**ROLLSIDE SEARCH**

The search is primarily developed for the music site [www.rollside.com](http://www.rollside.com) for searching and recommending music; but the search has the capability to extend itself on various different requirements of different kind of searches. The prerequisites for understanding this approach are:

1. Signal Handling in LINUX
2. C++
3. Data Structures and Algorithms
4. Synchronization Techniques(Semaphores)

**Search Algorithm**

**IDEA:** For every search we try to minimize the area of search and find out the most common entries which come at top of result.

**Search Layers and Processes:**

**PREPROCESSING**: The search input is preprocessed that is the search string passed by user is refined, unwanted spaces and characters which are not indexed for search or are not required. The search string can only have characters (a-z), (A-Z) and (0-9); any combination of these.

**SEARCH SCHEDULING**: The search system has intermediate processes to communicate with infinite search process known as ‘middleman’; no direct work can be accomplished, use of middleman is mandatory.

The middleman forward the search request to search process when the search is free to read the buffer file ‘searchInput.txt’ until then it waits for the search to be free for a fixed amount of time. If the wait period of middleman exceeds the allotted time then the request is cancelled and the middleman dies. Otherwise it writes the search request in the buffer and signals the search to execute the search thread.

**READ SEARCH INPUT**: The search process read the search input file content or search string corresponding to a particular search request ID.

**SearchRequestID** is generated before the search takes and it is unique for every search request, this helps in writing the search output when the thread completes its execution.

The search string is broken into individual words, each word is considered to be separate entity for search. After reading the search input, search thread is executed and search is freed for accepting next input.

**FINDING KEYWORD:** The search is based on an advanced data structure ‘TRIE , every word that is available and can be searched is indexed inside the data structure. So for every perfect match keyword validation is done in linear time. There arise total of 3 cases corresponding misspelled keywords:

1. The keyword is perfect match.
2. No such keyword exists.
3. The keyword needs to be checked for resemblance inside the data structure in optimum time.
   1. Replacement policy on the basis of same sound of characters or strings
   2. Partial correction for maximum preference word with same prefix.

**DIRECTORY ACCESS:** Every validated keyword is provided with the corresponding flags in this case the flags represent song, band, artist, type and album, every valid keyword will either have one flag or multiple flags set. Corresponding to each flag their directories are accessed. Every file in directory represents a particular keyword. Each file has a definite file structure.

File structure:

1. SongID: Unique id for every song.
2. Hash Value: hash value corresponding to song id.
3. Position of word: Position of word in actual string

**SORTING:** The input is taken in the form of songID and hash value that is stored in structure corresponding to each entry in the node there is count value, for multiple keywords the songID which is common in those keywords will have greater count value. The nodes are sorted corresponding to the count value in descending order and top result is collected.

**WRITE RESULT:** The sorted nodes are selected and the top result example top 10 songID are printed in the Output folder. Every output file is named as **SearchRequestID** and corresponding middleman is signaled that search is completed.

**ACCESS DATABASE:**  As now the file has top results there is possibility to pin point the database search means there is no requirement to use LIKE query just look for exact match of songid in the database which requires very less time in comparison to searching every possible entry. This may look like Proxy Design Pattern. Although if all the information that is required to correctly display a result is available in files then the search may never need to access database for searching purpose otherwise for keeping a backup, database is essential.

**Search Process**

The search is an infinite running process; there are 2 infinite search processes in execution; parent and child process with different responsibilities. The infinite process makes it faster and providing infinite learning capabilities.

**Parent Process:** Parent process has logical contribution to search responsibilities; the major role of parent comes when due to some reason or bug example segmentation fault, the child process dies. The parent keeps a logical copy of the data structure and instantly creates a new process with same configuration as that died process that is it forked to create new process.

**Child Process:**  The child process actually performs all the important operation including logical and physical; every change in output can be reflected by making change in this process although parent and child both have same logical data but parent does not perform any physical operation. By physical operation it means accessing directories and keyword files for searching and adding new entries. No middleman or intermediate process has direct communication with child process, all the request are first logically processed by parent and the request forwarded to child process, this sequential working of search within the 2 process but if synchronization is not required then they can run in concurrency.

**Responsibilities**

**Reconstruction of Trie:** The search when started initially does not contain any data or information ‘logically’ and until it has no logical data search cannot be performed. The data is fetch from the ‘trie’ table in database to fill up the trie data structure. Here it is assumed that the search was previously in operation to some error or enhancement in activity it was shut down and now restarted; the physical data is not disturbed. The data structure trie is provided with all the words and flags required for search.

**Release Search:** Once the reconstruction is completed the search is released and made open for receiving requests and providing response to the external environment.

**Search:** The search is based on keyword search, the trie data structure deals with each keyword individually without considering any relation between the keywords if request contains multiple keywords. The trie data structure provides validation of the keywords establishes a relation between keywords when requested is dealt at physical level by common hash value.

**Add Entry (Update):**  Adding a new value to search is both logical and physical, logical is updating all the keywords in the entry into trie data structure and Trie table in table along their corresponding flags. Physical update involves making entry to the corresponding keyword files in their respective flag directories.

