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

Amy X. Zhang, Lea Verou, Manali Naik
MIT CSAIL
Cambridge, MA USA
{axz, leaverou, mnaik}@mit.edu

ABSTRACT

Many applications on the web today use a combination of client-side and server-side data stores to facilitate fast interactive and data-intensive experiences. However, standard client side databases within browsers do not currently support full-text searching, and it is not clear the best way to engineer full-text indexes on top of the existing infrastructure. In this paper, we introduce Lucy.js, a fully client-side search engine built as an extension of the existing well-supported IndexedDB API. We compare the performance of the different indexes we implement on different types of full-text content and queries. We find that a hybrid approach combining inverted indexes and tries perform the best, particularly on larger datasets. We also implement various real-life applications of our system and compare the performance of our system with that of using a fully server side system. We find that given the costs of network latency, our approach is significantly faster than relying on a server-side search engine. Finally, we discuss how our system may open up novel web application and data storage designs that solely use client-side search engines or combine both client and server search engines.

1. INTRODUCTION

A typical web application stores the bulk of its data on a backend web server or on the cloud, and the client side queries the server when it needs to access data. As web applications have become more interactive and responsive in the past several years, developers have built larger and more complex web servers to support them. However, this separation of data from the client introduces significant network effects that can often lead to noticeable latency, as large amounts of data may need to be retrieved from servers that are very far away.

In recent years, more applications have been built that heavily rely on client-side storage and offload data processing to web browsers. Today most commonly used browsers support the current HTML5 standards, which include a persistent client-side DBMS with the name IndexedDB that is accessible via a set of JavaScript APIs. The ability to use local storage enables real-time interactivity on data-intensive and collaborative tasks by reducing the number of network roundtrips. These have been useful for many large and complex web applications such as massive multi-player games and collaborative editing tools that would have previously only existed as desktop applications. Issues around privacy and ownership of data have also given rise to a number of applications that aim to move sensitive data analysis and data storage to the client [**?**, **?**]. Finally, most smartphone browsers rely on wireless signals to transmit information, which is prone to failure in areas with poor signal, suggesting that applications that are entirely client-side or applications that retains some functionality during downtime could be useful [**?**].

While much work has focused on making client-side storage and query easier [**?**], currently there are few known ways to conduct full-text searching on the client side. This is a problem because search engines are a very important component of a large number of applications and are necessary to make sense of many forms of full-text data such as chat or private messages, social media content, search queries, and webpage titles and content. As personalization has become more important for search engines, many designers of systems and architectures have highlighted the opportunities for client-side data computation [**?**, **?**]. Issues of privacy are also important when it comes to social applications and searching over personal data. However, while it is possible to store full-text content on the client side, there is still no way to conduct full-text search on the client side at a reasonable speed, preventing most data computation necessary for a fully-featured search engine. The current standard of IndexedDB, a key-value store, only supports simple operations such as put and get on set of keys and secondary indexes on object fields. There are no built-in capabilities to create indexes for full-text content.

In this paper, we introduce Lucy.js, a fully client-side JavaScript search engine that is an extension of the current IndexedDB API [**?**]. It has been designed to be easy for anyone currently using IndexedDB in their application to begin indexing and searching full-text content, without needing to modify pre-existing code or move to a new framework or mode of communication. There are also no dependencies on a server or network, so it can be harnessed offline. In our implementation, we created several types of indexes commonly used in well-known search engines, such as Lucene and within PostgreSQL, including inverted indexes, prefix and suffix tries, and variations on these. We discuss how we create these indexes on top of the IndexedDB framework as well as how the different indexes are suited to different tasks. We compare their performance on different types of queries and with different sizes and types of datasets. We find that...FILL IN HERE.

We evaluate our system on several real-world applications and compare performance of fully client-side and fully server-side data architectures. We find that on the whole, using client-side indexes significantly decreases the time to complete search queries due to the network latency brought on by communicating with a remote server. Finally, we consider potential novel systems and architectures that could be implemented that take advantage of a client-side search engine. This research demonstrates not only the feasibility of having fast search engines within the browser but also the improved performance of existing applications and the potential for new applications through the additional capabilities provided by a client-side search engine.

