```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <inttypes.h>
#include <cmath>
#include <vector>
#include <time.h>
#include <chrono>

#include "./pcm/pcm-memory.cpp"
#include "./pcm/pcm-numa.cpp"
#include "./papi_util.cpp"

#include "microbench.h"

#include <cstring>
#include <cctype>
#include <atomic>

thread_local long skiplist_steps = 0;
std::atomic<long> skiplist_total_steps;

struct tuple_t {
    std::string key;
    std::string value;

    tuple_t(std::string key_arg, std::string value_arg)
            : key(std::move(key_arg)), value(std::move(value_arg))
    {}

    tuple_t(const tuple_t &t) = default;
    tuple_t(tuple_t &&t) = default;
};

struct res_t {
    int init_s;
    int init_f;
```

```
    float t1;

    res_t(int init_s_arg, int init_f_arg, float t1_arg)
        : init_s(init_s_arg), init_f(init_f_arg), t1(t1_arg)
    {}

    res_t(const res_t &t) = default;
    res_t(res_t &&r) = default;
};


struct res_extr_t{
    int insert_s;
    int insert_f;
    int read_s;
    int read_f;
    int update_s;
    int update_f;
    int delete_s;
    int delete_f;

    float t2;

    res_extr_t(int insert_s_arg, int insert_f_arg,
               int read_s_arg, int read_f_arg, int update_s_arg, int update_f_arg,
               int delete_s_arg, int delete_f_arg, float t2_arg)
           : insert_s(insert_s_arg), insert_f(insert_f_arg),
             read_s(read_s_arg), read_f(read_f_arg),
             update_s(update_s_arg), update_f(update_f_arg),
             delete_s(delete_s_arg), delete_f(delete_f_arg),
             t2(t2_arg)
    {}

    res_extr_t(const res_extr_t &t) = default;
    res_extr_t(res_extr_t &&r) = default;
};


struct thread_arg_t{
    size_t start_i;
    size_t end_i;
    int tr_idx;
};
```

```
static std::vector<tuple_t> tuples;
static std::vector<tuple_t> extr_tuples;
static std::vector<tuple_t> delete_tuples;

int iteration = 0;

static std::vector<res_t> init_results;
static std::vector<res_extr_t> extr_results;
static std::vector<res_t> del_results;

static std::vector<int> _sockets;

static char *server_address = "loki08.no.oracle.com";
//static char *server_address = "10.172.139.128";
static const u_int16_t server_port = 11217;

//#define USE_TBB

#ifdef USE_TBB
#include "tbb/tbb.h"
#endif

// Enable this if you need pre-allocation utilization
//#define BWTREE_CONSOLIDATE_AFTER_INSERT

#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
#ifdef USE_TBB
  #warning "Could not use TBB and BwTree consolidate together"
  #endif
#endif

#ifdef BWTREE_COLLECT_STATISTICS
#ifdef USE_TBB
#warning "Could not use TBB and BwTree statistics together"
#endif
#endif

// Whether insert interleaves
//#define INTERLEAVED_INSERT

// Whether read operatoin miss will be counted
//#define COUNT_READ_MISS
```

```
typedef uint64_t keytype;
typedef std::less<uint64_t> keycomp;

static const uint64_t key_type=0;
static const uint64_t value_type=1; // 0 = random pointers, 1 = pointers to keys

extern bool hyperthreading;

// This is the flag for whather to measure memory bandwidth
static bool memory_bandwidth = false;
// Whether to measure NUMA Throughput
static bool numa = false;
// Whether we only perform insert
static bool insert_only = false;

static bool del = false;

// We could set an upper bound of the number of loaded keys
static int64_t max_init_key = -1;

#include "util.h"

/*
 * MemUsage() - Reads memory usage from /proc file system
 */
size_t MemUsage() {
   FILE *fp = fopen("/proc/self/statm", "r");
   if(fp == nullptr) {
      fprintf(stderr, "Could not open /proc/self/statm to read memory usage\n");
      exit(1);
   }

   unsigned long unused;
   unsigned long rss;
   if (fscanf(fp, "%ld %ld %ld %ld %ld %ld %ld", &unused, &rss, &unused, &unused, &unused,
&unused, &unused) != 7) {
      perror("");
      exit(1);
   }

   (void)unused;
   fclose(fp);
```

```
    return rss * (4096 / 1024); // in KiB (not kB)
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int setup_connection(char* server_address, u_int16_t port)
{
    // Try to create a socket
    int _socket = socket(AF_INET, SOCK_STREAM, 0);
    if (_socket < 0) {
        perror("Cannot create a socket"); exit(1);
    }

    // Fill in the address of server
    struct sockaddr_in server;
    memset(&server, 0, sizeof(server));

    char* server_host = "localhost";
    if (server_address != nullptr) server_host = server_address;


    // Resolve the server address (convert from symbolic name to IP number)
    struct hostent *host = gethostbyname(server_host);
    if (host == NULL) {
        perror("Cannot define host address"); exit(1);
    }
    server.sin_family = AF_INET;
    u_int16_t server_port = port;
    server.sin_port = htons(server_port);

    // Write resolved IP address of a server to the address structure
    memmove(&(server.sin_addr.s_addr), host->h_addr_list[0], 4);

    // Connect to a remote server
    int res = connect(_socket, (struct sockaddr*) &server, sizeof(server));
    if (res < 0) {
        perror("Cannot connect"); exit(1);
    }
    //printf("Connected, ready to send data.\n");

    return _socket;
}
```

```
int send_query(int _socket, const std::string &op_key, const std::string &val)
{
   char buffer[512];
   long res = write(_socket, (op_key+" "+val+"\0").c_str(), ((op_key.length())+(val.length()))+2);
   if (res < 0){
      //std::cout << "An error occurred sending info to the server, shutting down. Error code: " +
to_string(res) + "\n";
      //exit(1);
      return -1;
   }
   long recv = read(_socket, buffer, 512);
