Inverted Search Engine Optimization

An Inverted Search Engine Optimizer based from the SIGMOD 2013 Contest

Compilation & Execution

In order to provide easier compilation a Makefile has been included. The Makefile accepts the following commands:

- `make clean` to clean the project's object files and executable
- `make clean_test` to clean the test's object files and test executable
- `make` to compile the project's files
- `make test` to compile the test files

Job Scheduler

> Complexity : **O(1)** insertion and removal in the queue (head and tail pointers).

Job Scheduler contains a thread-safe queue that allows to schedule jobs.

```
class JobScheduler {
  Queue *queue; // Queue of Jobs
  long numThreads; // Number of threads
  char lastJobType; // Last job type
  bool waitAllAndSignal; // Wait all and signal the main threa that all
jobs are done
  bool waitAll; // Wait all
  pthread_t *threadIDs; // Thread IDs
  pthread_mutex_t mutex; // Mutex for accesing the queue
  pthread_mutex_t condMutex; // Conditional mutex for signaling main thread
  pthread_cond_t cond; // Conditional variable for signaling main thread
  pthread_barrier_t barrier; // Barrier to wait for all threads
  pthread_barrier_t barrierAfter; // Barrier to wait for all threads
}
```

```
class Job {
  char type; // Type of job ['s','m','e']
  int status; // Status of job [0: dont execute it, 1: execute it]
  ErrorCode (*function)(int, void **); // Function to execute
  void **args; // Arguments to pass to function
  int numArgs; // Number of arguments
}
```

Job Scheduler's Parallelism Functionality

In order to parallelize the scheduler we group together the jobs of the same type and execute them in parallel. When the group type changes, jobs that are of different types have to wait until all the jobs of the previous type are done.

> For example if we try to execute a match document (m) job but a previous start query (s) job hasn't yet been completed, there is a possibility for the match document result to not contain the results of the previous start query.

It is obvious for the above example that by grouping together jobs of the same type we manage to prevent inconsistencies that can lead to wrong results from the tester.

- **Implementation of Waiting Threads**

  When a thread is attempting to get a job from the scheduler and that job has a different type than the current group jobs type that are the other threads are executing we set a variable to true. This allows the other threads to wait (provided they have finished executing their jobs) in a barrier with all the other threads. Once all the threads have finished executing their jobs we set the variable to false and the normal job execution flow is restored.

- **Synchronization of the Main Thread's GetNextAvailableRes Function**

  The problem: The main thread's match document function is built in a way such as each thread's code is executed asynchronously and not immediately. That creates the problem that after the creation of the match type jobs, the main thread will jump immediately to the GetNextAvailableRes function and try to get all the finished documents, which will almost certainly not be available.

  The solution: By utilizing a condition variable we manage to force the main thread to wait until all the match document jobs have finished. When the jobs are done executin,g the threads signal that condition variable to let know the main thread that the results are available.

- Joining the threads

  To join the threads we add NUM_THREADS*job to the job scheduler with type '.' and when the threads come across this job they will know that they have to exit.

Modifications in Core Functions

StartQuery now inserts a job in the Job Scheduler with type **'s'** and the function to execute the **query insertion**.

EndQuery now inserts a job in the Job Scheduler with type **'e'** and the function to execute the **query deletion**.

MatchDocument now inserts a job in the Job Scheduler with type **'m'** and the function to execute the **document matching.**

Mutexes Placement in Methods

In order to optimize the program's performance we placed the mutexes as close as possible to the methods that belong to the critical section. That ensures that reading tasks are not blocked by writing tasks and reduces the number of locks and unlocks.

Core Functions

Query Insertion

The main purpose of the StartQuery function is to take each word token of each query and store it depending on the MatchType in the corresponding Structure (HashTable for Exact Match, BK_Tree for Edit Distance, Hamming Array for Hamming Distance). In order to achieve this, it utilizes helper structs mainly the ExactInfo and the HEInfo to store in these three structs the information necessary for each word regarding the query it appears in.

> The maxQueryId property in the main structs object that is incremented for each query iteration will later be used as an index for the MatchArray array size.

Optimizations & Changes

- For each query we created an object that describes the query. This object remains the same during the whole program scope (we use a reference to the object itself). By implementing it in this way, when we change that object it changes everywhere (reduces time). Also it takes up less memory.
- Since now multiple queries can be inserted at the same time, we replaced strtok() with strtok_r(), since strtok cannot handle multiple threads.
- For the exact match (hashTable) we enclosed the SHA1 hashing function with mutexes, because it cannot be done in parallel. However locking and unlocking the mutex several times is not efficient. So we changed the SHA1 with a simpler mathematical function. This resulted in a significant drop in execution time.

Document Matching

The MatchDocument Function tokenizes the document into the words it contains and tries to find matches between these words and the words in the queries we processed in StartQuery.

- For the ExactMatch type we hash the word token and search in the respective bucket for a word that matches it. If there is a match we add that word and the exact info list that contains the info of all the queries the matched word appears in, into the match array to add it later to the matched documents (excluding the queries that have been deleted/flagged).