2. BACKGROUND AND RELATED WORK

Though previous HTML standards and browser capabilities were more limited, the idea of using of the client side for data storage and data computation has been around for many years [?]. Some research has looked into building tools to support an architecture that includes client-side databases, including toolkits to synchronize data between the client and server [?]. Another project facilitates ubiquitous access to client-side data across multiple browsers by enabling browser session migration from client to server to client [?].

When it comes to search engines, specifically web search, prior work has found that 40% to 60% of search queries are re-finding queries [?]. This finding has led to applications that store web content from the result of a search engine to improve speed and availability when the network is down [?, ?, ?]. Prior work in this area have also highlighted how web search can often be a highly personal activity, and that web search could be personalized to the user to improve accuracy. Personal information can then be stored on the client side as this information is primarily useful for this particular client [?]. Privacy is also a large concern for applications that collect personal data. Some applications have tackled personalized advertising using client-side stored profiles to mitigate this problem [?].

With the increased use of mobile phones in recent years, many web applications are now accessed via a smartphone. Here too, studies have shown that many queries to search engines are for re-finding. Using a client-side database is potentially even more impactful for a mobile phone browser because network connectivity is much less robust. Researchers have demonstrated this feasiblity by building a standalone search application for Android [?]. However, this search engine is not usable with other browsing apps or other phones. Because we build an extension on top of IndexedDB, our search engine will be able to work for all browsers, desktop or mobile, that support IndexedDB.

Some prototypes of client-side search engines have previously been built. However, all existing implementations only create basic inverted indexes. For instance, previous research has demonstrated that it is feasible to store an inverted index within IndexedDB [?] though it was found to

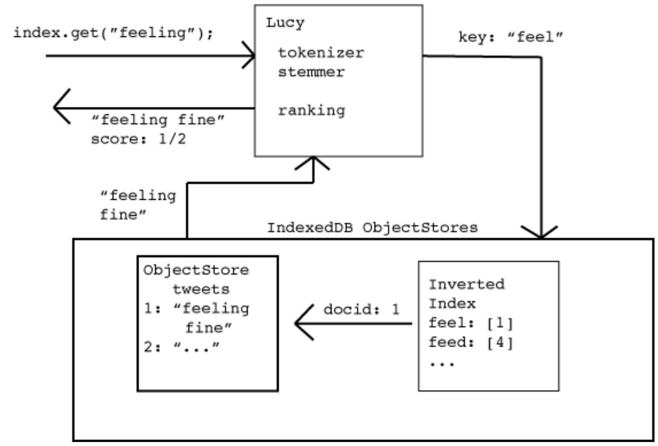


Figure 1: Architecture of Lucy.js

be much slower than using a server-side application such as Lucene. We improve on these works by implementing different types of indexes for full-text search and comparing their performance. Another existing implementation of a client side search engine called Lunr.js[1] stores indexes in the client-side memory, which is infeasible for large amounts of data and also non-persistent. Instead, our system uses IndexedDB object stores to hold indexes. Also, the YDN-DB module[2] which implements an inverted index, wraps around the IndexedDB API, requiring developers to migrate over to use the YDN system. Instead our system is simply an extension of the IndexedDB API, making it easy to integrate. We also demonstrate several real-life situations that may call for a hybrid client and server database architecture for full-text search and measure their speed.

3. ARCHITECTURE AND OVERVIEW

In this section, we explain the overall goals and architecture of the Lucy.js system, how it fits into IndexedDB, as well as give further details of our implementation. In Figure 1, the overall architecture is shown as well as a simplified workflow for a typical `get` call on an inverted index.

3.1 Goals

The two main goals of Lucy.js we aimed to achieve during implementation were: (i) to build a search engine extension to IndexedDB that is easy to use and similar to the existing IndexedDB API, and (ii) to create different types of indexes that can all be used for full-text searching and find out which are more suited to which tasks and what the trade-offs are. We design Lucy.js so that it is easy as a developer to be able to specify what type of index they wish to make on a field.