   if (recv < 0) {
      //std::cout << "An error occurred receiving info from the server, shutting down. Error code: "
+ to_string(recv) + "\n";
      //exit(1);
      return -1;
   }
   return atoi(buffer);
}


//=============================================================
// PREPARE
//=============================================================
void send_info(int info, int _socket)
{
   long res = write(_socket, (to_string(info)+"\0").c_str(), (size_t) to_string(info).length()+1);
   if (res < 0){
      std::cout << "An error occurred sending info to the server, shutting down. Error code: " +
to_string(res) + "\n";
      exit(1);
   }
   char buffer[512];
   long recv = read(_socket, buffer, 512);
   if (recv < 0) {
      std::cout << "An error occurred receiving info to the server, shutting down. Error code: " +
to_string(recv) + "\n";
      exit(1);
   }
}
void prepare_server(int wl, int index_type, int num_thread, int repeat_counter)
{
```

```
    printf("\nWant to send: %d %d %d %d %d %d %d %d %d\n", wl, index_type, num_thread,
insert_only, hyperthreading, memory_bandwidth, numa, repeat_counter, del);

    int _socket = setup_connection(server_address, 11211);

    send_info(wl, _socket);
    send_info(index_type, _socket);
    send_info(num_thread, _socket);
    send_info(insert_only, _socket);
    send_info(numa, _socket);
    send_info(hyperthreading, _socket);
    send_info(memory_bandwidth, _socket);
    send_info(repeat_counter, _socket);
    send_info(del, _socket);

    for (int i = 0; i < num_thread; i++)
        _sockets.emplace_back(setup_connection(server_address, 11211));

    close(_socket);
}


//================================================================
// EXECUTE
//================================================================
void* execute_init_load(void *thread_args)
{
    const int options = 2;
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++) {
        ret[i] = 0;
    }

    auto *thread_arg = (struct thread_arg_t*)thread_args;

    int share_size = thread_arg->end_i - thread_arg->start_i;
    send_info(share_size, _sockets[thread_arg->tr_idx]);

    auto s = std::chrono::high_resolution_clock::now();

    for (size_t i = thread_arg->start_i; i < thread_arg->end_i; i++){
        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            s = std::chrono::high_resolution_clock::now();
```

```
        int res = send_query(_sockets[thread_arg->tr_idx], tuples[i].key, tuples[i].value);
        ///Handling server feedback////////////////////////////////////////////////////////////////////////////////////

        if (res == BWTREE_STORED){
            ret[0]++;
        }
        else if (res == BWTREE_NOTSTORED){
            std::cout << "Op-key " + tuples[i].key + " with value: " + tuples[i].value + " was
unsuccessful, probably a duplicate-key.\n";
            ret[1]++;
        }
        else {
            //std::cout << "An unknown error happened during initial insertion, exiting...";
            //exit(1);
            ret[1]++;
        }
        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            printf("Update operation took: %.9f sec\n", (std::chrono::duration<double,
std::milli>(std::chrono::high_resolution_clock::now() - s).count())/1000);
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    }
    delete(thread_arg);

    return (void*) ret;
}


void* execute_delete_load(void *thread_args)
{
    const int options = 2;
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++)
        ret[i] = 0;

    auto *thread_arg = (struct thread_arg_t*) thread_args;

    int share_size = thread_arg->end_i - thread_arg->start_i;
    send_info(share_size, _sockets[thread_arg->tr_idx]);

    for (size_t i = thread_arg->start_i; i < thread_arg->end_i; i++) {
        int res = send_query(_sockets[thread_arg->tr_idx], delete_tuples[i].key,
delete_tuples[i].value);
```

```
        ///Handling server feedback////////////////////////////////////////////////////////////////////////////////////

        if (res == BWTREE_DELETED) {
            ret[0]++;
        } else if (res == BWTREE_NOTDELETED){
            std::cout << "Operation " + delete_tuples[i].key + " on key : " + delete_tuples[i].value + "
was unsuccessful, key was probably already deleted by a previous delete-operation.\n";
            ret[1]++;
        } else {
            std::cout << "An unknown error happened during deletions, exiting...";
            exit(1);
        }
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////

    }
    delete (thread_arg);

    return (void *) ret;
}


void* execute_extra_load(void *thread_args)
{
    const int options = 4*2; //4 operation types * 2 (success/failure)
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++)
        ret[i] = 0;

    auto *thread_arg = (struct thread_arg_t *) thread_args;

    int share_size = thread_arg->end_i - thread_arg->start_i;
    send_info(share_size, _sockets[thread_arg->tr_idx]);

    auto s = std::chrono::high_resolution_clock::now();

    int fail_counter = 0;

    for (size_t i = thread_arg->start_i; i < thread_arg->end_i; i++) {
        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            s = std::chrono::high_resolution_clock::now();

        int res = send_query(_sockets[thread_arg->tr_idx], extr_tuples[i].key, extr_tuples[i].value);
        ///Handling server feedback////////////////////////////////////////////////////////////////////////////////////
```

```
        if (res == BWTREE_STORED) //insert success
            ret[0]++;
