Intelligent Browsing Application: Progress Report

Steve McHenry <mchenry7@illinois.edu> (Captain)

This progress report is regarding the "Indexing, Organizing, and Querying Collections of User-Selected Web Pages" intelligent browsing Chrome extension application, whose initial proposal is available at [1]. Progress has so far been steady with no interruptive challenges. Phase 1, the design phase, is complete, and Phase 2, the implementation and unit testing phase, is in progress. Phase 3, the system testing phase, will begin after Phase 2 is complete. This report will discuss the progress made in Phases 1 and 2 along with a discussion of Phase 3 plan developments based upon peer review feedback.

Phase 1, the design phase, has been completed. Here, I'll discuss the high-level architecture of the Chrome extension. The application consists of three components split into two sections: the frontend, containing one component, and the backend, containing the remaining two components. The backend components provide the application's text retrieval functionality. The primary backend component is the extension service worker. The service worker performs two main tasks. It maintains the inverted index by adding, updating, and deleting document entries from the index. It also performs ranked document retrieval against the index, using Okapi BM25. The second backend component is the database. The database, implemented as a JavaScript IndexedDB database, maintains the inverted index as well as document (webpage) metadata and user-created document collections. The use of server-side storage was considered, but ultimately deemed to be out-of-scope for the application's primary goal of implementing and demonstrating text retrieval functionality.

The frontend consists of the user interface pop-up which provides the graphical user interface with which to interact with the application. The frontend does not perform any processing functionality; it only sends requests to the backend based upon user input and displays the corresponding results returned by the backend. The pop-up allows the user to manage (add, update, delete) documents and collections as well as perform document searches via communication with the service worker. Communication between the pop-up and the service worker is implemented using extension message passing. Those familiar with extensions may notice that this application does not utilize a content script; this is because the application does not interact with the web site's DOM (with a minor exception noted later). No problems were encountered during this phase. However, because this is my first time working with both the Chrome extension and JavaScript IndexedDB APIs (I am not a native JS developer), there was a bit of an early learning curve as I spent some time familiarizing myself with the APIs and their current best practices.

Phase 2, the implementation phase, is currently in progress. Here, I'll provide an overview of implementation details for completed items and plans for remaining items. The text extraction and tokenizer functionality have been completed. When a user adds a document to the index via the "Add" button within the pop-up while viewing the target webpage in the browser, the pop-up extracts the webpage's body text content (the exception noted earlier) and passes it to the service worker which runs the text through the tokenizer framework. The tokenizer performs three steps - two of them being optional. First, the tokenizer optionally pre-formats the string. In my implementation, I provide a formatter that removes ASCII punctuation and non-letter symbols (except for number grouping and decimal separators). Second, the tokenizer splits the data into words. In my implementation, this simply splits over spaces as a delimiter (I assuming Latin-based language strings). Finally, the tokenizer

optionally performs stop word removal based upon a provided list of stop words. In my implementation, I provide a sample set of common English stop words. At this point, tokenization is complete, and the output is an array of tokens in the order in their original ordering.

The inverted index management functionality is currently in progress. The output from the tokenizer is used to populate or update existing entries in both the index's dictionary and posting lists. The dictionary and posting lists are both implemented as IndexedDB object stores. Documents' metadata elements – the IndexedDB-generated document ID, page title, URL, and insertion timestamp – are stored in another object store. Document collections – the IndexedDB-generated collection ID, name, and associated documents – are stored as yet another object store. Removing a document from the index is simply the reverse of the process: remove the document from the posting list and decrement the document frequency of each related term in the index (or remove the term if this document contained the only instance). Currently, Chrome will reserve up to 60% of the total disk size for IndexedDB [2], which far exceeds reasonable storage requirement expectations for single-user use of the application. Therefore, I do not have concerns about disk usage.

The implementation of the pop-up user interface is currently in progress. A first iteration of the user interface is currently in use to assist with the testing of the backend functionality. After backend functionality has been finalized, the frontend interface will be refined into its final production state.

The implementation of the inverted index retrieval functionality and document ranking functionality are the remaining Phase 2 tasks on the backlog. The inverted index retrieval scoring functionality will retrieve the top-*k* documents for a user-specified *k* using Okapi BM25. The retrieved documents' metadata will then be returned to the frontend in descending order of relevancy for visual display. At this point, no notable challenges have been experienced during Phase 2.

Phase 3, the system testing phase, is not yet in progress. It will begin upon the completion of Phase 2. For this phase, a collection of webpages, approximately 100, covering an arbitrary variety of topics, will be collected and programmatically inserted into the application. Then, several sample queries will be constructed. For each query, a relevance judgement will be assigned to each webpage. Each query will be executed, and the returned results will be compared against the respective relevance judgements to measure the performance of the queries and successfulness of the application. At this point, no challenges are expected during Phase 3.

Quick References

[1] https://github.com/stevemchenry/CourseProject
[2] https://support.google.com/chrome/thread/176382498/chrome-indexeddb-storage-limits?hl=en-GB