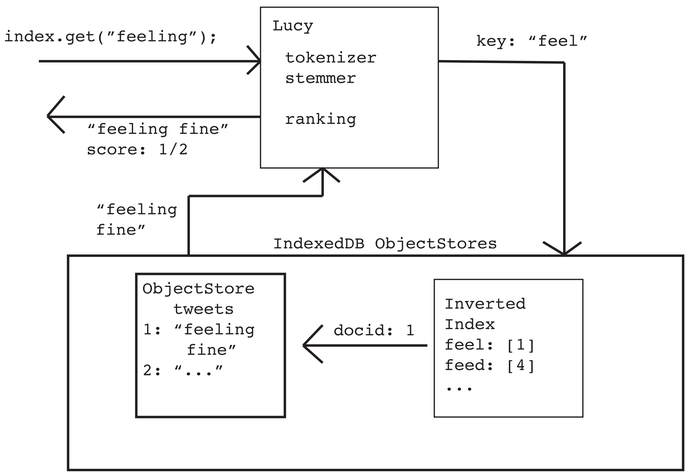
Lucy.js: Client-Side Indexes for Fast Full-Text Searching

Amy Zhang, Lea Verou, Manali Naik {axz, leaverou, mnaik}@mit.edu



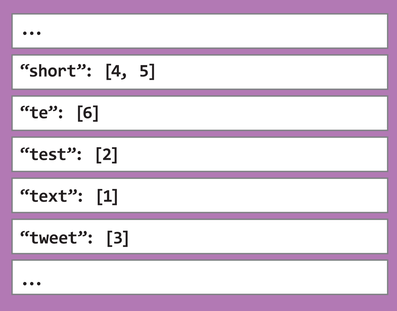
Uses:

- Fully client-side web applications (e.g. to-do lists)
- Client-server web applictions, for speed increase by reducing number of network roundtrips.

For example, a social web application (e.g. Facebook, Twitter etc) could store data related to the current user (posts, tweets) and use Lucy.js to search them.

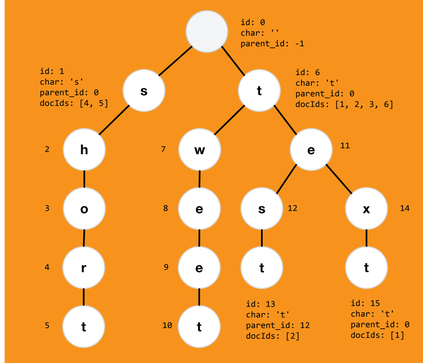
Types of indexes in Lucy.js:

Inverted index



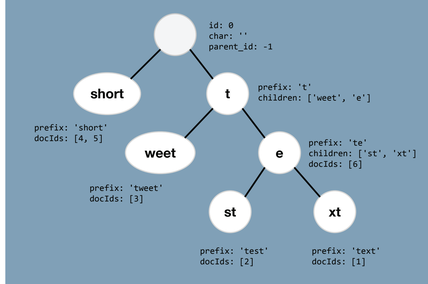
The keys of this index are tokens, words from documents. The values are a list of document ids that contain the token. This is a general purpose approach that performs well on a varieity of inputs.

Prefix & suffix trie

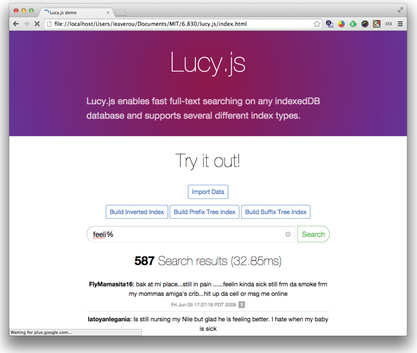


Every character is a separate node. Document ids are stored on every non-root node for its entire subtree. This approach works great for smaller datasets and bigger tokens.

Hybrid trie



Prefix paths are object store keys, with an array of strings to construct the children keys. Document ids are only stored on the leaves. Implements path and leaf shrink. This approach works great for bigger datasets and smaller tokens.



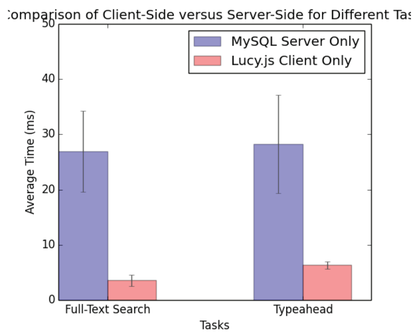


Chart: Comparison of client-side versus server-side for different tasks

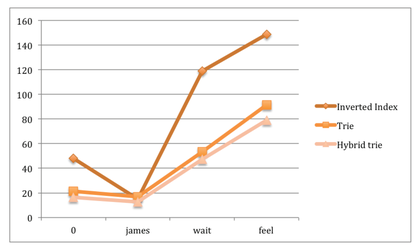


Chart: Query times (in ms) for 4 different queries, with all three index types

IndexedDB is a W3C specification for an asynchronous, multi-threaded client-side database, and is currently implemented in every modern browser.

Summary We implement fast client-side full text search by extending IndexedDB with several different types of optimized full-text indexes. We then demonstrate how this can help to significantly improve the speed of various types of web applications by reducing the number of network roundtrips.

Natural Language Processing We incorporate well-known text processing techniques such as **tokenization, stemming, and stopword removal**. We also support wildcard searches. Finally, we support **8 languages**.

Ranking We implement different document normalizations, as well as **cover density rank**, which takes into account query positions. Some of these call for us to store additional information in indexes, such as position, as well as document metadata, such as document length and number of unique words.

Real Life Tasks We implemented two real-life tasks - a full-text search bar and a typeahead search bar, and tested their performance using a remote MySQL database with full-text indexes against our client-side database indexes. After 8 trials with a randomly selected set of words on a small dataset of tweets, the client-side database was clearly faster.