        else if (res == BWTREE_READ) //read success
            ret[2]++;
        else if (res == BWTREE_UPDATED) //update success
            ret[4]++;
        else if (res == BWTREE_DELETED) //delete success
            ret [6]++;

        else if (res == BWTREE_NOTSTORED){ //insert failed
            std::cout << "Op-key " + extr_tuples[i].key + " with value: " + extr_tuples[i].value + " was
unsuccessful, probably a duplicate-key.\n";
            ret[1]++;
        } else if (res == BWTREE_NOTREAD){ //read failed
            std::cout << "Op-key: " + extr_tuples[i].key + " was unsuccessful, key does probably not
exist in the database.\n";
            ret[3]++;
        } else if (res == BWTREE_NOTUPDATED){ //update failed
            std::cout << "Op-key: " + extr_tuples[i].key + " with value " + extr_tuples[i].value + " was
unsuccessful, key does probably not exist in the database.\n";
            ret[5]++;
        } else if (res == BWTREE_NOTDELETED){ //delete failed
            std::cout << "Op-key: " + extr_tuples[i].key + " was unsuccessful, key does probably not
exist in the database.\n";
            ret[7]++;
        } else {
            //std::cout << "Something went wrong; error code: " + to_string(res) + "; Wanted to
execute " + extr_tuples[i].key + "\n";
            //exit(1);
            fail_counter++;
        }
        if (fail_counter >= 10){
            printf("To many errors, abort iteration ... \n");
            break;
        }

        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            printf("Update operation took: %.9f sec\n", (std::chrono::duration<double,
std::milli>(std::chrono::high_resolution_clock::now() - s).count())/1000);
        ////////////////////////////////////////////////////////////////////////////////////////////////////
    }
    delete (thread_arg);
```

```
    return (void *) ret;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//============================================================
// LOAD
//============================================================
inline void load(int wl, int kt, int index_type, int ins, int ext)
{

    std::string init_file;
    std::string txn_file;
    std::string delete_file;

    if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 10 && ext == 10){
//10/10
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/10mil/10mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/10mil/10mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 50){
//10/50
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/50mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 50){
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/50mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 50){
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadmono.dat";
```

```
        txn_file = "../index-microbench-master/workloads/10mil/50mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 50){
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/50mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 100){
//10/100
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/100mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 100){
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/100mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 100){
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/100mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 100){
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/100mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 200){
//10/200
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 200){
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 200){
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 200){
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/monoupdate.dat";


    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 50 && ext == 10){
//50/10
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/50mil/10mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 50 && ext == 10){
```

```
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/50mil/10mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 50 && ext == 50){
//50/50
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/50mil/50mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/50mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/50mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/50mil/50mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/50mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/50mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 100){
//50/100
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 100){
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 100){
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 100){
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 200){
//50/200
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadrand.dat";
```

```
        txn_file = "../index-microbench-master/workloads/50mil/200mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 200){
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/200mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 200){
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/200mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 200){
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/200mil/monoupdate.dat";


    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 10){
//100/10
