time_debug12:                 95.3181 15.8863 181.878% 33468.7
(15.8614,15.8935,15.8816,15.8992,15.9306,15.8518,0,)
    time_debug13:                 0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug14:                 0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_debug15:                 0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_rec_loop_tryRead:        0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_rec_finding_empty_slot:  0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate1:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate2:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate3:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate4:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate5:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate6:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate7:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate8:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_silo_validate9:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_locktable_get:           16.9353 2.82255 32.3146% 5946.45
(2.30687,2.64078,2.12636,5.14479,2.11503,2.60149,0,)
    time_locktable_get_validation:0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_locktable_release:       0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_get_row_before:          0.847544 0.141257 1.61721% 297.595
(0.123859,0.16625,0.126017,0.141167,0.134412,0.155838,0,)
    time_get_row_after:           8.00229 1.33371 15.2693% 2809.82
(1.19266,1.57466,1.35802,0.946161,1.34645,1.58434,0,)
    time_log_create:              0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_log_serialLogTxn:        0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_cleanup_1:               0.0261125 0.00435208 0.0498257% 9.16879
(0.00436762,0.00456649,0.00439663,0.0038963,0.00435827,0.00452717,0,)
    time_cleanup_2:               0.0449531 0.00749219 0.0857758% 15.7842
(0.00760759,0.00780601,0.00758073,0.00655782,0.00758384,0.00781711,0,)
    time_insideSLT1:              0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_insideSLT2:              0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_STLother:                0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    time_logging_thread:          0 -nan -nan% 0 (0,0,0,0,0,0,0,)
    int_serialLogFail:            0 (0,0,0,0,0,0,0,)
Projected Disk Bandwidth Utilized - avg -nan real -nan
Total time measured 25.6613
Total wall time observed 25.721
Estimated CPU_FREQ is 2.19489
replace 0 tuple
thread 0 use 4967020 tuple
thread 1 use 4647517 tuple
thread 2 use 4815079 tuple
thread 3 use 4167536 tuple
thread 4 use 4410761 tuple
thread 5 use 4267674 tuple
Closing pool file.
```

注意

1. 可通过 STATS_ENABLE 和 STAT_VERBOSE 控制输出信息，前一个控制是否输出，后一个控制统计信息等级（等级为 0 时仅输出部分信息，目前等级为 1）
2. 目前，下面分线程输出时括号里有 7 项，前 6 项表示 6 个线程，最后一项是“log 线程”（？）的信息，不过全是 0
3. 和恢复相关的都没算进去

---

1. run_time: 单个线程中 在 thread.cpp 里的 run()函数里的主 while()循环里执行 run_txn()及同时进行的一些相关操作的时间的累加和。{ 在 run_txn()里会再调用提交不同的事务如 payment }

   即单个线程的所有事务的执行时间以及其他相关操作时间的累加和

   代码位置：

```
  while (true) {
    ts_t starttime = get_sys_clock();
    if (WORKLOAD != TEST) {
      int trial = 0;
      if (_abort_buffer_enable) {
        m_query = NULL;
        while (trial < 2) {
          ts_t curr_time = get_sys_clock();
          ts_t min_ready_time = UINT64_MAX;
          if (_abort_buffer_empty_slots < _abort_buffer_size) {
            for (int i = 0; i < _abort_buffer_size; i++) {
              if (_abort_buffer[i].query != NULL &&
                  curr_time > _abort_buffer[i].ready_time) {
                m_query = _abort_buffer[i].query;
                _abort_buffer[i].query = NULL;
                _abort_buffer_empty_slots++;
                break;
              } else if (_abort_buffer_empty_slots == 0 &&
                         _abort_buffer[i].ready_time < min_ready_time)
                min_ready_time = _abort_buffer[i].ready_time;
            }
          }
          if (m_query == NULL && _abort_buffer_empty_slots == 0) {
            assert(trial == 0);
            // M_ASSERT(min_ready_time >= curr_time, "min_ready_time=%ld,
            // curr_time=%ld\n", min_ready_time, curr_time);
            assert(min_ready_time >= curr_time);
            usleep(min_ready_time - curr_time);
          } else if (m_query == NULL) {
            m_query = query_queue->get_next_query(_thd_id);
#if CC_ALG == WAIT_DIE
            m_txn->set_ts(get_next_ts());
#endif
          }
          if (m_query != NULL)
            break;
        }
      } else {
```

