```
#ifndef MEMCACHEDINTERFACE_MAIN_HPP
#define MEMCACHEDINTERFACE_MAIN_HPP

// These are YCSB workloads
enum {
   WORKLOAD_R,
   WORKLOAD_U,
   WORKLOAD_D,
   WORKLOAD_RU,
};

// These are key types we use for running the benchmark
enum {
   RAND_KEY,
   MONO_KEY,
   RDTSC_KEY,
   EMAIL_KEY,
};

void reset_mysql();
void *execute_init_load(void *thread_args);
void *execute_extra_load(void *thread_args);
void *execute_delete_load(void *thread_args);

#endif //MEMCACHEDINTERFACE_MAIN_HPP
\end{verbatim}

\textbf{main.cpp}
\begin{verbatim}
#include <cstring>
#include <cstdlib>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <list>
#include <pthread.h>
#include <cmath>
#include "main.hpp"
#include <iostream>
#include <fstream>
#include <ctime>
#include <chrono>
```

```
#include <netinet/tcp.h>
#include <libmemcached/memcached.h>
#include <vector>
#include <mysql/mysql.h>
#include <unistd.h>
#include <cassert>

struct tuple_t {
    enum operation_t { INSERT, READ, UPDATE, DELETE };
    operation_t operation;
    std::string key;
    std::string value;

    tuple_t(const std::string &operation_arg, std::string key_arg, std::string value_arg)
    : key(std::move(key_arg)), value(std::move(value_arg))
    {
        if (operation_arg == "INSERT")
            operation = INSERT;
        else if (operation_arg == "READ")
            operation = READ;
        else if (operation_arg == "UPDATE")
            operation = UPDATE;
        else if (operation_arg == "DELETE")
            operation = DELETE;
        else
            assert(false);
    }

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


struct thread_arg_t{
    int start_i;
    int end_i;
};


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;
};

static std::vector<tuple_t> tuples;

static std::vector<tuple_t> extr_tuples;

static std::vector<std::string> del_keys;
```

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

//static const char *server_address = "frigg04.no.oracle.com";
//static const char *server_address = "10.172.139.128";
static const char *server_address = "loki08.no.oracle.com";
static const u_int16_t memcached_port = 11217;
static const u_int16_t mysqld_port = 13000;

bool init = true;
bool del = false;
bool insert_only = false;
bool extras_only = false;
bool inserted = false;
bool no_inserts = false;

void load_file(const std::string &path)
{
   if (insert_only && ((path.find("read") != std::string::npos) || (insert_only && path.find("update")
!= std::string::npos) || (insert_only && path.find("delete") != std::string::npos)))
       return;

   std::string line;
   std::string op;
   std::string key;

   std::ifstream file(path);

   auto value = (uint64_t)&tuples;
   auto extr_value = (uint64_t)&extr_tuples;

   if (file.is_open())
   {
      printf("Starting to load data.\n");
      while (file >> op >> key)
      {
         if (init){
            tuples.emplace_back(op, key, std::to_string(value++));
         } else if (del){
            del_keys.emplace_back(key);
         } else{
            extr_tuples.emplace_back(op, key, std::to_string(extr_value++));
```

```
            }
        }
        if (init){
            printf("Done loading init-data\n");
        }
        else if (del){
            printf("Done loading delete-data\n");
        }
        else {
            printf("Done loading extra-data\n");
        }

        file.close();
    } else {
        printf("Error, could not open file.\n");
        exit(1);
    }
    init = false;
}


void setup_mysql(bool retry)
{
    MYSQL *connection;
    connection = mysql_init(nullptr);

    if(!(mysql_real_connect(connection, server_address, "admin", "", "test", mysqld_port, nullptr,
0)))
    {
        fprintf(stderr, "\nError: %s [%d]\n", mysql_error(connection), mysql_errno(connection));
        exit(1);
    }
    printf("Connected to server! Preparing server ...\n");


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

    std::vector<int> mysql_feedback;
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    // Query_01
    mysql_feedback.emplace_back(mysql_query(connection, "create database
innodb_memcache"));
```

```
   // Query_02
   mysql_feedback.emplace_back(mysql_query(connection, "use innodb_memcache"));