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/100mil/10mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/10mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/10mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/100mil/10mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/10mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/10mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 50){
//100/50
        init_file = "../index-microbench-master/workloads/100mil/50mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/100mil/50mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 50){
        init_file = "../index-microbench-master/workloads/100mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/50mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 50){
        init_file = "../index-microbench-master/workloads/100mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/50mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 50){
        init_file = "../index-microbench-master/workloads/100mil/50mil/loadmono.dat";
```

```
        delete_file = "../index-microbench-master/workloads/100mil/50mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 50){
        init_file = "../index-microbench-master/workloads/100mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/50mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 50){
        init_file = "../index-microbench-master/workloads/100mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/50mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 100){
//100/100
        init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/100mil/100mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 100){
        init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/100mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 100){
        init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/100mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 100){
        init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/100mil/100mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 100){
        init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/100mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 100){
        init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/100mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 200){
//100/200
        init_file = "../index-microbench-master/workloads/100mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/200mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 200){
        init_file = "../index-microbench-master/workloads/100mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/200mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 200){
        init_file = "../index-microbench-master/workloads/100mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/200mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 200) {
        init_file = "../index-microbench-master/workloads/100mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/100mil/200mil/monoupdate.dat";
    }

    else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 10 && ext == 10) {
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
```

```
    txn_file = "../index-microbench-master/workloads/10mil/10mil/read_update_rand.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 50 && ext == 50) {
    init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/50mil/50mil/read_update_rand.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 100 && ext == 100) {
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/read_update_rand.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 10 && ext == 10) {
    init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/10mil/10mil/read_update_mono.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 50 && ext == 50) {
    init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/50mil/50mil/read_update_mono.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 100 && ext == 100) {
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/read_update_mono.dat";
} else {
    printf("Unknown file, aborting ...\n");
    exit(1);
}

std::ifstream infile_load(init_file);

std::string op;
std::string key;
int range;
std::string line;

std::string insert("INSERT");
std::string read("READ");
std::string update("UPDATE");
std::string scan("SCAN");
std::string remove("DELETE");

auto value = (uint64_t)&tuples;
printf("Loading init-file...\n");
while (infile_load >> op >> key) {

    if (op.compare(insert) != 0) {
        std::cout << "READING LOAD FILE FAIL!\n";
        return;
    }
    tuples.emplace_back(key, to_string(value++));
```

```
    }
    infile_load.close();

    // If we do not perform other transactions, we can skip txn file
    if(insert_only) {
        return;
    }

    if(!delete_file.empty()){
        printf("Loading delete-file...\n");
        std::ifstream infile_del(delete_file);

        while (infile_del >> op >> key) {
            if (op.compare(remove) == 0)
                delete_tuples.emplace_back((to_string(OP_REMOVE)+" "+key), ""); //Using "" here
will make bwtree.h return a list of all stored k,v-pairs-ish
            else {
                std::cout << "UNRECOGNIZED CMD!\n";
                infile_del.close();
                return;
            }
        }
        infile_del.close();
    }

    // If we also execute transaction then open the
    // transacton file here
    if(!txn_file.empty()){
        printf("Loading extras-file...\n");
        std::ifstream infile_txn(txn_file);
        auto extr_value = (uint64_t)&extr_tuples;

        while (infile_txn >> op >> key) {
            if (op.compare(insert) == 0)