```
        if (rc == RCOK)
          m_query = query_queue->get_next_query(_thd_id);
      }
    }
    INC_STATS(_thd_id, time_query, get_sys_clock() - starttime);
    m_txn->set_start_time(get_sys_clock());
    m_txn->abort_cnt = 0;
    //#if CC_ALG == VLL
    //    _wl->get_txn_man(m_txn, this);
    //#endif
    m_txn->set_txn_id(get_thd_id() + thd_txn_id * g_thread_cnt); // qfs
    thd_txn_id++;

    if ((CC_ALG == HSTORE && !HSTORE_LOCAL_TS) || CC_ALG == MVCC ||
        CC_ALG == HEKATON || CC_ALG == TIMESTAMP)
      m_txn->set_ts(get_next_ts());

    rc = RCOK;
#if CC_ALG == HSTORE
    rc = part_lock_man.lock(m_txn, m_query->part_to_access, m_query->part_num)
#elif CC_ALG == VLL
    vll_man.vllMainLoop(m_txn, m_query);
#elif CC_ALG == MVCC || CC_ALG == HEKATON
    glob_manager->add_ts(get_thd_id(), m_txn->get_ts());
#elif CC_ALG == OCC
    // In the original OCC paper, start_ts only reads the current ts without
    // advancing it. But we advance the global ts here to simplify the
    // implementation. However, the final results should be the same.
    m_txn->start_ts = get_next_ts();
#endif
    if (rc == RCOK) {
#if CC_ALG != VLL
      rc = m_txn->run_txn(m_query);
#endif
#if CC_ALG == HSTORE
      part_lock_man.unlock(m_txn, m_query->part_to_access, m_query->part_num);
#endif
    }
    if (rc == Abort) {
      // cout << m_txn->get_txn_id() << " Aborted" << endl;
      uint64_t penalty = 0;
      if (ABORT_PENALTY != 0) {
        double r;
        // drand48_r(&buffer, &r);
        r = erand48(buffer);
        penalty = r * ABORT_PENALTY;
      }
      if (!_abort_buffer_enable)
        usleep(penalty / 1000);
      else {
        assert(_abort_buffer_empty_slots > 0);
        for (int i = 0; i < _abort_buffer_size; i++) {
          if (_abort_buffer[i].query == NULL) {
            _abort_buffer[i].query = m_query;
            _abort_buffer[i].ready_time = get_sys_clock() + penalty;
            _abort_buffer_empty_slots--;
```