   // Query_03
   mysql_feedback.emplace_back(mysql_query(connection, "CREATE  TABLE IF NOT EXISTS
`cache_policies` (\n"
                       "    `policy_name` VARCHAR(40) PRIMARY KEY,\n"
                       "    `get_policy` ENUM('innodb_only', 'cache_only', 'caching','disabled')\n"
                       "    NOT NULL ,\n"
                       "    `set_policy` ENUM('innodb_only', 'cache_only','caching','disabled')\n"
                       "    NOT NULL ,\n"
                       "    `delete_policy` ENUM('innodb_only', 'cache_only', 'caching','disabled')\n"
                       "    NOT NULL,\n"
                       "    `flush_policy` ENUM('innodb_only', 'cache_only', 'caching','disabled')\n"
                       "    NOT NULL\n"
                       "    ) ENGINE = innodb"));

   // Query_04
   mysql_feedback.emplace_back(mysql_query(connection, "CREATE  TABLE IF NOT EXISTS
`containers` (\n"
                       "    `name` varchar(50) not null primary key,\n"
                       "    `db_schema` VARCHAR(250) NOT NULL,\n"
                       "    `db_table` VARCHAR(250) NOT NULL,\n"
                       "    `key_columns` VARCHAR(250) NOT NULL,\n"
                       "    `value_columns` VARCHAR(250),\n"
                       "    `flags` VARCHAR(250) NOT NULL DEFAULT \"0\",\n"
                       "    `cas_column` VARCHAR(250),\n"
                       "    `expire_time_column` VARCHAR(250),\n"
                       "    `unique_idx_name_on_key` VARCHAR(250) NOT NULL\n"
                       "    ) ENGINE = InnoDB"));

   // Query_05
   mysql_feedback.emplace_back(mysql_query(connection, "CREATE  TABLE IF NOT EXISTS
`config_options` (\n"
                       "    `name` varchar(50) not null primary key,\n"
                       "    `value` varchar(50)) ENGINE = InnoDB"));

   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
   // Query_06
   mysql_feedback.emplace_back(mysql_query(connection,
            R"(INSERT INTO cache_policies VALUES("cache_policy", "innodb_only",
"innodb_only", "innodb_only", "innodb_only"))"));
```

```
    // Query_07
    mysql_feedback.emplace_back(mysql_query(connection, R"(INSERT INTO config_options
VALUES("separator", "|"))"));

    // Query_08
    mysql_feedback.emplace_back(mysql_query(connection, R"(INSERT INTO containers
VALUES ("desc_t1", "test", "t1", "c1", "c2",  "c3", "c4", "c5", "PRIMARY"))"));

    // Query_09
    mysql_feedback.emplace_back(mysql_query(connection, "USE test"));

    // Query_10
    mysql_feedback.emplace_back(mysql_query(connection, "CREATE TABLE t1 (c1
VARCHAR(32), c2 VARCHAR(1024), c3 INT, c4 BIGINT UNSIGNED, c5 INT, primary key(c1))
ENGINE = INNODB"));

    // Query_11
    mysql_feedback.emplace_back(mysql_query(connection, "INSTALL PLUGIN
daemon_memcached SONAME 'libmemcached.so'"));
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    int failure = 0;
    for (int i = 0; i < (int)mysql_feedback.size(); i++) {
        if (mysql_feedback[i] != 0){
            std::cout << "An error occurred during server setup on operation nr: " +
std::to_string(i+1) + " with result " + std::to_string(mysql_feedback[i]) + ". Retrying ...\n";
            failure = 1;
            break;
        }
    }
    mysql_close(connection);

    if (failure){
        reset_mysql();
        if (!retry)
            setup_mysql(true);

        else if(retry) {
            printf("Can't connect. Aborting...\n");
            exit(1);
        }
    }
```

