```
#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 }
```