  > Time Complexity of finding the right bucket : O(1)

  > Time Complexity of finding the right word inside the bucket : O(n)

- For the Hamming/Edit types we use the BK_Tree lookup function which does the following: Instead of calling the lookup function multiple times for all the possible threshold values(1-3) we do the following:

    - We begin with the minimum threshold(1 for root/ threshold parameter for subtrees) and for each iteration we compare the match type distance with the threshold iterator. If it is a match we update the match array with the info list of the word.

    - We then iterate for the starting threshold up to the maximum threshold(3) and compare the edge weights (distance between the child and its parent node) to see if we are inside the distance range. If the weight is inside the range we recursively call the lookup function **using the current threshold iterator value as a starting threshold for its subtree.**

        > If we call the lookup function for a tree node with a specific threshold value e.g. 2 , it is certain that it will also be called for its subtree with all the possible threshold values greater than 2. This is used for optimizing the stack calls of the lookup function for each possible threshold and checking only the values necessary for matching the word token.

        > Time Complexity of the tree lookup function : O(log(n))

- Finally we construct the document with the matched query ids that are stored inside the matchArray and we push it into the document list that stores all the matched documents.

    <u>Optimizations & Changes</u>

    - By having the match array stored on the heap we have dedicated memory space for each document match. This is important because each thread has its own match array in its own memory space and manages to avoid memory conflicts with the results of the other arrays. This makes the use of mutexes (locks & unlocks) redundant in the lookup functions of the Edit & Hamming BK trees and the hash table. **The above design makes the whole document matching process completely parallelized, thus ensuring a fast execution time.**
    - The only exception that requires the use of a mutex lock/unlock is when we store the results inside the document list. This is because the document list is a shared resource between all the threads and we need to ensure that the document list is not being modified by the other threads.
    - All of the results are stored into the Match Array. The Match Array contains [max query] times Match Trees. By having one Match Tree per query id , we can gather all of the words of a specific matched query id into one Match Tree. When we have matched a word of a query we can easily insert it (O(logn)) into the query's Match Tree and at the same time know if we have already matched the same word to not count the same word twice. When we are inserting a new word into the match tree we also check if that query is completed (all the words of the query have been matched) and if so we add the query id to the result list. **By updating the result list at the same time as the insertion of the word we can avoid the traversal of the whole match array to find the queries that are matched, reducing drastically the execution time.**

GetNextAvailableRes

All the results that are created in the Document Matching process are added in O(1) time to the document list. The document list stores the document id and the result list for each corresponding document. We can easily access the document list and obtain the result of the first document, also in O(1) time. This makes the GetNextAvailableRes function substantially faster. In addition, this function also removes the necessity of mutexes (lock/unlock) since the threads are not accessing the document list at the same time.

Query Deletion

During the initialization and insertion of each query we store a dedicated object that is describes that query. Inside this object we store a boolean flag (true = active, false = deleted) that is by default true. These objects are stored in a hash table for easier and faster access. When a query has to be deleted we use the hash table to find the relevant query object that describes it and set the flag to false.The deletion time is **O(1)** because we only have to find the query object in the hash table and change the flag to false. Now anywhere there is a reference to that query object we can see that the flag is set to false. However, when we try to match a document we have to add an additional check to see if the flag is true. If it is false we know that the query has been deleted and we can skip that query.

Design Considerations of Data Structures

Encapsulation of Data

In order to encapsulate the data for each query we use two lists: the exactInfoList and the HEInfoList. Each list contains a list of ExactInfo and HEInfo objects respectively.

```
typedef struct ExactInfo {
  QueryID query_id; // the query id of the query
  bool flag; // a flag indicating if the query is active or deleted
  unsigned int maxQueryWords; // the number of words in the query
} ExactInfo;

class exactInfoList {
  exactInfoNode *head; // the head of the list
  exactInfoNode *last; // the last node of the list
  int count; // the number of nodes in the list
  pthread_mutex_t mutex; // the mutex lock for the list
};

typedef struct HEInfo {
  QueryID query_id; // the query id of the query
  bool flag; // a flag indicating if the query is active or deleted
  unsigned int maxQueryWords; // the number of words in the query
  unsigned int matchDist; // the match distance of the query
} HEInfo;

class heInfoList {
private:
  heInfoNode *head; // the head of the list
  heInfoNode *last; // the last node of the list
  int count; // the number of nodes in the list
  pthread_mutex_t mutex; // the mutex lock for the list
};
```

For example in the Hash Table ,where we hash a word of a query,that word can also belong to other queries. In conjunction with the word itself we also store a list of queries that contain the word. This list is stored in the bucket node. This ensures that ,when we need to obtain all the queries that this word appears in, we can easily retrieve them from that list.

Match Array

The Match Array is a Hash Table, where the buckets are not linked lists but Match Trees. The Hash Table is taking as input a query id, hashes it, and links it to the equivalent Match Tree bucket. Each Tree in the Hash Table signifies that it stores the results of one specific query.