                extr_tuples.emplace_back((to_string(OP_INSERT)+" "+key), to_string(extr_value++));
            else if (op.compare(read) == 0)
                extr_tuples.emplace_back((to_string(OP_READ)+" "+key), "0");
            else if (op.compare(update) == 0)
                extr_tuples.emplace_back((to_string(OP_UPSERT)+" "+key),
to_string(extr_value++));
            else if (op.compare(remove) == 0)
```

```
            extr_tuples.emplace_back((to_string(OP_REMOVE)+" "+key), ""); //Using "" here will
make bwtree.h return a list of all stored k,v-pairs-ish
        else {
            std::cout << "UNRECOGNIZED CMD!\n";
            infile_txn.close();
            return;
        }
    }

    infile_txn.close();
  }
  //printf("\n\nLength 1: %lu, length 2: %lu\n\n", tuples.size(), extr_tuples.size());
  return;
}

//===============================================================
// EXEC
//===============================================================
inline void exec(int num_thread)
{

  //WRITE ONLY TEST----------------
  int count = (int)tuples.size();
  fprintf(stderr, "Populating the index with %d keys using %d threads\n", count, num_thread);

  auto start = std::chrono::high_resolution_clock::now();

  void *ret[num_thread];
  unsigned int loc_res[12];
  for (unsigned int &loc_re : loc_res)
    loc_re = 0;

  int share_size = static_cast<int>(ceil((double)tuples.size() / (double)num_thread));
  auto *threads = (u_long*) malloc(num_thread * sizeof(pthread_t));

  for (int i = 0; i < num_thread; i++)
  {
    auto *thread_arg = new thread_arg_t;

    thread_arg->tr_idx = i;

    thread_arg->start_i = i * share_size;
```

```
        if (i != num_thread -1)
        {
            thread_arg->end_i = thread_arg->start_i + share_size;
        } else {
            thread_arg->end_i = (int)tuples.size();
        }

        pthread_create(&threads[i], NULL, execute_init_load, thread_arg);
    }

    for (size_t i = 0; i < num_thread; i++){
        pthread_join(threads[i], &ret[i]);
        auto *var = reinterpret_cast<unsigned int *>(ret[i]);
        loc_res[0] += var[0];
        loc_res[1] += var[1];
    }


    auto end = std::chrono::high_resolution_clock::now();
    init_results.emplace_back(loc_res[0], loc_res[1], (std::chrono::duration<double, std::milli>(end
- start).count())/1000);

    printf("Inserts iterasjon it: %d | successes: %d | failures: %d | time: %.2f sec\n", iteration,
loc_res[0], loc_res[1], (std::chrono::duration<double, std::milli>(end - start).count())/1000);


    //printf("\nExecuting input file 1 took %f seconds.\n", (std::chrono::duration<double,
std::milli>(end - start).count())/1000);
    //printf("Succeeded operations: %u, Failed operations: %u\n\n", all_res[iteration][0],
all_res[iteration][1]);
    if (loc_res[0] + loc_res[1] != tuples.size()) {
        printf("An error happened. The program handled %d records, when there are supposed to
be %lu\n", loc_res[0] + loc_res[1],
               tuples.size());
    }

    if (insert_only){
        free(threads);
        return;
    }

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    ////DELETE PART START////////////////////////////////////////////////////////////////////////////////
```

```
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    if (del){
        start = std::chrono::high_resolution_clock::now();
        share_size = static_cast<int>(ceil((double)delete_tuples.size() / (double)num_thread));

        printf("\nNumber of keys to delete: %lu\n", delete_tuples.size());

        for (int i = 0; i < num_thread; i++)
        {
            auto *thread_arg = new thread_arg_t;

            thread_arg->tr_idx = i;

            thread_arg->start_i = i * share_size;

            if (i != num_thread -1)
            {
                thread_arg->end_i = thread_arg->start_i + share_size;
            } else {
                thread_arg->end_i = (int)delete_tuples.size();
            }

            pthread_create(&threads[i], NULL, execute_delete_load, thread_arg);
        }

        for (size_t i = 0; i < num_thread; i++){
            pthread_join(threads[i], &ret[i]);
            auto *var = reinterpret_cast<unsigned int *>(ret[i]);
            //printf("threadnr: %zu, var0: %u, var1: %u\n", i, var[0], var[1]);
            loc_res[2] += var[0];
            loc_res[3] += var[1];
        }

        end = std::chrono::high_resolution_clock::now();
        del_results.emplace_back(loc_res[2], loc_res[3], (std::chrono::duration<double,
std::milli>(end - start).count())/1000);
        //printf("\nExecuting input file 2 took %f seconds.\n", (std::chrono::duration<double,
std::milli>(end - start).count())/1000);
        //printf("Succeeded operations: %u, Failed operations: %u\n\n", all_res[iteration][2],
all_res[iteration][3]);
        if (loc_res[2] + loc_res[3] != delete_tuples.size()) {
```

```
            printf("An error happened. The program handled %d records, when there are supposed
to be %lu\n", loc_res[2] + loc_res[3],
                    delete_tuples.size());
        }

    }
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    ////DELETE PART END////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////

    start = std::chrono::high_resolution_clock::now();
    share_size = static_cast<int>(ceil((double)extr_tuples.size() / (double)num_thread));

    printf("\nNumber of additional operations: %lu\n", extr_tuples.size());

    for (int i = 0; i < num_thread; i++)
    {
        auto *thread_arg = new thread_arg_t;