```
                break;
            }
        }
    }
}

ts_t endtime = get_sys_clock();
uint64_t timespan = endtime - starttime;
INC_FLOAT_STATS_V0(run_time, timespan);
```

2. total_run_time:6 个线程 run_time 的和
3. num_commits:单个线程提交的总事务数
4. total_num_commits:6 个线程提交的总事务数

以下公式里忽略掉单位相关的乘数

1. Throughput: 吞吐量

   公式：total_num_commits/ (total_run_time/6)

2. MaxThr: 不知道是啥，跟吞吐量差不多

   公式：total_num_commits/ max_run_time

3. PerThdThr:线程平均吞吐量?

   公式：6 个线程的（ num_commits/ run_time ）的和

4. run_time:最前面的是 total_run_time,后面的是 6 个线程各自的 run_time。以下 5，6，7，8 格式与这里相同。
5. latency: start 是在 thread.cpp 的主循环里 run_txn()函数前的 set_start_time()，end 是在 cleanup() 里，latency=end-start。latency 比 run_time 大的原因：run_time 在输出前会除以 CPU_FREQ(=2.2)，而 latency 不会；实际上二者确实是大约相差两倍。
6. num_commits:见前
7. num_aborts:线程 abort 事务数
8. num_latency_count:计算 lantency 的次数。单个线程中，lantency 每累加、改变一次，该变量加 1
9. time_ts_alloc: 线程里获得全局时间戳耗费的时间的累加（比如包括互斥锁的开销）。

   第一项是 6 个线程的 total 值，第二项是去掉 0 值的 6 个线程平均值 nonzero_avg；第三项是一个百分数：nonzero_avg / (total_run_time / 6) ；第四项是 total / total_num_commits；最后面是 6 个线程的各自的值。下面的各个变量格式也是这样。

10. time_man:线程里 get_row()的时间的累加和 finish()的时间的累加的和。（目前 finish()里基本上只有 cleanup()）
11. time_cleanup:线程里 finish()的时间的累加。（目前 finish()里基本上只有 cleanup()）

12. time_index:线程里 查索引(index_read()函数)的时间的累加。

13. time_phase1_1: new_order 事务里第 1-1 部分 SQL 语句耗费的时间 加上 cleanup()里见下图的部分感觉主要是 return_row()函数耗费的时间 的和

system > txn.cpp

```cpp
  uint64_t starttime = get_sys_clock();
  // start to release the locks
#if CC_ALG != SILO // updating the data is already handled in silo_validate
  for (int rid = row_cnt - 1; rid >= 0; rid--) {
    row_t *orig_r = accesses[rid]->orig_row;
    access_t type = accesses[rid]->type;
    if (type == WR && rc == Abort)
      type = XP; // means we need to roll back the data value

#if (CC_ALG == NO_WAIT || CC_ALG == DL_DETECT) &&                         \
    ISOLATION_LEVEL == REPEATABLE_READ
    if (type == RD) {
      accesses[rid]->data = NULL;
      continue;
    }
#endif

    char *newdata;
    if (ROLL_BACK && type == XP &&
        (CC_ALG == DL_DETECT || CC_ALG == NO_WAIT || CC_ALG == WAIT_DIE)) {
#if NEW_ROW
      newdata = ((row_t *)accesses[rid]->orig_data)->tuple_data->data;
#else
      newdata = ((row_t *)accesses[rid]->orig_data)
                    ->data; // fixed a bug left from the original code base
#endif

    } else {
      newdata = accesses[rid]->data;
    }
#if USE_LOCKTABLE
    LockTable &lt = LockTable::getInstance();
    uint64_t current_time = get_sys_clock();
// assert((uint64_t)newdata != 0);
#if LOG_ALGORITHM == LOG_TAURUS
    lt.release_lock(orig_r, type, this, newdata, lsn_vector, NULL, rc);
#elif LOG_ALGORITHM == LOG_SERIAL
    lt.release_lock(orig_r, type, this, newdata, NULL, &_max_lsn, rc);
#else
    lt.release_lock(orig_r, type, this, newdata, NULL, NULL, rc);
#endif
    INC_INT_STATS(time_locktable_release, get_sys_clock() - current_time);
#else
    orig_r->return_row(type, this, newdata, accesses[rid]->row_data, rc);
#endif
#if CC_ALG != TICTOC && CC_ALG != SILO
    accesses[rid]->data = NULL; // will not need this any more
#endif
  }
#if VERBOSE_LEVEL > 0
  stringstream ssk;
  ssk << GET_THD_ID << " finishes" << endl << endl;
  cout << ssk.str();
#endif