```
    usleep(1000000);
}


void reset_mysql()
{
    printf("Resetting...\n");
    MYSQL *connection;
    connection = mysql_init(nullptr);

    if(!(mysql_real_connect(connection, server_address, "admin", "", "test", mysqld_port, nullptr,
0)))
    {
        fprintf(stderr, "\nError: %s [%d]\n", mysql_error(connection), mysql_errno(connection));
        return;
        //exit(1);
    }

    mysql_query(connection, "DROP TABLE t1");
    mysql_query(connection, "UNINSTALL PLUGIN daemon_memcached");
    mysql_query(connection, "DROP DATABASE innodb_memcache");
    mysql_close(connection);
}


static memcached_st* setup_memcached()
{
    std::string server_string;
    server_string.append("--SERVER=");
    server_string.append(server_address);
    server_string.append(":");
    server_string.append(std::to_string(memcached_port));
    memcached_st *server= memcached(server_string.c_str(), server_string.length());
    memcached_behavior_set(server, MEMCACHED_BEHAVIOR_RCV_TIMEOUT, 3000);
    memcached_behavior_set(server, MEMCACHED_BEHAVIOR_POLL_TIMEOUT, 3000);

    return server;
}


void choose_files(int wl, int kt, int ins, int ext)
{
```

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

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


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

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


    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 10){
//100/10
        load_file("../../index-microbench-master/workloads/100mil/10mil/loadrand.dat");
```

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

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

```
}


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. number of keys to insert in millions (10, 50, 100)\n";
        std::cout << "4. number of extra operations to execute in millions (10, 50, 100, 200)\n";
        std::cout << "5. number of threads (integer)\n";
        std::cout << "6 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;
    }

    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);
}


// Read amount insert ops in million
int insert_ops;
if (strcmp(argv[3], "10") == 0) {
    insert_ops = 10;
} else if (strcmp(argv[3], "50") == 0) {
    insert_ops = 50;
} else if (strcmp(argv[3], "100") == 0) {
    insert_ops = 100;
} else {
    fprintf(stderr, "Unknown amount of inserts: %s\n", argv[3]);
    exit(1);
}

// Read amount extra ops in million
int extra_ops;
if (strcmp(argv[4], "10") == 0) {
    extra_ops = 10;
} else if (strcmp(argv[4], "50") == 0) {
    extra_ops = 50;
} else if (strcmp(argv[4], "100") == 0) {
    extra_ops = 100;
} else if (strcmp(argv[4], "200") == 0) {
    extra_ops = 200;
} else {
    fprintf(stderr, "Unknown amount of extras: %s\n", argv[4]);
    exit(1);
}

size_t num_threads = atoi(argv[5]);

// Then read all remianing arguments
int repeat_counter = 1;
char **argv_end = argv + argc;
```

```
for(char **v = argv + 6;v != argv_end;v++) {
    if(strcmp(*v, "--insert-only") == 0) {
        insert_only = true;
    } else if(strcmp(*v, "--repeat") == 0) {
        repeat_counter = 5;
    } else if(strcmp(*v, "--extras-only") == 0) {
        extras_only = true;
    } else if(strcmp(*v, "--no-inserts") == 0) {
        no_inserts = true;
    } else {
            fprintf(stderr, "Unknown switch: %s\n", *v);
            exit(1);
    }

}

tuples.reserve(100000000);
if (del){
    del_keys.reserve(100000000);

    extr_tuples.reserve(0);
}
else {
    extr_tuples.reserve(200000000);
}

choose_files(wl, kt, insert_ops, extra_ops);

//printf("len1: %lu\n", tuples.size());
//printf("len1: %lu\n", extr_tuples.size());

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
auto *threads = (u_long *) malloc(num_threads * sizeof(pthread_t));

if (num_threads > tuples.size() && !tuples.empty()) {
    num_threads = tuples.size();
} else if (num_threads > extr_tuples.size() && tuples.empty()){
    num_threads = extr_tuples.size();
}
```

```
    if (num_threads == 0){
        printf("No threads remaining, aborting ...\n");
        exit(1);
    }

    int iteration = 0;

    while (iteration < repeat_counter){

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

        void *ret[num_threads];