        thread_arg->tr_idx = i;

        thread_arg->start_i = i * share_size;

        if (i != num_thread -1)
        {
            thread_arg->end_i = thread_arg->start_i + share_size;
        } else {
            thread_arg->end_i = (int)extr_tuples.size();
        }

        pthread_create(&threads[i], NULL, execute_extra_load, thread_arg);
    }

    for (size_t i = 0; i < num_thread; i++) {
        pthread_join(threads[i], &ret[i]);
        auto *var = reinterpret_cast<unsigned int *>(ret[i]);
        for (int j = 0; j < 8; j++)
            loc_res[j+4] += var[j];
    }
    free(threads);

    end = std::chrono::high_resolution_clock::now();
```

```
    extr_results.emplace_back(loc_res[4], loc_res[5], loc_res[6], loc_res[7], loc_res[8], loc_res[9],
loc_res[10], loc_res[11], (std::chrono::duration<double, std::milli>(end - start).count())/1000);

    printf("Read/Update iteration %d: INS_S: %d | INS_F: %d | READ_S: %d | READ_F: %d |
UPD_S: %d | UPD_F: %d | DEL_S: %d | DEL_F: %d | took %.2f sec\n",
           iteration, loc_res[4], loc_res[5], loc_res[6], loc_res[7], loc_res[8], loc_res[9], loc_res[10],
loc_res[11], (std::chrono::duration<double, std::milli>(end - start).count())/1000);

    if (loc_res[4] + loc_res[5] + loc_res[6] + loc_res[7] + loc_res[8] + loc_res[9] + loc_res[10] +
loc_res[11] != extr_tuples.size()) {
        printf("An error happened. The program handled %d extra records, when there are
supposed to be %lu", loc_res[4] + loc_res[5] + loc_res[6] + loc_res[7] + loc_res[8] + loc_res[9] +
loc_res[10] + loc_res[11],
               extr_tuples.size());
    }

    return;
}

/*
 * run_rdtsc_benchmark() - This function runs the RDTSC benchmark which is a high
 *                         contention insert-only benchmark
 *
 * Note that key num is the total key num
 */
void run_rdtsc_benchmark(int index_type, int thread_num, int key_num)
{
    Index<keytype, keycomp> *idx = getInstance<keytype, keycomp>(index_type, key_type);

    auto func = [idx, thread_num, key_num](uint64_t thread_id, bool) {
        size_t key_per_thread = key_num / thread_num;

        threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

        uint64_t *values = new uint64_t[key_per_thread];

        int gc_counter = 0;
        for(size_t i = 0;i < key_per_thread;i++) {
            // Note that RDTSC may return duplicated keys from different cores
            // to counter this we combine RDTSC with thread IDs to make it unique
            // The counter value on a single core is always unique, though
            uint64_t key = (Rdtsc() << 6) | thread_id;
            values[i] = key;
```

```
            //fprintf(stderr, "%lx\n", key);
            idx->insert(key, reinterpret_cast<uint64_t>(values + i), ti);
            gc_counter++;
            if(gc_counter % 4096 == 0) {
               ti->rcu_quiesce();
            }
         }

         ti->rcu_quiesce();

         delete [] values;

         return;
      };

      if(numa == true) {
         PCM_NUMA::StartNUMAMonitor();
      }

      double start_time = get_now();
      StartThreads(idx, thread_num, func, false);
      double end_time = get_now();

      if(numa == true) {
         PCM_NUMA::EndNUMAMonitor();
      }

      // Only execute consolidation if BwTree delta chain is used
#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
      idx->AfterLoadCallback();
#endif

      double tput = key_num * 1.0 / (end_time - start_time) / 1000000; //Mops/sec
      std::cout << "insert " << tput << "\n";

      return;
}


int main(int argc, char *argv[])
{
      if (argc < 6) {
         std::cout << "Usage:\n";
```

```
    std::cout << "1. workload type: r, u, d,\n";
    std::cout << "2. key distribution: rand, mono\n";
    std::cout << "3. index type: bwtree skiplist masstree btreeolc btreertm\n";
    std::cout << "4. number of keys to insert in millions (10, 50, 100)\n";
    std::cout << "5. number of extra operations to execute in millions (10, 50, 100, 200)\n";
    std::cout << "6. number of threads (integer)\n";
    std::cout << "7 extra settings\n";
    std::cout << "   --hyper: Whether to pin all threads on NUMA node 0\n";
    std::cout << "   --mem: Whether to monitor memory access\n";
    std::cout << "   --numa: Whether to monitor NUMA throughput\n";
    std::cout << "   --insert-only: Whether to only execute insert operations\n";

    return 1;
}

// Then read the workload type
int wl;
if (strcmp(argv[1], "r") == 0) {
    wl = WORKLOAD_R;
} else if (strcmp(argv[1], "u") == 0) {
    wl = WORKLOAD_U;
} else if (strcmp(argv[1], "d") == 0) {
    wl = WORKLOAD_D;
    del = true;
} else if (strcmp(argv[1], "ru") == 0) {
    wl = WORKLOAD_RU;
} else {
    fprintf(stderr, "Unknown workload: %s\n", argv[1]);
    exit(1);
}

// Then read key type
int kt;
if (strcmp(argv[2], "rand") == 0) {
    kt = RAND_KEY;
} else if (strcmp(argv[2], "mono") == 0) {
    kt = MONO_KEY;
} else if (strcmp(argv[2], "rdtsc") == 0) {
    kt = RDTSC_KEY;
} else {
    fprintf(stderr, "Unknown key type: %s\n", argv[2]);
    exit(1);
}
```

```
int index_type;
if (strcmp(argv[3], "bwtree") == 0)
    index_type = TYPE_BWTREE;
else if (strcmp(argv[3], "masstree") == 0)
    index_type = TYPE_MASSTREE;
else if (strcmp(argv[3], "btreeolc") == 0)
    index_type = TYPE_BTREEOLC;
else if (strcmp(argv[3], "skiplist") == 0)
    index_type = TYPE_SKIPLIST;
else if (strcmp(argv[3], "btreertm") == 0)
    index_type = TYPE_BTREERTM;
else if (strcmp(argv[3], "none") == 0)
    // This is a special type used for measuring base cost (i.e.