#endif
  uint64_t cleanup_1_begin = get_sys_clock();
  INC_INT_STATS(time_phase1_1, cleanup_1_begin - starttime);
```

为啥是这两部分，没啥规律啊

14. time_phase1_2: new_order 事务里第 1-2 部分 SQL 语句耗费的时间 加上 cleanup()里一部分和 log 相关的代码（我感觉目前 log_no 时实际上这一部分什么也没执行）耗费的时间的和。

15. time_phase2: new_order 事务里第 2 部分 SQL 语句耗费的时间。

16. time_phase3: new_order 事务里第 3 部分 SQL 语句耗费的时间。

17. time_phase1_1_raw: new_order 事务里第 3 部分 SQL 语句后面一部分语句耗费的时间。

18. time_phase1_2_raw: new_order 事务里再往后,,,,,

19. time_phase2_raw: new_order 事务里再往后,,,,,

20. time_debug5: 在 log.cpp 里几个函数里耗费的时间(不清楚这几个函数的用途和是否调用) 加上 try_commit_txn()里耗费的时间

21. time_debug11:在 tpcc_txn.cpp 的 run_txn（）里执行 run_payment()函数耗费的时间(见下图) 加上 log.cpp 里几个函数里耗费的时间(不清楚这几个函数的用途和是否调用)

```
RC tpcc_txn_man::run_txn(base_query *query, bool rec) {
  _query = (tpcc_query *)query;
  RC rc = RCOK;
  //  cout << _query->type << endl;
  uint64_t starttime = get_sys_clock();
  switch (_query->type) {
  case TPCC_PAYMENT:
    rc = run_payment(_query);
    COMPILER_BARRIER
    INC_INT_STATS(time_debug11, get_sys_clock() - starttime);
    break;
```

22. time_debug12:在 tpcc_txn.cpp 的 run_txn（）里执行 run_new_order()函数耗费的时间(见下图) 加上 log.cpp 里几个函数里耗费的时间(不清楚这几个函数的用途和是否调用)

```
RC tpcc_txn_man::run_txn(base_query *query, bool rec) {
  _query = (tpcc_query *)query;
  RC rc = RCOK;
  //  cout << _query->type << endl;
  uint64_t starttime = get_sys_clock();
  switch (_query->type) {
  case TPCC_PAYMENT:
    rc = run_payment(_query);
    COMPILER_BARRIER
    INC_INT_STATS(time_debug11, get_sys_clock() - starttime);
    break;
  case TPCC_NEW_ORDER:
    rc = run_new_order(_query);
    COMPILER_BARRIER
    INC_INT_STATS(time_debug12, get_sys_clock() - starttime);
    break;
```

23. time_locktable_get: 见下图，基本上是只有 810 行 get_row()里再调用另一个 get_row()函数所耗费的时间

```
    uint64_t right_before_get = get_sys_clock();
    INC_INT_STATS_V0(time_get_row_before, right_before_get - starttime);
    // if(row_cnt == 19)
    //  printf("Encounter 19, before get_row as %" PRIu64 ", type %d\n",
    //(uint64_t)row, type);
#if USE_LOCKTABLE && CC_ALG != SILO
    LockTable &lt = LockTable::getInstance();

#if LOG_ALGORITHM == LOG_TAURUS
    rc = lt.get_row(row, type, this, accesses[row_cnt]->data, lsn_vector, NULL);
#elif LOG_ALGORITHM == LOG_SERIAL
    rc = lt.get_row(row, type, this, accesses[row_cnt]->data, NULL,
                    &_max_lsn); //, true, 0, true);
#else
    rc = lt.get_row(row, type, this, accesses[row_cnt]->data, NULL, NULL);
#endif

#else
    // if (accesses[row_cnt]->row_data == NULL)
    //   cout << "qfs::" << 1 << " " << type << "!!\n";
    rc = row->get_row(type, this, accesses[row_cnt]->data,
                      accesses[row_cnt]->row_data);
#endif