        unsigned int loc_res[12];

        for (unsigned int &loc_re : loc_res)
            loc_re = 0;

        if (!inserted){

            if (!no_inserts)
                setup_mysql(false);

            printf("Populating the index with %lu keys using %zu threads\n", tuples.size(),
num_threads);
            start = std::chrono::high_resolution_clock::now();
            int share_size = static_cast<int>(ceil((double)tuples.size() / (double)num_threads));

            for (size_t i = 0; i < num_threads; i++) {
                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                auto *thread_arg = new thread_arg_t;
                thread_arg->start_i = i * share_size;

                if (i != num_threads - 1)
                    thread_arg->end_i = thread_arg->start_i + share_size;
                else
                    thread_arg->end_i = tuples.size();

                /*
                for (tuple_t &tuple : tuples) {
                    if (std::stoll(tuple.key) % num_threads == i)
                        thread_arg->tuples.push_back(tuple);
```

```
            }

            printf("Thread %zu has %lu operations to insert.\n", i, thread_arg->tuples.size());
            */
            printf("Thread %zu has %i operations to insert.\n", i, share_size);
            pthread_create(&threads[i], nullptr, execute_init_load, thread_arg);
            ////////////////////////////////////////////////////////////////////////////////////////////////////////
        }

        for (size_t i = 0; i < num_threads; 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];
            delete[] (var);
        }

        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);

        if (loc_res[0] + loc_res[1] != tuples.size() && !no_inserts) {
            printf("An error happened. The program handled %d records, when there are
supposed to be %lu", loc_res[0] + loc_res[1],
                   tuples.size());
        }

        if (extras_only){
            inserted = true;
            for (int i = 0; i < repeat_counter - 1; i++)
                init_results.emplace_back(0, 0, 0);
        }

        if (insert_only){
            iteration++;
            printf("\nFinished iteration %d of %d\n\n", iteration, repeat_counter);
            reset_mysql();
            continue;
        }
    }
    ////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////DELETE PART START////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
    if (!del_keys.empty()){
        start = std::chrono::high_resolution_clock::now();

        printf("\nNumber of keys to delete: %lu\n", del_keys.size());
        int share_size = static_cast<int>(ceil((double)del_keys.size() / (double)num_threads));

        for (size_t i = 0; i < num_threads; i++) {
            ////////////////////////////////////////////////////////////////////////////////////////////////////////////

            auto *thread_arg = new thread_arg_t;

            thread_arg->start_i = i * share_size;

            if (i != num_threads - 1)
                thread_arg->end_i = thread_arg->start_i + share_size;
            else
                thread_arg->end_i = del_keys.size();

            /*
            for (std::string &del_key : del_keys) {
                if (std::stoll(del_key) % num_threads == i)
                    thread_arg->tuples.emplace_back("DELETE", del_key, "");
            }
            printf("Thread %zu has %lu operations to delete.\n", i, thread_arg->tuples.size());
            */
            printf("Thread %zu has %lu operations to delete.\n", i, del_keys.size());
            pthread_create(&threads[i], nullptr, execute_delete_load, thread_arg);
            ////////////////////////////////////////////////////////////////////////////////////////////////////////////
        }

        for (size_t i = 0; i < num_threads; i++) {
            pthread_join(threads[i], &ret[i]);
            auto *var = reinterpret_cast<unsigned int *>(ret[i]);
            for (int j = 0; j < 2; j++)
                loc_res[j+2] += var[j];
            delete[] (var);
        }
        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);

        if (loc_res[2] + loc_res[3] != del_keys.size()) {
```