To store a matched word of a query we use the hash table's lookup function in O(1) time to find the match tree of the query. Then we use the tree's insert function in O(logn) time to insert the word into the match tree. Duplicates are not allowed in the match tree.

When we insert a word into a Match Tree we also check if the query is completed (all the words of the query have been matched) and if so we add the query id to the result list.

```
class MatchArray {
  MatchTree **array; //array of match trees
  int size; // size of the array
  ResultList *matchedIds; // the list of matched query ids
};

class MatchTree {
  MatchTreeNode *root; // the root of the tree
  int maxWords; // the maximum number of words in the query
  bool flag; // a flag indicating if the query has been already matched
  int count; // the number of words in the tree
};

class ResultList {
  ResultListNode *head; // the head of the list
  ResultListNode *last; // the last node of the list
  int count; // the number of nodes in the list
  pthread_mutex_t mutex; // the mutex lock for the list
};
class ResultListNode {
  int id; // the id of the query
  ResultListNode *next; // the next node in the list
};
```

Query Hash Table

During the initialization and insertion of each query we store a dedicated object that is describes that query. Inside this object we store a boolean flag (true = active, false = deleted) that is by default true. These objects are stored in a hash table for easier and faster access. When a query has to be deleted we use the hash table to find the relevant query object that describes it and set the flag to false.

> The deletion time is **O(1)** for finding the bucket that contains the query and **O(n)** to locate the query in the list.

> The space complexity is **O(n)** where n is the number of queries.

> The default size of the hash table is **500K**. That means that the size of each bucket remains small even if the number of queries is large.

```
class HEQueryHashTable {
  HEQueryBucket **table;
  int size;
};
class HEQueryBucketNode {
  HEInfo *info;
  HEQueryBucketNode *next;
};
class HEQueryBucket {
  HEQueryBucketNode *head;
  HEQueryBucketNode *last;
  int count;
  pthread_mutex_t mutex;
};


class ExactQueryHashTable {
  ExactQueryBucket **table;
  int size;
};
class ExactQueryBucket {
  ExactQueryBucketNode *head;
  ExactQueryBucketNode *last;
  int count;
  pthread_mutex_t mutex;
};

class ExactQueryBucketNode {
  ExactInfo *info;
  ExactQueryBucketNode *next;
};
```

Document List

By using head and last pointers in the Document List and a previous pointer in each Document Node, we can easily add and remove documents from the list in O(1) time. Also, the space complexity of the list is O(n).

```
 //Document List
class DocumentList {
 DocumentNode *head; // head of the list
 DocumentNode *last; // last node of the list
 pthread_mutex_t mutex; // mutex for the list
 int count; // list size
};

 //Document Node
 class DocumentNode {
 Document doc; // document
 DocumentNode *next; // next node
 DocumentNode *prev; // previous node
 };
```

Hash Table & Bucket

The default Hash Table can contain up to 500K Buckets. We create all of the buckets before start time , ensuring that the hash table is ready to use. We use a hash function to map the word token to the bucket. We use a linked list to store the words in the bucket. We also use a mutex lock/unlock to ensure that the bucket is not being modified by the other threads.

```
class HashTable {
 Bucket **table;
 int size;
 pthread_mutex_t mutex;
 };

 class Bucket {
 bucketNode *head;
 bucketNode *last;
 int count;
 pthread_mutex_t mutex;
 };
 class bucketNode {
 String *word;
 exactInfoList *list;
 bucketNode *next;
 };
```

🚅 Performance

Time Complexity

All of the following time measurements are calculated for 10 iterations of the program and averaging the results. The program's execution was done in the Linux environment with 4 CPU cores.

- **Serial Execution of the Program (1 thread + main thread)**

  The time was **1395ms**.

- **Parallel Execution of the Program (2 threads)**

  The time was **771**ms.

- **Parallel Execution of the Program (3 threads)**

  The time was **588**ms.

- **Parallel Execution of the Program (4 threads)**

  The time was **522**ms.

- **Parallel Execution of the Program (5 threads)**

  The time was **520**ms.

- **Parallel Execution of the Program (6 threads)**

  The time was **531**ms.

- **Parallel Execution of the Program (8 threads)**

  The time was **540**ms.

- **Parallel Execution of the Program (16 threads)**

  The time was **630**ms.

- **Parallel Execution of the Program (32 threads)**

  The time was **704**ms.

- **Parallel Execution of the Program (64 threads)**

  The time was **830**ms.

Conclusion By exceeding the maximum number of threads that the CPU can handle it is clear that the load that the OS has to handle is too high thus the perfromance drops.

> The number of threads are inversely proportional to the execution time of the program. For example by doubling the threads from 2 to 4 we get almost the halv execution time. Since the machine has 4 cores, it is clear that the best results come from using 4 threads (1 thread per core).

Space Complexity

By running the program with valgrind we saw that the bytes allocated was **4.100.000.000 bytes = 4MB**

We can see that the optimazation of the exactInfo and heInfo being referenced by every structure worked and our program allocates only 4MB.

Contributors

- Nikolas Iliopoulos (1115201800332)
- Michalis Volakis (1115201800022)