**Update Preference:** The search is learning infinitely the preference of keywords that are searched frequently in the Trie Table and at the end of every day the trie data structure is updated with new preference which is a floating point value. This is how trend is reflected in the search.

**Building Preferences:** Build preference is an advanced feature which is added as functionality in the trie data structure to abolish delays in finding the max preferred words for providing recommendations, the trie is modified and segmented. Each node preference stored. Build preference operation is a very costly which can be performed once to recognize trend change update preference and build preference are interdependent operations.

**Availability:** The involvement of middleman or intermediate processes to communicate creates dependency halts the process when the search semaphore gets locked and the middleman dies; search remains in locked state so here multiple requests comes into use which detects the current state of search process and overrides the previous state of the search process and begins with new search request. This approach makes the search process available for receiving request and responding.

**Compilation Commands**

The names of the compiled files should be kept same as there further dependent modules which uses the same names in compiled form.

**Search: g++ -pthread Search.cpp –o search**

**Search Middleman: g++ -pthread SearchMiddleMan.cpp –o smm**

**Reconstruction Middleman: g++ -pthread ReconstructMiddleMan.cpp –o rmm**

**Update Middleman: g++ -pthread UpdateMiddleMan.cpp –o umm**

**Preference Middleman: g++ -pthread PrefMiddleMan.cpp –o pmm**

**Build Preference: g++ -pthread BuildPref.cpp –o bp**

**Release Search: g++ -pthread ReleaseSearch.cpp –o rs**

**Execution and Passing Arguments**

**Search:** No arguments are passed; the process is infinite and receives values only through buffer files.

**Search Middleman:** ./smm request\_id search\_request

request\_id: unique id for every request.

Search\_request: search string composing of keywords

**Update Middleman:** ./umm songid , sngname , art1 art2 art3 , album , band , type

Max 3 artist can be indexed for one song, this constraint comes from the front end of the site otherwise any number is allowed by search.

**Reconstruction Middleman**: ./rmm done word song\_flag artist\_flag album\_flag type\_flag band\_flag pref

Done is set ‘1’ for last word otherwise ‘0’

**Preference Middleman:** ./pmm word flag pref

**Build Preference:** ./bp

**Release Search:** ./rs

**File Descriptions**

**executionsearch:** Storing execution time for search for analysis.

**Executionupdate:** Store time value for adding new entry logical and physical for analysis purpose.

**MiddleManLog:** Every search middleman which has permission to write search input file make entry to MiddleManLog as its process\_ID. In case the search gets locked the died middleman can be checked from the log and its execution can be overridden.

**PrefMiddleManLog:** It meets the same purpose as that of above just the application shifts to preference intermediate processes.

**PrefWord:** Buffer file having keyword and the corresponding preference value in floating point.

**ReconFail:** In case some middleman fails to send signal to search then that keyword is added in this file. The process can be run again for these words.

**ReplacementPolicy:** This file includes the replacements based on same sound of certain characters. The replacement policy is one to one.

**SearchInput:** It’s the buffer file making search requests to the process. It contains request\_id and search String.

**SearchPID:** Process id of the search process (child process).

**SearchPPID:** Process id of the search process (parent process)

**TrieAddress:** Address of trie root node corresponding to the parent (not in use yet).

**TrieReconstruct:** Buffer file having keyword , corresponding preference value in floating point and the flag value.

**TrieReconstructLog:** Log for reconstruction processes.

**TrieUpdate:** Buffer file for adding new entry in the file.

**TrieUpdateLog:** Log for the update processes.

**Intermediate Processes**

Search works on client server architecture and itself follows the same. The requests are forwarded to middleman intermediate processes which act as clients to search process. Each middleman has a particular purpose and their respective buffer files if required to pass data along with corresponding signals. Following are the intermediate process:

**Search Middleman (smm):** Passes the search string to search process through search input buffer file when it is possible under time constraints of passing data, it waits until search semaphore is available and the signal is SIGALRM.

**Update Middleman (umm):**  Passes data to be updated in fixed format or sequence of arguments through TrieUpdate buffer file. The time limit for update is higher than the rest; this can also be the waiting period. It is synchronized by semaphore. The signal used is SIGUSR1.

**Reconstruction Middleman (rmm):** The reconstruction restricts the search process from receiving any further search and update requests. The buffer file TrieReconstruct which passes each keyword, all flag values and preference and constructs the logical data in the trie.All the middleman process work in synchronization. After receiving the last word search is released. Signal used is SIGUSR2.

**Preference Middleman (pmm):** The preference is updated in a similar fashion to reconstruction but in this case not all flags are passed. The buffer file is PrefWord and process follow sequence using semaphores. The signal used is SIGPROF.

**Build Preference (bp):** The preference has dependency over current value preference in trie; it locks the search and build pref trie in parent and child processes. The signal used is SIGURG.

**Release Search (rs):** Release opens all the search semaphore incases a situation occur which can stuck the system. It is used for handling exceptions.

The search is closed loop system, update is done in database and reflected in the search; it forms one way entry in the system with continuous feedback regarding new trend in keyword use. The search reflects answers in the form of songs; user preference is not for keywords actually, it’s the entity which forms the result that are songs so all words corresponding to that song are updated with preference to govern a trend change.