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

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

    if (extr_tuples.size() == 0){
        printf("No extra operations, should not be here, error in code. Aborting ...\n");
        exit(1);
    }

    printf("\nNumber of additional operations: %lu\n", extr_tuples.size());
    int share_size = static_cast<int>(ceil((double)extr_tuples.size() / (double)num_threads));

    for (size_t i = 0; i < num_threads; i++) {
        ////////////////////////////////////////////////////////////////////////////////////////////

        auto *thread_arg = new thread_arg_t;

        thread_arg->start_i = i * share_size;

        if (i != num_threads - 1)
            thread_arg->end_i = thread_arg->start_i + share_size;
        else
            thread_arg->end_i = extr_tuples.size();

        /*
        for (tuple_t &tuple : extr_tuples) {
            if (std::stoll(tuple.key) % num_threads == i)
                thread_arg->tuples.push_back(tuple);
        }
        printf("Thread %zu has %lu operations to work on.\n", i, thread_arg->tuples.size());
        */
        printf("Thread %zu has %i operations to work on.\n", i, thread_arg->end_i -
thread_arg->start_i);
        pthread_create(&threads[i], nullptr, execute_extra_load, thread_arg);
        ////////////////////////////////////////////////////////////////////////////////////////////
    }
```

```
        for (size_t i = 0; i < num_threads; 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];
            delete[] (var);
        }
        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);

        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());
        }

        iteration++;
        printf("\nFinished iteration %d of %d\n\n", (iteration), repeat_counter);
        if (!extras_only)
            reset_mysql();
    }

    free(threads);

    for (int i = 0; i < repeat_counter; i++){
        if (!insert_only && del_keys.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 (!del_keys.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\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);
```

```
        } /*else if (extras_only){
            printf("Iter%d 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), 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);
        }
    }
}


inline memcached_return_t send_query(memcached_st *memcached, tuple_t::operation_t op,
const std::string &key, const std::string &val)
{
    if (op == tuple_t::INSERT){
        memcached_return_t result = memcached_add(memcached, key.c_str(), key.length(),
val.c_str(), val.length(), 0, 0);
        if (result == MEMCACHED_SUCCESS)
            return MEMCACHED_STORED;
        return result;
    }
    else if (op == tuple_t::READ){
        size_t ret_val_len;
        uint32_t flags;
        memcached_return_t ret;
        const char* res = memcached_get(memcached, key.c_str(), key.length(), &ret_val_len,
&flags, &ret);

        if (res != nullptr){
            delete res;
            return MEMCACHED_SUCCESS;
        }
        delete res;
        return MEMCACHED_READ_FAILURE;
    }
    else if (op == tuple_t::UPDATE){
        memcached_return_t result = memcached_replace(memcached, key.c_str(), key.length(),
val.c_str(), val.length(), 0, 0);
        if (result == MEMCACHED_SUCCESS)
            return MEMCACHED_DATA_EXISTS;
```

```
        else if (result == MEMCACHED_NOTSTORED)
            return MEMCACHED_DATA_DOES_NOT_EXIST;
        return result;
    }
    else if (op == tuple_t::DELETE){
        memcached_return_t result = memcached_delete(memcached, key.c_str(), key.length(),
0);
        if (result == MEMCACHED_SUCCESS || result == MEMCACHED_TIMEOUT)
            return MEMCACHED_DELETED;
        else if (result != MEMCACHED_FAILURE)
            return result;
        return MEMCACHED_NOTFOUND;
    }
    else {
        printf("Op %d failed.\n", op);
        return MEMCACHED_NOT_SUPPORTED;
    }
}


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;
    }

    int res[60];
    for (int &re : res)
        re = 0;

    int ops_counter = 0;
    auto *thread_arg = (struct thread_arg_t*)thread_args;
    memcached_st* memcached = setup_memcached();

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

    if (!no_inserts){

        for (int 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();
```

```
        memcached_return_t result = send_query(memcached, tuples[i].operation, tuples[i].key,
tuples[i].value);
        ///Handling server feedback////////////////////////////////////////////////////////////////////////////////
        if (result == MEMCACHED_STORED){ //insert success
            ret[0]++;
        }
        else if (result == MEMCACHED_NOTSTORED){ //insert failed
            if (ops_counter % 400000 == 0)
                std::cout << "Operation " << tuples[i].operation << " on key-value-pair: " +
tuples[i].key + ": " + tuples[i].value + " was unsuccessful, probably a duplicate-key.\n";
            ret[1]++;
        } else {
            //std::cerr << "Something went wrong, error code \"" <<
memcached_strerror(memcached, result) << "\" from Memcachedlib; Wanted to " <<
tuple.operation << " " + tuple.key + "\n";
            res[result]++;
        }
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////

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

        ops_counter++;
        if(ops_counter % 500000 == 0){
            printf("Thread handled %.2f%% of its insert-operations, %d of %i.\n",
((float(ops_counter)*100.0)/float(thread_arg->end_i - thread_arg->start_i)), ops_counter,
thread_arg->end_i - thread_arg->start_i);
        }
      }
   }

   memcached_flush_buffers(memcached);
   memcached_free(memcached);
   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;

    int res[60];
    for (int &re : res)
        re = 0;

    int ops_counter = 0;
    auto *thread_arg = (struct thread_arg_t*)thread_args;
    memcached_st* memcached = setup_memcached();

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

    for (int 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();

        memcached_return_t result = send_query(memcached, extr_tuples[i].operation,
extr_tuples[i].key, extr_tuples[i].value);