    // only loading the workload files but do not invoke the index)
    index_type = TYPE_NONE;
else {
    fprintf(stderr, "Unknown index type: %d\n", index_type);
    exit(1);
}

// Read amount insert ops in million
int insert_ops;
if (strcmp(argv[4], "10") == 0) {
    insert_ops = 10;
} else if (strcmp(argv[4], "50") == 0) {
    insert_ops = 50;
} else if (strcmp(argv[4], "100") == 0) {
    insert_ops = 100;
} else {
    fprintf(stderr, "Unknown amount of inserts: %s\n", argv[4]);
    exit(1);
}

// Read amount extra ops in million
int extra_ops;
if (strcmp(argv[5], "10") == 0) {
    extra_ops = 10;
} else if (strcmp(argv[5], "50") == 0) {
    extra_ops = 50;
} else if (strcmp(argv[5], "100") == 0) {
    extra_ops = 100;
} else if (strcmp(argv[5], "200") == 0) {
```

```
    extra_ops = 200;
} else {
    fprintf(stderr, "Unknown amount of extras: %s\n", argv[5]);
    exit(1);
}

// Then read number of threads using command line
int num_thread = atoi(argv[6]);
if(num_thread < 1 || num_thread > 96) {
    fprintf(stderr, "Do not support %d threads\n", num_thread);
    exit(1);
} else {
    fprintf(stderr, "Number of threads: %d\n", num_thread);
}

// Then read all remianing arguments
int repeat_counter = 1;
char **argv_end = argv + argc;
for(char **v = argv + 7;v != argv_end;v++) {
    if(strcmp(*v, "--hyper") == 0) {
        // Enable hyoerthreading for scheduling threads
        hyperthreading = true;
    } else if(strcmp(*v, "--mem") == 0) {
        // Enable memory bandwidth measurement
        memory_bandwidth = true;
    } else if(strcmp(*v, "--numa") == 0) {
        numa = true;
    } else if(strcmp(*v, "--insert-only") == 0) {
        insert_only = true;
    } else if(strcmp(*v, "--repeat") == 0) {
        // If we repeat, then exec() will be called for 5 times
        repeat_counter = 5;
    } else if(strcmp(*v, "--max-init-key") == 0) {
        max_init_key = atoll(*(v + 1));
        if(max_init_key <= 0) {
            fprintf(stderr, "Illegal maximum init keys: %ld\n", max_init_key);
            exit(1);
        }

        // Ignore the next argument
        v++;
    } else {
        fprintf(stderr, "Unknown switch: %s\n", *v);
```

```
            exit(1);
        }
    }

    if(max_init_key != -1) {
        fprintf(stderr, "Maximum init keys: %ld\n", max_init_key);
        fprintf(stderr, "  NOTE: Memory is not affected in this case\n");
    }

#ifdef COUNT_READ_MISS
    fprintf(stderr, "  Counting read misses\n");
#endif

#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
    fprintf(stderr, "  BwTree will considate after insert phase\n");
#endif

#ifdef USE_TBB
    fprintf(stderr, "  Using Intel TBB to run concurrent tasks\n");
#endif

#ifdef INTERLEAVED_INSERT
    fprintf(stderr, "  Interleaved insert\n");
#endif

#ifdef BWTREE_COLLECT_STATISTICS
    fprintf(stderr, "  BwTree will collect statistics\n");
#endif

    fprintf(stderr, "Leaf delta chain threshold: %d; Inner delta chain threshold: %d\n",
            LEAF_DELTA_CHAIN_LENGTH_THRESHOLD,
            INNER_DELTA_CHAIN_LENGTH_THRESHOLD);

#ifndef BWTREE_USE_MAPPING_TABLE
    fprintf(stderr, "  BwTree does not use mapping table\n");
  if(wl != WORKLOAD_C) {
    fprintf(stderr, "Could only use workload C\n");
    exit(1);
  }

  if(index_type != TYPE_BWTREE) {
    fprintf(stderr, "Could only use BwTree\n");
    exit(1);
```

```
  }
#endif

#ifndef BWTREE_USE_CAS
   fprintf(stderr, "  BwTree does not use CAS\n");
#endif

#ifndef BWTREE_USE_DELTA_UPDATE
   fprintf(stderr, "  BwTree does not use delta update\n");
  if(index_type != TYPE_BWTREE) {
   fprintf(stderr, "Could only use BwTree\n");
  }
#endif

#ifdef USE_OLD_EPOCH
   fprintf(stderr, "  BwTree uses old epoch\n");
#endif

   // If we do not interleave threads on two sockets then this will be printed
