```cpp
// SimQP.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "dbmsproj.h"
#include "semihashjoin.h"
#include <string>
#include <sstream>
#include <stdio.h>
#define GET_VARIABLE_NAME(Variable) (#Variable)
/***Using vectors represent disk to hold records block*****/

//NYC - This site has a copy of branch, account and depositor tables.
std::vector<block_t<account_t>> account_NYC;
std::vector<block_t<branch_t>> branch_NYC;
std::vector<block_t<depositor_t>> depositor_NYC;

//SFO - This site has complete branch table and a fragment of account table.
//Account table will have only records that are local to SFO.
std::vector<block_t<branch_t>> branch_SFO;
std::vector<block_t<account_t>> frag_account_SFO;

//OMA - This site has fragments of Account and Depositor tables.
//These tables store records that are local to OMA.
std::vector<block_t<account_t>> frag_account_OMA;
std::vector<block_t<depositor_t>> frag_depositor_OMA;

//HOU (Houston) – This site has a complete copy of customer table.
std::vector<block_t<customer_t>> customer_HOU;


//unsigned int MAX_RECORDS_PER_BLOCK=7;


int main(int argc, char* argv[])
{

    //initialize
    char* filename;

    //fill account table at NYC
    filename = "Account_NYC.txt";
    fillTable<account_t>(filename, account_NYC, 6);

    //fill branch table at NYC
    filename = "Branch_NYC.txt";
    fillTable<branch_t>(filename, branch_NYC, 1);

    //fill depositor table at NYC
    filename = "Depositor_NYC.txt";
    fillTable<depositor_t>(filename, depositor_NYC, 4);

    //fill customer table at HOU
    filename= "Customer_HOU.txt";
    fillTable<customer_t>(filename, customer_HOU, 8);

    //fill fragment of account table at SFO
    filename = "Frag_Account_SFO.txt";
    fillTable<account_t>(filename, frag_account_SFO, 2);
```

```
60
61      //fill fragment of depositor table at OMA
62      filename = "Frag_Depositor_OMA.txt";
63      fillTable<depositor_t>(filename, frag_depositor_OMA, 2);
64
65      //fill fragment of account table at OMA
66      filename = "Frag_Account_OMA.txt";
67      //fillTable<account_t>(filename, frag_account_OMA, 3);
68      //printf("All tables are filled.\n");
69      //printf("============================================================\n");
70      std::istringstream ss(argv[1]);
71      int query;
72      if (!(ss >> query)) std::cerr << "Invalid number" << argv[1] << std::endl;
73
74      std::vector<std::string> command;
75      std::vector<std::string> parameters;
76      std::string require_site;
77      std::string current_site;
78
79
80      switch (query)
81      {
82      case 1:
83              {
84                  std::vector<block_t<customer_t>> out;
85                  printf("Query 1: \n");
86                  printf("Require site: NYC\n");
87                  printf("Statement: Select* from customer |X depositor:\n");
88                  printf("Processing:\n");
89                  printf("The customer table is shipped from HOU to NYC site. \n");
90
91                  printf("\n");
92                  SemiHashJoin<customer_t,depositor_t>
                      (customer_HOU,depositor_NYC,"customer_name", out);
93                  printf("The customer table are SemiHashJoined with depositor table at NYC
                       site. \n");
94                  printf("Project all fields in customer table.\n");
95                  printf("\n");
96                  printf("Output: \n");
97                  printf("=========================================\n");
98                  printf("Name    Street  City \n");
99                  printf("=========================================\n");
100                 unsigned int count = 0;
101                 for each ( auto block in out)
102                 {
103
104                     for each (auto record in block.entries)
105                         {
106                             record.project(3, "customer_name",
                        "customer_city","customer_street");
107                             count++;
108                         }
109
110                 }
111                 printf("=========================================\n");
112                 printf("The query output %d records.\n",count);
113                 break;
114         }
```

```
115     case 2:
116           {
117
118               printf("Query 2: \n");
119               printf("Require site: SFO\n");
120               printf("Statement: Select name, balance from depositor |X| account where
                    branch_name='Chinatown':\n");
121               printf("Processing:\n");
122
123               printf("3) Ship the results from NYC to SFO and then do projection. \n");
124               printf("\n");
125                std::vector<block_t<account_t>> table_selec;
126               printf("Select the records with branch_name='Chinatown' using the fragment
                    of account table at SFO.\n");
127                select(account_NYC, table_selec, "branch_name", "Chinatown");
128
129                if (table_selec.size() == 0)
130                {
131                    printf("no record is selected according to the condition.\n");
132                    break;
133                }
134                else
135                {
136                    unsigned int rec_selected = (table_selec.size() - 1)*table_selec
                      [0].maxRecords + table_selec[table_selec.size()-1].nreserved;
137                    //printf("%d records are selected.\n", rec_selected);
138                }
139                printf("Ship the selection results from SFO to HOU  and hash joined with
                    depositor table. \n");
140                std::vector<join_t<depositor_t,account_t>> joinout;
141                HashJoin<depositor_t,account_t>(depositor_NYC,
                    table_selec,"account_number", joinout);
142                printf("The join results at HOU is shipped back to SFO. \n");
143                printf("\n");
144                printf("Output: \n");
145                printf("=========================================\n");
146                printf("Name Balance\n");
147                printf("=========================================\n");
148                unsigned int count=0;
149                for each (auto var in joinout)
150               {
151                        var.display(2,"customer_name","balance");
152                        count++;
153               }
154                printf("=========================================\n");
155                printf("The query output %d records.\n", count);
156                break;
157           }
158     case 3:
159     {
160
161         printf("Query 3: \n");
162         printf("Require site: SFO\n");
163         printf("Statement: Select street, city from customer|X (depositor |X account
              where account_number='A10352'):\n");
164         printf("Processing:\n");
165
166         printf("\n");
```

```
167         std::vector<block_t<account_t>> table_selec;
168
169         printf("Search the fragment of account table at SFO to check if there have
              records with account_number='A10352' \n");
170         select(frag_account_SFO, table_selec, "account_number", "A10352");
171         if (table_selec.size() == 0)
172         {
173             printf("no record is found according to the condition.\n");
174             select(account_NYC, table_selec, "account_number", "A10352");
175             printf("Search the account table at NYC.\n");
176         }
177
178         if (table_selec.size() == 0)
179         {
180             printf("no record is selected according to the condition.\n");
181             break;
182         }
183         else
184         {
185
186             unsigned int rec_selected = (table_selec.size() - 1)*table_selec
                [0].maxRecords + table_selec[table_selec.size() - 1].nreserved;
187             //printf("%d records are selected.\n", rec_selected);
188         }
189
190         std::vector<block_t<depositor_t>> semi_out;
191         printf("The depositor table is semi-joined with intermediate result from select
              operation at NYC.\n");
192         SemiHashJoin<depositor_t,account_t>(depositor_NYC,table_selec,"account_number",
              semi_out);
193
194         printf("The intermedate results from semijoin are shipped from NYC to HOU. \n");
195         printf("The customer table at HOU is hash joined with the intermedate results.
