Inverted Search Engine Optimization

An Inverted Search Engine Optimizer based from the SIGMOD 2013 Contest

:building_construction: 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 clean_all` to clean all files and executables
- `make` to compile the project's files
- `make test` to compile the test files

Job Scheduler | Job | Queue

Job Scheduler contains a thread-safe queue that allows to schedule jobs. The queue has **O(1)** insertion and removal time complexity by having head and tail pointers.

```cpp
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
}
```

```cpp
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 executing the threads signal that condition variable to let to know the main thread that the results are available.

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 use them. That ensures that reading tasks are not blocked by writing tasks and reduces the number of locks and unlocks.

Core Functions

StartQuery

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.

MatchDocument

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. To optimize this we used the lookup methods for all 3 data structures mentioned above:

- 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).

- 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.

-

- 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.

GetNextAvailableRes

We only modified the GetNextAvailableRes function to support our document list object

EndQuery

//todo complete endquery description

Optimizations & Changes

Query Insertion

- 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).

- Since now multiple queries can be inserted at the same time, we replaced strtok() with strtok_r() to avoid memory errors.

- 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

- 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.**

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.

🚄 Performance

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 **0.00027**ms.

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

  The time was **0.00027**ms.

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

  The time was **0.00027**ms.

- **Parallel Execution of the Program (4 threads)**
  The time was **0.00027**ms.

- **Parallel Execution of the Program (5 threads)**
  The time was **0.00027**ms.

- **Parallel Execution of the Program (6 threads)**
  The time was **0.00027**ms.

- **Parallel Execution of the Program (8 threads)**
  The time was **0.00027**ms.

- **Parallel Execution of the Program (16 threads)**
  The time was **0.00027**ms.

- **Parallel Execution of the Program (32 threads)**
  The time was **0.00027**ms.

- **Parallel Execution of the Program (64 threads)**
  The time was **0.00027**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).

:desktop_computer: Contributors

- [Nikolas Iliopoulos](#) (1115201800332)
- [Michalis Volakis](#) (1115201800022)