#if LOG_ALGORITHM == LOG_TAURUS
    partition_accesses_cnt[logPartition((uint64_t)row)]++;
#endif

    uint64_t starttime2 = get_sys_clock();
    INC_INT_STATS_V0(time_locktable_get, starttime2 - right_before_get);
```

24. time_get_row_before: 见下图，get_row()里这一部分所耗费的时间。

```
    uint64_t starttime = get_sys_clock();
    RC rc = RCOK;
    if (row_cnt == num_accesses_alloc) {
      // accesses[row_cnt] == NULL // bad
      Access *access = (Access *)MALLOC(sizeof(Access), GET_THD_ID);
      accesses[row_cnt] = access;

#if (CC_ALG == SILO || CC_ALG == TICTOC)
      access->data = new char[MAX_TUPLE_SIZE];
      //#if LOG_ALGORITHM == LOG_TAURUS
      //    access->orig_data = (char*) row;
      //#else
      access->orig_data = NULL;
//#endif
#elif (CC_ALG == DL_DETECT || CC_ALG == NO_WAIT || CC_ALG == WAIT_DIE)
      accesses[row_cnt]->orig_data = (char *)MALLOC(sizeof(row_t), GET_THD_ID);
      ((row_t *)(accesses[row_cnt]->orig_data))->init(MAX_TUPLE_SIZE);
#endif

      num_accesses_alloc++;
    }
    uint64_t right_before_get = get_sys_clock();
    INC_INT_STATS_V0(time_get_row_before, right_before_get - starttime);
```

25. time_get_row_after:get_row()里从 time_locktable_get 结束的时间点到函数最后所耗费的时间。

26. time_cleanup_1: cleanup()里这一部分代码耗费的时间。

```
    uint64_t cleanup_1_begin = get_sys_clock();
    INC_INT_STATS(time_phase1_1, cleanup_1_begin - starttime);
    if (rc == Abort) { // remove inserted rows.
        for (UInt32 i = 0; i < insert_cnt; i++) {
            row_t *row = insert_rows[i];
            assert(g_part_alloc == false);
#if CC_ALG != HSTORE && CC_ALG != OCC
            mem_allocator.free(row->manager, 0);
#endif
            row->free_row();
            mem_allocator.free(row, sizeof(row));
        }
    }
    uint64_t cleanup_1_end = get_sys_clock();
    INC_INT_STATS(time_cleanup_1, cleanup_1_end - cleanup_1_begin);
```

27. time_cleanup_2: cleanup()里这一部分代码耗费的时间。

```
    uint64_t cleanup2_begin = get_sys_clock();
    INC_INT_STATS(time_phase1_2, cleanup2_begin - cleanup_1_end);
    try_commit_txn(); // no need to try_commit_txn if abort
#else              // LOG_ALGORITHM == LOG_NO
    uint64_t cleanup2_begin = get_sys_clock();
    INC_INT_STATS_V0(num_latency_count, 1);
    INC_FLOAT_STATS_V0(latency, get_sys_clock() - _txn_start_time);
#endif

#if LOG_ALGORITHM == LOG_PLOVER
    memset(_log_entry_sizes, 0, sizeof(uint32_t) * g_num_logger);
#else
    _log_entry_size = 0;
#endif
    row_cnt = 0;
    wr_cnt = 0;
    insert_cnt = 0;
#if LOG_ALGORITHM == LOG_PARALLEL
    _num_raw_preds = 0;
    _num_waw_preds = 0;
#elif LOG_ALGORITHM == LOG_SERIAL
    _max_lsn = 0;
#elif LOG_ALGORITHM == LOG_TAURUS
    _max_lsn = 0;
    memset(partition_accesses_cnt, 0, sizeof(uint64_t) * g_num_logger);
    memset(lsn_vector, 0, sizeof(lsnType) * g_num_logger);
#endif
#if CC_ALG == DL_DETECT
    dl_detector.clear_dep(get_txn_id());
#endif
    INC_INT_STATS(time_cleanup_2, get_sys_clock() - cleanup2_begin);
```