              \n");
196         std::vector<join_t<customer_t, depositor_t>> joinout;
197         HashJoin<customer_t, depositor_t>(customer_HOU, semi_out, "customer_name",
              joinout);
198
199         printf("The final result is shipped from HOU to the SFO. \n");
200
201         printf("Output: \n");
202         printf("=========================================\n");
203         printf("Street      City\n");
204         printf("=========================================\n");
205
206         unsigned int count=0;
207         for each (auto var in joinout)
208         {
209             var.display(2, "customer_street", "customer_city");
210             count++;
211         }
212         printf("=========================================\n");
213         printf("The query output %d records.\n", count);
214         break;
215     }
216     case 4:
217     {
218
219         printf("Query 4: \n");
```

```
220         printf("Require site: NYC\n");
221         printf("Statement: Select account_number, balance, branch_name branch_city from
              branch|X| acocunt where account_number='A10352'):\n");
222         printf("Processing:\n");
223
224         std::vector<block_t<account_t>> table_selec;
225         printf("Select the record from the account table at NYC where
              account_number='A10352' \n");
226         select(account_NYC, table_selec, "account_number", "A10352");
227         if (table_selec.size() == 0)
228         {
229             //printf("no record is selected according to the condition.\n");
230             break;
231         }
232         else
233         {
234             unsigned int rec_selected = (table_selec.size() - 1)*table_selec
                [0].maxRecords + table_selec[table_selec.size() - 1].nreserved;
235             //printf("%d records are selected.\n", rec_selected);
236         }
237
238         std::vector<join_t<branch_t, account_t>> joinout;
239         printf("The branch table is hash joined with intermediate result from select
              operation at NYC.\n");
240         HashJoin<branch_t, account_t>(branch_NYC, table_selec, "branch_name", joinout);
241         printf("=========================================\n");
242
243         printf("Output: \n");
244         printf("=========================================\n");
245         printf("account_nmber   balance branch_name branch_city\n");
246         printf("=========================================\n");
247         unsigned int count = 0;
248         for each (auto var in joinout)
249         {
250             var.display(4, "account_number", "balance", "branch_name", "branch_city");
251             count++;
252         }
253         printf("=========================================\n");
254         printf("The query output %d records.\n", count);
255         break;
256     }
257     default:
258         break;
259     }
260
261     printf("\n");
262
263 }
264
265
```

```
#pragma once
/*
* DBMS Implementation
*
* Contact: geopiskas@gmail.com
*/


#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <vector>

#include "dbmsproj.h"
#include "recordOps.h"
#include "bufferOps.h"
#define GET_VARIABLE_NAME(Variable) (#Variable)

/*
* seed: seed to use in hash function
* buffer: buffer used, already loaded with a relation to hash
* size: the size in blocks of the relation loaded on buffer
* field: which field will be used for joining
*
* returns the pointer to the hash index
*/
template <typename T>linkedRecordPtr** createHashIndex(char *seed, block_t<T> *buffer,
  unsigned int size, std::string const& field, int debugmode = 0) {
    //initialize

    // the hash index consists of a maximum of hashSize linked lists where
    // each list has pointers to the records with common hash value

    // the size of hashIndex
    unsigned int hashSize = size * buffer->maxRecords;
    linkedRecordPtr **hashIndex = (linkedRecordPtr**)malloc(hashSize * sizeof
      (linkedRecordPtr*));
    for (unsigned int i = 0; i < hashSize; i++) {
        hashIndex[i] = nullptr;
    }

    recordPtr start = newPtr(0,buffer->maxRecords);
    unsigned int offset = (size - 1)*buffer->maxRecords + (buffer+size-1)->nreserved;
    recordPtr end = newPtr(offset,buffer->maxRecords);

    // starting from the very first record, all valid records in valid blocks
    // are hashed
    for (; start < end; incr(start,buffer->maxRecords)) {
        if (!buffer[start.block].valid) {
            start.record = buffer->maxRecords - 1;
            continue;
        }
        T record = getRecord<T>(buffer, start);
        if (record.valid) {
            unsigned int index = hashRecord<T>(seed, record,hashSize,field);
            linkedRecordPtr *ptr = (linkedRecordPtr*)malloc(sizeof(linkedRecordPtr));
            ptr->ptr = start;
```

```
57              ptr->next = hashIndex[index];
58              hashIndex[index] = ptr;
59          }
60      }
61
62      // returns hashIndex
63      printf("Hash index is created successfully!\n");
64      return hashIndex;
65  };
66
67  /*
68  * infile: filename of the file whose records will be joined with the ones on buffer
69  * inBlocks: size of infile
70  * buffer: the buffer that is used (a file is already loaded on it)
71  * nmem_blocks: size of buffer
72  * size: the size of the file already loaded on buffer
73  * out: file descriptor of the outfile
74  * nres: number of pairs
75  * nios: number of ios
76  * field: which field will be used for joining
77  */
78
79  template <typename T1, typename T2> void hashAndProbe(std::vector<block_t<T1>> &r,
      block_t<T1> *buffer, block_t<T2>* buildbuffer,unsigned int size, std::string const&
      field, std::vector<block_t<T1>> &out, int debugmode = 0) {
80
81      unsigned int mod;
82
83      mod = size*buildbuffer->maxRecords;
84
85      char *seed = "agagagepiggeeq331516166fwhfsfrs";
86
87      // hash index for the records already on buffer is created
88      linkedRecordPtr **hashIndex = createHashIndex<T2>(seed, buildbuffer,size, field);
89      // pointer to the buffer block where blocks of infile are loaded
90      block_t<T1> *bufferIn = buffer;
91      // pointer to the last buffer block, where pairs for output are written
92      block_t<T1> *bufferOut = buffer + 1;
93
94
95      for (unsigned int i = 0; i < r.size(); i++) {
96          // if the block loaded is invalid, loads the next one
97          readBlocks<T1>(r, bufferIn, 1,i);
98          if (!(*bufferIn).valid) {
99              continue;