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

        if (result == MEMCACHED_STORED) //insert success
            ret[0]++;
        else if (result == MEMCACHED_SUCCESS) //read success
            ret[2]++;
        else if (result == MEMCACHED_DATA_EXISTS) //update success
            ret[4]++;
        else if (result == MEMCACHED_DELETED) //delete success
            ret [6]++;

        else if (result == MEMCACHED_NOTSTORED){ //insert failed
            //std::cout << "Operation " + extr_ops[i] + " on key-value-pair: " + extr_tuples[i].key + ": "
+ extr_tuples[i].value + " was unsuccessful, probably a duplicate-key.\n";
            ret[1]++;
        } else if (result == MEMCACHED_READ_FAILURE){ //read failed
            //std::cout << "Operation " + extr_ops[i] + " on key : " + extr_tuples[i].key + " was
unsuccessful, key does probably not exist in the database.\n";
            ret[3]++;
        } else if (result == MEMCACHED_DATA_DOES_NOT_EXIST){ //update failed
```

```
            //std::cout << "Operation " + extr_ops[i] + " on 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 (result == MEMCACHED_NOTFOUND){ //delete failed
            //std::cout << "Operation " + extr_ops[i] + " on key : " + extr_tuples[i].key + " was
unsuccessful, key does probably not exist in the database.\n";
            ret[7]++;
        } else {
            //std::cerr << "Something went wrong, error code \"" <<
memcached_strerror(memcached, result) << "\" from Memcachedlib; Wanted to " <<
tuple.operation << " " + tuple.key + "\n";
            if (tuples[i].operation == tuple_t::INSERT){
                ret[1]++;
                res[result]++;
            }
            else if (tuples[i].operation == tuple_t::READ){
                ret[3]++;
                res[result]++;
            }
            else if (tuples[i].operation == tuple_t::UPDATE){
                ret[5]++;
                res[result]++;
            }
            else if (tuples[i].operation == tuple_t::DELETE){
                ret[7]++;
                res[result]++;
            }
        }
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////

        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);
        ops_counter++;
        if(ops_counter % 500000 == 0){
            printf("Thread handled %.2f%% of its extra-operations, %d of %i.\n",
((float(ops_counter)*100.0)/float(thread_arg->end_i - thread_arg->start_i)), ops_counter,
thread_arg->end_i - thread_arg->start_i);
        }
    }

    memcached_flush_buffers(memcached);
    memcached_free(memcached);
```

```
    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;

    int res[60];
    for (int &re : res)
        re = 0;

    int ops_counter = 0;
    auto *thread_arg = (struct thread_arg_t*)thread_args;
    memcached_st* memcached = setup_memcached();

    for (int i = thread_arg->start_i; i < thread_arg->end_i; i++){
        memcached_return_t result = send_query(memcached, tuple_t::DELETE, del_keys[i], "");
        ///Handling server feedback////////////////////////////////////////////////////////////////////////////////
        if (result == MEMCACHED_DELETED) {//delete success
            ret [0]++;
        } else if (result == MEMCACHED_NOTFOUND){ //delete failed
            //std::cout << "Operation " << tuple.operation << " on key : " + tuple.key + " was
unsuccessful, key was probably already deleted by a previous delete-operation.\n"; //TODO
            ret[1]++;
        } else {
            //std::cerr << "Something went wrong, error code \"" <<
memcached_strerror(memcached, result) <<  "\" from Memcachedlib; Wanted to " <<
tuple.operation << " " + tuple.key + "\n";
            res[result]++;
            ret[1]++;
        }
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////

        ops_counter++;
        if(ops_counter % 500000 == 0){
```

```
            printf("Thread handled %.2f%% of its delete-operations, %d of %i.\n",
((float(ops_counter)*100.0)/float(thread_arg->end_i - thread_arg->start_i)), ops_counter,
thread_arg->end_i - thread_arg->start_i);
        }
    }

    for (int i = 0; i < 60; i++){
        if (res[i] != 0)
            printf("%d was %d\n", i, res[i]);
    }


    memcached_flush_buffers(memcached);
    memcached_free(memcached);
    delete(thread_arg);

    return (void*) ret;
}
```