   if(hyperthreading == true) {
      fprintf(stderr, "  Hyperthreading for thread 10 - 19, 30 - 39\n");
   }

   if(repeat_counter != 1) {
      fprintf(stderr, "  Repeat for %d times (NOTE: Memory number may not be correct)\n",
            repeat_counter);
   }

   if(memory_bandwidth == true) {
      if(geteuid() != 0) {
         fprintf(stderr, "Please run the program as root in order to measure memory
bandwidth\n");
         exit(1);
      }

      fprintf(stderr, "  Measuring memory bandwidth\n");

      PCM_memory::InitMemoryMonitor();
   }

   if(numa == true) {
      if(geteuid() != 0) {
         fprintf(stderr, "Please run the program as root in order to measure NUMA operations\n");
```

```
        exit(1);
    }

    fprintf(stderr, "  Measuring NUMA operations\n");

    // Call init here to avoid calling it mutiple times
    PCM_NUMA::InitNUMAMonitor();
}

if(insert_only == true) {
    fprintf(stderr, "Program will exit after insert operation\n");
}

tuples.reserve(100000000);
extr_tuples.reserve(200000000);


fprintf(stderr, "  BTree element pair count: %lu\n",
        (uint64_t)btreeolc::BTreeLeaf<uint64_t, uint64_t>::maxEntries);

// If the key type is RDTSC we just run the special function
if(kt != RDTSC_KEY) {

    load(wl, kt, index_type, insert_ops, extra_ops);
    printf("Finished loading workload file (mem = %lu)\n", MemUsage());

    prepare_server(wl, index_type, num_thread, repeat_counter);

    if(index_type != TYPE_NONE) {
        // Then repeat executing the same workload

        while(iteration < repeat_counter) {

            exec(num_thread);
            iteration++;
            printf("\nFinished iteration %d of %d\n\n", iteration, repeat_counter);
            printf("Finished running benchmark (mem = %lu)\n", MemUsage());
        }
    } else {
        fprintf(stderr, "Type None is selected - no execution phase\n");
    }

    for (int i = 0; i < repeat_counter; i++){
```

```
            if (!insert_only && delete_tuples.empty()){
                printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec - part 2: INS_S: %d |
INS_F: %d | READ_S: %d | READ_F: %d | UPD_S: %d | UPD_F: %d | DEL_S: %d | DEL_F: %d
| took %.2f sec\n",
                        (i+1), init_results[i].init_s, init_results[i].init_f, init_results[i].t1,
                        extr_results[i].insert_s, extr_results[i].insert_f, extr_results[i].read_s,
extr_results[i].read_f, extr_results[i].update_s, extr_results[i].update_f, extr_results[i].delete_s,
extr_results[i].delete_f, extr_results[i].t2);
            } else if (!delete_tuples.empty()){
                printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec - part 2: DEL_S: %d |
DEL_F: %d | took %.2f sec - part 3 INS_S: %d | INS_F: %d | READ_S: %d | READ_F: %d |
UPD_S: %d | UPD_F: %d | DEL_S: %d | DEL_F: %d | took %.2f sec\n",
                        (i+1), init_results[i].init_s, init_results[i].init_f, init_results[i].t1, del_results[i].init_s,
del_results[i].init_f, del_results[i].t1,
                        extr_results[i].insert_s, extr_results[i].insert_f, extr_results[i].read_s,
extr_results[i].read_f, extr_results[i].update_s, extr_results[i].update_f, extr_results[i].delete_s,
extr_results[i].delete_f, extr_results[i].t2);
            } else {
                printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec\n",
                        (i+1), init_results[i].init_s, init_results[i].init_f, init_results[i].t1);
            }
        }

    } else {
        fprintf(stderr, "Running RDTSC benchmark...\n");
        run_rdtsc_benchmark(index_type, num_thread, 50 * 1000 * 1000);
    }

    for (int i = 0; i < num_thread; i++)
        close(_sockets[i]);

    exit_cleanup();

    return 0;
}
```