100         }
101         // each record of the loaded block is hashed
102         // then the linked list of the hash index for the corresponding hash value
103         // is examined, and if a record has same value as the current one, both
104         // are written to the output block
105
106         for (unsigned int j = 0; j < bufferIn->nreserved; j++) {
107             T1 record = (*bufferIn).entries[j];
108             if (record.valid) {
109                 unsigned int index = hashRecord<T1>(seed, record, mod,field);
110                 linkedRecordPtr *element = hashIndex[index];
111                 while (element) {
112                     T2 tmp = getRecord(buildbuffer, element->ptr);
```

```
113                     if (compareRecords<T1,T2>(record, tmp, field) == 0) {
114
115                         bufferOut->entries.push_back(record);
116                         bufferOut->nreserved++;
117                         //(*bufferOut).entries[(*bufferOut).nreserved++] = tmp;
118                         //(*nres) += 1;
119
120                         // if output block becomes full, writes it to the outfile
121                         // and empties it
122
123                         if (bufferOut->nreserved == bufferOut->maxRecords) {
124                             // writeBlocks(out, bufferOut, 1);
125                             out.push_back(*bufferOut);
126                             emptyBlock<T1>(bufferOut, bufferOut->maxRecords);
127                             bufferOut->blockid += 1;
128                         }
129                     }
130                     element = element->next;
131                 }
132             }
133         }
134     }
135
136     if (bufferOut->nreserved != 0) {
137         out.push_back(*bufferOut);
138     }
139     destroyHashIndex(hashIndex, size);
140 };
141 template <typename T1, typename T2> void hashAndProbefull(std::vector<block_t<T1>> &r,
      block_t<T1> *buffer, block_t<T2>* buildbuffer, unsigned int size, std::string const&
      field, std::vector<join_t<T1,T2>> &joinout, int debugmode = 0) {
142
143     printf("Building in-memory hash index and probing...\n");
144     unsigned int mod;
145     mod = size*buildbuffer->maxRecords;
146     char *seed = "agagagepiggeeq331516166fwhfsfrs";
147
148     // hash index for the records already on buffer is created
149     linkedRecordPtr **hashIndex = createHashIndex<T2>(seed, buildbuffer, size, field);
150     // pointer to the buffer block where blocks of infile are loaded
151     block_t<T1> *bufferIn = buffer;
152     // pointer to the last buffer block, where pairs for output are written
153     block_t<T1> *bufferOut = buffer + 1;
154
155
156     for (unsigned int i = 0; i < r.size(); i++) {
157         // if the block loaded is invalid, loads the next one
158         readBlocks<T1>(r, bufferIn, 1, i);
159         printf("Loading 1 block of proble relation into input buffer block.\n");
160         if (!(*bufferIn).valid) {
161             continue;
162         }
163         // each record of the loaded block is hashed
164         // then the linked list of the hash index for the corresponding hash value
165         // is examined, and if a record has same value as the current one, both
166         // are written to the output block
167
168         for (unsigned int j = 0; j < bufferIn->nreserved; j++) {
169             T1 record = (*bufferIn).entries[j];
```

```
170                 if (record.valid) {
171
172                     unsigned int index = hashRecord<T1>(seed, record, mod, field);
173                     printf("Get 1 record from input buffer, hash it \n");
174                     printf("Look up the hash index to match the records with same index=%d
                          \n",index);
175                     linkedRecordPtr *element = hashIndex[index];
176                     while (element) {
177
178                         T2 tmp = getRecord(buildbuffer, element->ptr);
179                         if (compareRecords<T1, T2>(record, tmp, field) == 0) {
180
181                             bufferOut->entries.push_back(record);
182                             bufferOut->nreserved++;
183                             //(*bufferOut).entries[(*bufferOut).nreserved++] = tmp;
184                             //(*nres) += 1;
185
186                             // if output block becomes full, writes it to the outfile
187                             // and empties it
188                             printf("Join the two records. Get next record... \n");
189                             join_t<T1, T2> rec;
190                             rec.rec1 = record;
191                             rec.rec2 = tmp;
192                             joinout.push_back(rec);
193                             if (bufferOut->nreserved == bufferOut->maxRecords) {
194                                 // writeBlocks(out, bufferOut, 1);
195                                 printf("block is full, write back to disk. \n");
196                                 emptyBlock<T1>(bufferOut, bufferOut->maxRecords);
197                                 bufferOut->blockid += 1;
198                             }
199                         }
200                         element = element->next;
201                     }
202                 }
203             }
204         }
205
206         destroyHashIndex(hashIndex, size);
207         printf("End of probing. \n");
208         printf("==================================== \n");
209         printf("\n");
210     };
211     /*
212     * filename: the name of the file to be partitioned
213     * size: the size of the file
214     * seed: a seed for the hash function
215     * buffer: the buffer that is used
216     * nmem_blocks: size of buffer
217     * bucketFilenames: array with the filenames of the bucket files to be produced
218     * mod: to be used for hashing
219     * nios: number of ios
220     * field: which field will be used for joining
221     */
222     template<typename T> void createBuckets(std::vector<block_t<T>> &r, block_t<T> *buffer,
          std::vector<std::vector<block_t<T>>> &partition, unsigned int mod, std::string const&
          field, int debugmod = 0) {
223
224
```

```
225     unsigned int block_counts = r.size();
226     block_t<T> *bufferIn = buffer + MAX_MEMORY_BLOCKS - 1;
227
228     for (unsigned int i = 0; i <block_counts; i++)
229     {
230
231         *bufferIn = r[i];//load one block from r into the last buffer block.
