MAMgo Dev Manual

# Entities

MAMgo is a search engine, composed of entities that make up the entire system. The following are the elements in MAMgo:

**Database**

The name of the database used is ‘search’, and is composed for four tables:

* doc\_links – storing each download HTML document’s details
* tag\_index – storing the terms and their number of occurrence in each HTML document’s tags
* pos\_index – storing the terms and their position respective of the tag number in each HTML document’s tags
* queries – storing the list of queries entered by users, each having a count of times searched for

**Crawler**

Crawler is standalone, and so can work irrespective of any other entity in the system. The crawler works indefinitely and therefore should be manually stopped by admin.

**Indexer**

Indexer requires the database entity to be active. The indexer does not work indefinitely and halts when all documents available to it are indexed.

**Query Processor**

Query Processor requires the database entity to be active. The query processor works indefinitely and needs to be manually stopped by admin.

**Web UI**

The web UI can operate in two modes – normal and test. The normal mode is the normal operation of this entity, and requires the database entity to be active and the query processor entity to be running. The test mode is the test version of this entity, and only requires the database entity to be active.

The web UI is an even based entity, and so its code or parts of it runs based on the user’s usage of the system.

# Operation

* The crawler works indefinitely, crawling and downloading HTML documents from the World Wide Web into a folder.
* The indexer works by indexing all the HTML documents download from the crawler into the database, terminating upon indexing all documents.
* The query processor works indefinitely, and listens to a specified port, awaiting to receive a query search entered by the user in the web UI, to process it and create VIEW having the result set of the details of the required HTML documents to display to the user. The query processor responds back to the web UI confirming completion of operation.
* The web UI awaits the user to enter a query search into the search bar. Upon receiving one, the web UI sends query as a string over a port to the query processor, and awaits a response. Upon receiving a response from the query processor, it reads the result set in the VIEW created by the query processor, and displays the HTML document’s details as search records to the user
* The ranker…

# Justifications

***Indexing only the <title>, <h1, h2…h6>, <b>, <em>, <body> tags***

* <title> tag to keep record of what the HTML document is about in the first place
* <h1, h2…h6>, <b> and <em> tags to keep record of the importance of plaintext of those tags are in the HTML document
* <body> tag because it will cover the entire content of the HTML document.

NOTE

The <head> tag is not included because this tag is only meant to store information and metadata about the HTML document, but what interests the indexer is just the content

***Not eliminating stop words when indexing***

The elimination of stop-words had to be removed from the Indexer prior to the problem that would otherwise arise if the user types into the search bar a stop word, or if the phrase search of the user involves a stop word.

***Storing both the original and stemmed form of the word***

Previously, only the stemmed form of the word was stored, but this gave rise to a problem when the user phrase searches, which involves searching for the exact string literals in the same exact order of the search query, therefore, the original form of the word had to be stored as well.

***Not removing duplicates when indexing***

The removal of duplicates would have caused problems when storing the position of the words, and so was removed from the implementation of the indexer. Besides, keeping duplicates won’t cause problems as long as encountering the same term again just increases the count of this term.

***Separating the query processor from the web UI***

Honestly, this had to be done because the query processor is in Java, while the web UI is in HTML and CSS surely, but also in PHP. To solve this problem, a port was setup to allow for communication between the web UI (PHP) and the query processor (Java), treating web UI (PHP) as the client and the query processor (Java) as the server.

Even though this solution was meant to solve the original problem of different languages, it made way the search engine more memory efficient, though unfortunately, not more time efficient.

***Using a VIEW, and not sending the result set as an array over the port***

The following are the reasons:

* It’s better to keep the data where they originally are i.e. in the database instead of reading it, storing it as an array and sending the array to the web UI
* Fetching the entire result set into an array on the server side would consume a lot of memory, compared to later fetching the results at the web UI. Thus, by creating a VIEW, any sort of overhead from the size of the result set is eliminated from the query processor (server) and is only there, if any, at the web UI (client) depending on whether the UI will extract the entire result set of the VIEW or only parts of it at a time.
* Sending a very large array over a port could take a lot of time thereby keeping the port occupied, thus making the service unavailable during that period for other users
* Eliminates the need to create a session for the results, which could cause problems when multiple users are using the search engine

***Not displaying the required description snippet when query terms are either only in title or in stemmed form of the document***

* For the title, this is not considered content, it is only the title of the webpage, therefore, it sounds wrong to make a description snippet for it because the word doesn’t exist in the actual content of the page
* For the stemmed word, this is impossible to achieve efficiently following our research