232         printf("Loading 1 block from disk into input buffer block. \n");
233         unsigned int max_records = bufferIn->entries.size();
234         for (unsigned int j = 0; j < max_records; j++)
235         {
236             T record = bufferIn->entries[j];
237             if (record.valid)
238             {
239                 unsigned int index = hashRecord<T>
                      ("1235peqwtpqtuqewptuqptup1qtptu3421-58-12-35", record, mod,field);
240                 buffer[index].nreserved++;
241                 buffer[index].entries.push_back(record);
242                 //printf("record is hashed and put into %dth buffer block. \n",index);
243                 if (buffer[index].nreserved == buffer->maxRecords)
244                 {
245                     printf("%dth buffer block is full, write back to %dth partition.
                          \n",index,index);
246                     buffer[index].valid = true;
247                     partition[index].push_back(buffer[index]);
248                     printf("Empty %dth buffer block. \n",index);
249                     emptyBlock<T>(buffer + index, buffer->maxRecords);
250                 }
251             }
252         }
253         printf("All the records in the input buffer have been processed, loading next
              block... \n");
254     }
255     printf("All the blocks in the disk have been processed.\n");
256 //put all non full blocks into the corresponding partition
257     for (unsigned int i = 0; i < mod; i++)
258     {
259         if (buffer[i].nreserved != 0)
260         {
261             buffer[i].valid = true;
262             partition[i].push_back(buffer[i]);
263             emptyBlock<T>(buffer + i, buffer[i].entries.size());
264
265         }
266
267     }
268
269
270     if (debugmod !=0)
271     {
272     printf("write all non-full buffer blocks back to corrsponding partition. \n");
273     printf("The table has been partitioned into %d partitions: \n", mod);
274     printf("========================================================================\n");
275     //display the results
276     unsigned int numPartition = 0;
277     unsigned int numBlocks=0;
278     unsigned int numRecords = 0;
279
280     for each (auto blocks in partition)
```

```
281     {
282
283         numBlocks = 0;
284         for each (auto block in blocks)
285         {
286             printf("The %dth Partition, %dth block:\n",numPartition,numBlocks);
287             block.printrecord();
288             numBlocks += 1;
289         }
290         numPartition += 1;
291         printf("  \n");
292     }
293
294     }
295     printf("End of partitioning the table. \n");
296     printf("================================================== \n");
297     printf("\n");
298 };
299
300
301 template <typename T1,typename T2> void SemiHashJoin(std::vector<block_t<T1>> &r,
      std::vector<block_t<T2>> &s, char* field, std::vector<block_t<T1>> &out,int
      debugmode=0) {
302
303     //figure out how many partitions to create
304     unsigned int smallSize;
305     if (r.size() > s.size()) smallSize = s.size(); else smallSize = r.size();
306     unsigned int mod=4;
307     /*mod = smallSize / (MAX_MEMORY_BLOCKS - 1);
308     if (smallSize % (MAX_MEMORY_BLOCKS - 1)) mod += 1;
309     if (mod > MAX_MEMORY_BLOCKS) mod = MAX_MEMORY_BLOCKS;
310     */
311     //create the partitions for r and s;
312     std::vector<std::vector<block_t<T1>>> r_partition(mod);
313     block_t<T1> T1buffer[MAX_MEMORY_BLOCKS];
314     std::vector<std::vector<block_t<T2>>> s_partition(mod);
315     block_t<T2> T2buffer[MAX_MEMORY_BLOCKS];
316     printf("Partitioning...\n");
317
318     createBuckets<T1>(r, T1buffer,r_partition,mod,field);
319     printf("The %s table has been partitioned into %d partitions: \n", GET_VARIABLE_NAME
        (s), mod);
320     printf("============================================================================\n");
321     createBuckets<T2>(s, T2buffer, s_partition, mod, field);
322
323     //process each partition each time
324     for (unsigned int i = 0; i < mod; i++)
325     {
326         unsigned int inblocks=r_partition[i].size();
327         block_t<T2> buildbuffer[MAX_MEMORY_BLOCKS-2];
328         block_t<T1> buffer[2];
329         if (s_partition[i].size()!=0)
330         {
331         //load one of s partitions into build buffer
332             for (unsigned int j = 0; j < s_partition[i].size(); j++)
333             {
334                 buildbuffer[j] = s_partition[i][j];
335             }
336             hashAndProbe(r_partition[i], buffer, buildbuffer, s_partition[i].size()
```

```
            field, out);
337
338         }
339     }
340 }
341
342 template <typename T1, typename T2> void HashJoin(std::vector<block_t<T1>> &r,
      std::vector<block_t<T2>> &s, char* field, std::vector<join_t<T1,T2>> &out, int
      debugmode = 0) {
343
344     //figure out how many partitions to create
345     unsigned int smallSize;
346     if (r.size() > s.size()) smallSize = s.size(); else smallSize = r.size();
347     unsigned int mod = 4;
348     /*mod = smallSize / (MAX_MEMORY_BLOCKS - 1);
349     if (smallSize % (MAX_MEMORY_BLOCKS - 1)) mod += 1;
350     if (mod > MAX_MEMORY_BLOCKS) mod = MAX_MEMORY_BLOCKS;
351     */
352     //create the partitions for r and s;
353     std::vector<std::vector<block_t<T1>>> r_partition(mod);
354     block_t<T1> T1buffer[MAX_MEMORY_BLOCKS];
355     std::vector<std::vector<block_t<T2>>> s_partition(mod);
356     block_t<T2> T2buffer[MAX_MEMORY_BLOCKS];
357     char* name = GET_VARIABLE_NAME(r);
358
359
360     printf("================================================================================\n");

361     createBuckets<T1>(r, T1buffer, r_partition, mod, field);
362
363     printf("The %s table has been partitioned into %d partitions: \n", GET_VARIABLE_NAME
        (s), mod);
364     printf("================================================================================\n");
365     createBuckets<T2>(s, T2buffer, s_partition, mod, field);
366
367     for (unsigned int i = 0; i < mod; i++)
368     {
369         unsigned int inblocks = r_partition[i].size();
370         block_t<T2> buildbuffer[MAX_MEMORY_BLOCKS - 2];
371         block_t<T1> buffer[2];
372
373         if (s_partition[i].size()!=0)
374         {
375         //load one of s partitions into build buffer
376
377         for (unsigned int j = 0; j < s_partition[i].size(); j++)
378         {
379             buildbuffer[j] = s_partition[i][j];
380             printf("Loading all blocks in %dth build partition into buffer. \n",j);
381         }
382         hashAndProbefull(r_partition[i], buffer, buildbuffer, s_partition[i].size(),
            field, out);
383     }
384   }
385
386 }
```

```
1  /*
2  * DBMS Implementation
3  */
4
5  #ifndef _DBMSPROJ_H
6  #define _DBMSPROJ_H
7
8  #include "stdafx.h"
9  #include <fstream>
10 #include <string>
11 #include <stdio.h>
12 #include <iostream>
13 #include <sstream>
14 #include <vector>
15 #include <cstdarg>
16 //#include "bufferOps.h"
17
18 #define TUPLES_PER_ACCOUNT_BLOCK 10
19 #define TUPLES_PER_BRANCH_BLOCK 7
20 #define TUPLES_PER_CUSTOMER_BLOCK 8
21 #define TUPLES_PER_DEPOSITOR_BLOCK 15
22
23 #define NUM_BLOCKS_ACCOUNT 6
24 #define NUM_BLOCKS_BRANCH 1
25 #define NUM_BLOCKS_CUSTOMER 8
26 #define NUM_BLOCKS_DEPOSITOR 4
27
28 #define MAX_RECORDS_PER_BUCKET 20
29
30
31 #define MAX_MEMORY_BLOCKS 5
32
33 //definition of record
34 //structure of records in account table;
35 enum field_code                                                              ↵
     {e_account_number,e_branch_name,e_balance,e_branch_city,e_assets,e_customer_name,e_cust ↵
     omer_street,e_customer_city};
36 inline field_code getfield(std::string const& field)
37 {
38     if (field == "account_number") return e_account_number;
39     if (field == "branch_name") return e_branch_name;
40     if (field == "balance") return e_balance;
41     if (field == "branch_city") return e_branch_city;
42     if (field == "assets") return e_assets;
43     if (field == "customer_name") return e_customer_name;
44     if (field == "customer_street") return e_customer_street;
45     if (field == "customer_city") return e_customer_city;
46 };
47 typedef struct {
48     std::string account_number;
49     std::string branch_name;
50     int balance = 0;
51     bool valid=false; //if set, then the record is valid
52     void setCol(std::string &val1, std::string &val2, std::string &val3)
53     {
54         account_number = val1;
55         branch_name = val2;
56         balance = std::stoi(val3,0);
```

```
57          valid = true;
58      }
59      std::string getCol(std::string const& field)
60      {
61          field_code fields = getfield(field);
62
63          switch (fields)
64          {
65          case e_account_number: return account_number; break;
66          case e_branch_name: return branch_name; break;
67          case e_balance: return std::to_string(balance); break;
68          default: return (" "); break;
69          }
70      }
71
72      void display() {
73          std::cout << account_number << "    " << branch_name << "   " << balance <<
              std::endl;
74      }
75      void project(int num, const char* field, ...)
76      {
77          std::string fields(field);
78          if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
79          va_list arguments;
80          va_start(arguments, field);
81          for (size_t i = 0; i < num - 1; i++)
82          {
83              std::string fields(va_arg(arguments, char*));
84              if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
85          }
86          std::cout << std::endl;
87          va_end(arguments);
88      }
89
90  }account_t;
91
92  //structure of records in branch table;
93  typedef struct {
94      std::string branch_name;
95      std::string branch_city;
96      int assets = 0;
97       bool valid=false; //if set, then the record is valid
98      std::string getCol(std::string const& field)
99      {
100         field_code fields = getfield(field);
101
102         switch (fields)
103         {
104         case e_branch_name: return branch_name; break;
105         case e_branch_city: return branch_city; break;
106         //case 3: return assets; break;
107         default:return (" ");
108             break;
109         }
110     }
111     void setCol(std::string &val1, std::string &val2, std::string &val3)
112     {
113         branch_name = val1;
114         branch_city = val2;
```

```
115         assets = std::stoi(val3, 0);
116         valid = true;
117     }
118     void display() {
119         std::cout << branch_name << "   " << branch_city << "   " << assets << std::endl;
120     }
121     void project(int num, const char* field, ...)
122     {
123         std::string fields(field);
124         if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
125         va_list arguments;
126         va_start(arguments, field);
127         for (size_t i = 0; i < num - 1; i++)
128         {
129             std::string fields(va_arg(arguments, char*));
130             if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
131         }
132         std::cout << std::endl;
133         va_end(arguments);
134     }
135 }branch_t;
136
137 //structure of records in customer table;
138 typedef struct {
139     std::string customer_name;
140     std::string customer_street;
141     std::string customer_city;
142     bool valid=false; //if set, then the record is valid
143     void setCol(std::string &val1, std::string &val2, std::string &val3)
144     {
145         customer_name = val1;
146         customer_street = val2;
147         customer_city = val3;
148         valid = true;
149     }
150     std::string getCol(std::string const& field)
151     {
152         field_code fields = getfield(field);
153         switch (fields)
154         {
155         case e_customer_name: return customer_name; break;
156         case e_customer_street: return customer_street; break;
157         case e_customer_city: return customer_city; break;
158         default:return (" ");
159             break;
160         }
161     }
162     void display() {
163         std::cout << customer_name << " " << customer_street << "   " << customer_city <<
              std::endl;
164     }
165     void project(int num, const char* field, ...)
166     {
167         std::string fields(field);
168         if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
169         va_list arguments;
170         va_start(arguments, field);
171         for (size_t i = 0; i < num - 1; i++)
```

```
172         {
173             std::string fields(va_arg(arguments, char*));
174             if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
175         }
176         std::cout << std::endl;
177         va_end(arguments);
178     }
179 }customer_t;
180
181 //structure of records in depositor table;
182 typedef struct {
183     std::string customer_name;
184     std::string account_number;
185     bool valid=false; //if set, then the record is valid
186     void setCol(std::string &val1, std::string &val2, std::string &val3)
187     {
188         customer_name = val1;
189         account_number = val2;
190         valid = true;
191     }
192
193     std::string getCol(std::string const& field)
194     {
195         field_code fields = getfield(field);
196         switch (fields)
197         {
198         case e_customer_name: return customer_name; break;
199         case e_account_number: return account_number; break;
200         default: return (" "); break;
201         }
202     }
203     void display() {
204         std::cout << account_number << "    " << customer_name<< std::endl;
205     }
206     void project(int num, const char* field, ...)
207     {
208         std::string fields(field);
209         if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
210         va_list arguments;
211         va_start(arguments, field);
212         for (size_t i = 0; i < num - 1; i++)
213         {
214             std::string fields(va_arg(arguments, char*));
215             if (getCol(fields) != (" ")) std::cout << getCol(fields) << "   ";
216         }
217         std::cout << std::endl;
218         va_end(arguments);
219     }
220 }depositor_t;
221
222 template<typename T> struct block_t {
223     unsigned int blockid;
224     unsigned int nreserved;
225     unsigned int maxRecords;
226     bool valid;
227     std::vector<T> entries;
228
229     block_t()
230     {
```

```
231         blockid = 0;
232         nreserved = 0;
233         valid = false;
234         if (std::is_same<T, account_t>::value) maxRecords = 10;
235         if (std::is_same<T, branch_t>::value)  maxRecords = 7;
236         if (std::is_same<T, customer_t>::value)  maxRecords = 8;
237         if (std::is_same<T, depositor_t>::value)  maxRecords = 15;
238         //entries.reserve(maxRecords);
239     }
240
241     void printrecord()
242     {
243         for (unsigned int i = 0; i < entries.size(); i++)
244         {
245             entries[i].display();
246         }
247     }
248
249 };
250
251
252
253
254 template<typename T1, typename T2> struct join_t {
255     T1 rec1;
256     T2 rec2;
257     void display(int num, std::string field, ...)
258     {
259         std::string fields(field);
260         if (rec1.getCol(fields) != (" ")) std::cout << rec1.getCol(fields) << " ";
261         if (rec2.getCol(fields) != (" ")) std::cout << rec2.getCol(fields) << " ";
262         va_list arguments;
263         va_start (arguments, field);
264         for (size_t i = 0; i < num-1; i++)
265         {
266             std::string fields(va_arg(arguments, char*));
267             if (rec1.getCol(fields) != (" ")) std::cout << rec1.getCol(fields) <<"  ";
268             if (rec2.getCol(fields) != (" ") ) std::cout << rec2.getCol(fields)<<"  ";
269         }
270         std::cout << std::endl;
271         va_end(arguments);
272     }
273 };
274
275
276
277 /*
      ------------------------------------------------------------------------------------------
      --------------------------------
278   Functions
279
    ------------------------------------------------------------------------------------------
    ----------------------------
280  */
281
282 template<typename T> void fillTable(char* filename, std::vector<block_t<T>> &table,
      unsigned int numblocks, int degbugmode = 0) {
283
284     std::ifstream file(filename);
285
```

```
286     int rec_counts = 0;
287     int block_counts = 0;
288
289     for (unsigned int i = 0; i < numblocks; i++)
290     {
291         std::string   line;
292         unsigned int index = 0;
293         block_t<T> block;
294
295         while ((index<block.maxRecords) && (std::getline(file, line)))
296         {
297             std::string   val1;
298             std::string   val2;
299             std::string   val3;
300             T rec;
301             std::stringstream   linestream(line);
302             std::string         data;
303             std::getline(linestream, val1, '\t');
304             std::getline(linestream, val2, '\t');
305             std::getline(linestream, val3, '\t');
306
307             rec.setCol(val1, val2, val3);
308             rec.valid = true;
309             block.entries.push_back(rec);
310             rec_counts++;
311             index++;
312         }
313         block.blockid = i;
314         block.nreserved = index;
315         block.valid = true;
316         table.push_back(block);
317         block_counts++;
318     }
319
320     if (degbugmode !=0)
321     {
322
323     printf("%s Table in NYC site has %d records in %d blocks.\n", filename,rec_counts,
          block_counts);
324     for each (block_t<T> block in table)
325     {
326         block.printrecord();
327         printf("\n");
328     }
329     printf("\n");
330     }
331 }
332 template <typename T> void select(std::vector<block_t<T>> &in, std::vector<block_t<T>>
      &out, std::string const & field, std::string const & val,int debugmod=0)
333 {
334     printf("Start processing selection.... \n");
335     block_t<T> buffer[MAX_MEMORY_BLOCKS];
336     //figure out hom many times need to load the blocks for the input relation
337     unsigned int size = ((in.size() + MAX_MEMORY_BLOCKS - 2)) / (MAX_MEMORY_BLOCKS - 1);
338
339     //set the out block pointer to last block of the buffer
340     block_t<T> *bufferOut = buffer + MAX_MEMORY_BLOCKS - 1;
341
342     //process the blocks loaded into the buffer each time
```

```
343     for (unsigned int i = 0; i < size; i++)
344     {
345         int recordcounts = 0;
346         int nblocks=readBlocks<T>(in, buffer, (MAX_MEMORY_BLOCKS - 1), i*
              (MAX_MEMORY_BLOCKS - 1));
347
348         recordPtr start = newPtr(0, buffer->maxRecords);
349         //calculate the total records of the loaded blocks in the buffer
350         unsigned int offset = (nblocks - 1)*buffer->maxRecords + (buffer + nblocks - 1)-
              >nreserved;
351         recordPtr end = newPtr(offset, buffer->maxRecords);
352
353         // starting from the very first record, all valid records in valid blocks are
              hashed
354         for (; start < end; incr(start, buffer->maxRecords))
355         {
356             if (!buffer[start.block].valid) {
357                 start.record = buffer->maxRecords - 1;
358                 continue;
359             }
360             T record = getRecord<T>(buffer, start);
361             if (record.getCol(field) == val)
362             {
363                 recordcounts++;
364                 bufferOut->entries.push_back(record);
365                 bufferOut->nreserved++;
366                 if (bufferOut->nreserved == bufferOut->maxRecords)
367                 {
368                     out.push_back(*bufferOut);
369                     emptyBlock<T>(bufferOut, bufferOut->maxRecords);
370                 }
371             }
372         }
373         printf("%d records are selected... \n",recordcounts);
374     }
375     //if there has any records left in the output buffer, push into the out vector
376     if (bufferOut->nreserved != 0) out.push_back(*bufferOut);
377     printf("End of selection operation.... \n");
378
379 }
380
381 #endif
382
```

```
1  /*
2  * DBMS Implementation
3  */
4
5  #ifndef BUFFEROPS_H
6  #define BUFFEROPS_H
7
8  #include <io.h>
9  #include <fcntl.h>
10 #include <sys/types.h>
11
12 #include "dbmsproj.h"
13 #include<iostream>
14
15
16 // empties a block
17 template<typename T> void emptyBlock(block_t<T> *buffer, unsigned int size) {
18
19     buffer->nreserved = 0;
20     buffer->entries.clear();
21     //printf("empty buffer block... \n");
22 };
23
24 // empties the whole buffer
25
26 template<typename T> void emptyBuffer(block_t<T> *buffer, unsigned int size) {
27     for (uint i = 0; i < size; i++) {
28         emptyBlock(buffer + i);
29         buffer[i].valid = true;
30     }
31     printf("empty buffer ...");
32 };
33
34 // opens filename for writing (append mode), and writes size blocks
35 // starting from pointer buffer
36
37 template<typename T> unsigned int writeBlocks(block_t<T> &relation, block_t<T> &buffer,
     unsigned int offset, unsigned int size) {
38     unsigned int writecounts = 0;
39
40     for (size_t i = 0; i < size; i++)
41     {
42         relation[i+offset] = buffer[i];
43         writecounts++;
44     }
45     printf("Write %d blocks into disk...\n", readcount);
46     return size;
47 };
48
49 // reads size blocks to buffer
50
51 template <typename T> unsigned int readBlocks(std::vector<block_t<T>>& relation,
     block_t<T>* buffer, unsigned int size) {
52     unsigned int readcount = 0;
53     for (unsigned int i = 0; i < size; i++)
54     {
55         if (i<relation.size())
56         {
57             buffer[i] = relation[offset + i];
58             readcount++;
```

```
59          }
60          else
61          {
62              break;
63          }
64      }
65      printf("loading %d blocks into buffer...\n",readcount);
66      return readcount;
67  };
68  template <typename T> unsigned int readBlocks(std::vector<block_t<T>>& relation,
      block_t<T>* buffer,unsigned int size,unsigned int offset) {
69      unsigned int readcount = 0;
70      for (unsigned int i = 0; i < size; i++)
71      {
72          if ((offset+i)<relation.size())
73          {
74              buffer[i] = relation[offset+i];
75              readcount++;
76          }
77          else
78          {
79              break;
80          }
81
82      }
83      printf("loading %d blocks into buffer...\n",readcount);
84      return readcount;
85  };
86  #endif
87
```

```h
/*
* DBMS Implementation
*/

#ifndef RECORDPTR_H
#define RECORDPTR_H

#include <sys/types.h>
#include "dbmsproj.h"

// location of a specific record in the buffer
// block is the index number of the block the record is in
// record is the index number of the record in the entries table

typedef struct {
    unsigned int block;
    unsigned int record;
} recordPtr;

// struct for making linked lists of recordPtrs

struct linkedRecordPtr {
    recordPtr ptr;
    linkedRecordPtr *next;
};

//overloading operators for use with struct recordPtr
inline bool operator==(const recordPtr &ptr1, const recordPtr &ptr2) {
    if (ptr1.block==ptr2.block && ptr1.record==ptr2.record){
        return true;
    }
    else return false;
};
inline bool operator<(const recordPtr &ptr1, const recordPtr &ptr2) {
    if (ptr1.block == ptr2.block) {
        if (ptr1.record < ptr2.record)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else
    {
        if (ptr1.block < ptr2.block)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

}
inline bool operator<=(const recordPtr &ptr1, const recordPtr &ptr2)
{
    if (ptr1 < ptr2 || ptr1 == ptr2) { return true; }
```

```
61      else return false;
62  }
63
64  inline recordPtr newPtr(recordPtr ptr,unsigned int offset,unsigned int size) {
65      recordPtr result;
66      result.block = ptr.block + offset / size;
67      int rest = offset % size;
68
69      if (ptr.record + rest >= size) {
70          result.block += 1;
71          result.record = ptr.record + rest - size;
72      }
73      else {
74          result.record = ptr.record + rest;
75      }
76      return result;
77  }
78
79  inline recordPtr newPtr(unsigned int offset,unsigned int size) {
80      recordPtr zero;
81      zero.block = 0;
82      zero.record = 0;
83      return newPtr(zero,offset,size);
84  }
85
86  // increases the ptr so that it points to the next record
87  // if pointing at the end of a block, moves to the start of the next
88
89  inline void incr(recordPtr &ptr,unsigned int size) {
90      if (ptr.record < size - 1) {
91          ptr.record += 1;
92      } else {
93          ptr.record = 0;
94          ptr.block += 1;
95      }
96  }
97
98  // decreases the ptr so that it points to the next record
99  // if pointing at the start of a block, moves to the end of the previous
100
101 inline void decr(recordPtr &ptr,unsigned int size) {
102     if (ptr.record > 0) {
103         ptr.record -= 1;
104     } else if (ptr.block > 0) {
105         ptr.record = size - 1;
106         ptr.block -= 1;
107     }
108 }
109
110 #endif
```

```
/*
* DBMS Implementation
*/

#ifndef RECORDOPS_H
#define RECORDOPS_H

#include <string.h>

#include "dbmsproj.h"
#include "recordPtr.h"

// given a buffer and a recordPtr, returns corresponding record

template <typename T> T getRecord(block_t<T> *buffer, recordPtr ptr) {
    return buffer[ptr.block].entries[ptr.record];
};

// given a buffer, a record and a recordPtr, places the record where recordPtr points

template <typename T> void setRecord(block_t<T> *buffer, T rec, recordPtr ptr) {
    buffer[ptr.block].entries[ptr.record] = rec;
};

// given a buffer and 2 recordPtrs, swaps the records where ptrs point

template <typename T> void swapRecords(block_t<T> *buffer, recordPtr ptr1, recordPtr ptr2) {
    T tmp = getRecord(buffer, ptr1);
    setRecord(buffer, getRecord(buffer, ptr2), ptr1);
    setRecord(buffer, tmp, ptr2);
};

// given 2 records and field, compares them
// -1 is returned if rec1 has lower field value than rec2
// 0 is returned if rec1 and rec2 have equal field values
// 1 is returned if rec1 has higher field value than rec2
template <typename T1, typename T2> int compareRecords(T1 &rec1, T2 &rec2, std::string const& field) {
    if (rec1.getCol(field) == rec2.getCol(field)) { /*printf("Two records matched! \n");*/ return 0; }
    else return -1;
}

// hash function for integers

inline unsigned int hashInt(unsigned int num, unsigned int mod, unsigned int seed) {
    num += seed;
    num = (num + 0x7ed55d16) + (num << 12);
    num = (num^0xc761c23c) ^ (num >> 19);
    num = (num + 0x165667b1) + (num << 5);
    num = (num + 0xd3a2646c) ^ (num << 9);
    num = (num + 0xfd7046c5) + (num << 3);
    num = (num^0xb55a4f09) ^ (num >> 16);
    return num % mod;
}

// hash function for strings

```

```
57  inline unsigned int hashString(std::string str_text, unsigned int mod, unsigned int seed)
      {
58
59      char *str = new char[str_text.length() + 1];
60
61      std::strcpy(str, str_text.c_str());
62      str[str_text.length()] = '\0';
63      unsigned long long hash = 5381;
64      int c;
65
66      while ((c = *str++)) {
67          hash = ((hash << 5) + hash) + c;
68      }
69      return hashInt(hash, 8701123, seed) % mod;
70  }
71
72  // given a record and the field of interest, hashes it and returns a value
73  template<typename T> int hashRecord(std::string seed, T rec, unsigned int mod, std::string
       const& field)
74  {
75      unsigned int s = hashString(seed, 8701123, 0);
76      std::string test = rec.getCol(field);
77      return hashString(rec.getCol(field), mod, s);
78  }
79
80  // frees memory allocated for a hash index
81  void destroyHashIndex(linkedRecordPtr **hashIndex, unsigned int size);
82
83  #endif
84
```

```
1  /*
2  * DBMS Implementation
3  */
4  #include "stdafx.h"
5  #include "recordOps.h"
6  #include <stdlib.h>
7
8  // frees the memory allocated to a hash index
9
10 void destroyHashIndex(linkedRecordPtr **hashIndex, unsigned int size) {
11     for (unsigned int i = 0; i < size; i++) {
12         if (!hashIndex[i]) {
13             continue;
14         }
15         linkedRecordPtr *temp1 = hashIndex[i];
16         linkedRecordPtr *temp2 = temp1->next;
17         while (temp1) {
18             free(temp1);
19             temp1 = temp2;
20             if (temp2) {
21                 temp2 = temp2->next;
22             }
23         }
24     }
25     free(hashIndex);
26 }
27
```