3.2 IndexedDB

Indexed Database (IndexedDB) is a collection of Javascript APIs for a multi-threaded, asynchronous client-side key-value store. It is an official W3C specification [?], at the Proposed Recommendation stage as of November 20th, 2014. It is supported by almost every modern browser, both on desktop and mobile devices. Chrome, Firefox, Internet Explorer, Opera, and Safari are only some of the browsers that

[1]http://www.lunrjs.com
[2]https://github.com/yathit/ydn-db-fulltext

have added support for IndexedDB in their modern versions. For our implementation, we focused our testing and demos on the Chrome browser as it is currently the most widely used.

On Chrome browsers, IndexedDB interfaces with a key-value store named LevelDB built at Google in C++ and inspired by BigTable. In LevelDB, keys and values are arbitrary byte arrays and data is stored in sorted order by key. It supports batch writing, secondary indexes, `put` and `get` commands using the key, and forward and backward iteration over keys (range queries).

The IndexedDB API supports transactions that rely on shared read and exclusive write locking. At transaction creation time, the code must specify what kind of transaction it is (read-only, write-only, or read-write) and what object stores (similar to tables in a relational data store) it will access. Read-only transactions can run concurrently but transactions that write will lock the specified object stores for the entire duration of the transaction. All transactions in IndexedDB are asynchronous, meaning that one must request for a database operation, such as a `put` or a `get`, to happen and then pass a callback function. Then a notification via a DOM event is triggered when the operation actually completes, and the type of event returned tells whether the operation succeeded or failed.

3.3 Extending IndexedDB with Lucy.js

3.3.1 Creating a Full-Text Index

The interface to a database is maintained through an `IDBDatabase` object. To alter the schema of an IndexedDB database, for instance to create a new object store, the code must update the database version from its current version. This is not necessary when creating a normal secondary index on an existing object store. However, it will need to be updated when creating one of our new full-text indexes, as they are stored in IndexedDB as regular object stores. Thus the developer needs to persist locally the current version of the database and increment it before creating a new full-text index. Otherwise, the creation of a full-text index is very similar to that of a normal index. Given the `IDBObjectStore` that the index will be created on, we repurpose the existing `createIndex`(*index_name*, *key_path*) function for creating full-text indexes. We add an additional optional parameter of `type` which allows the developer to specify a type of full-text index they would like to build. The full-text index options we allow are `inverted`, `prefix`, and `suffix`. If the developer does not specify a type, then the function call is deferred to the native IndexedDB API and a regular index would be created. Below is an example of JavaScript code to create an inverted index called `tweets_text` on a field `text` in an object store of Twitter tweets.

```
var newDBVersion = incrementDBVersion();
var DBRequest = indexedDB.open(databaseName,
                               newDBVersion);
DBRequest.onupgradeneeded = function(evt) {
     var tn = evt.target.transaction;
     var objStore = tn.objectStore("tweets");
     objStore.createIndex("tweets_text", "text",
                          {"type": "inverted"});
};
DBRequest.onsuccess = function(evt) {
     evt.target.result.close();
};
```

3.3.2 Searching over a Full-Text Index

There are two ways to retrieve items given an `IDBIndex` object using the IndexedDB API. The first is to use a `get` function, which returns an `IDBRequest` object with the `result` field populated with the value in the object store that corresponds to the given index key. The second is to call `openCursor`, which returns an `IDBCursor` object that can be iterated over. One can pass in a range of keys instead of just one key to either function, using the `IDBKeyRange` interface.

In our case, a given search query may return many results that have different levels of relevance to the query. For this reason, each object returned has an additional field called `score`, with higher being more relevant. In the next section, we go into more detail into how the score is calculated for different types of search queries. By default we return all relevant (`score` > 0) objects in descending`score` order. However, a query can return a large number of items with low relevancy if the query contains a term that is common. We extend the `IDBKeyRange` interface to include ranges of `score` values, so that users can set a cutoff or range of the relevancy score.

In the follow code snippet, we provide an example of a `get` function on the inverted index created in the last section. The object returned by the second line is an `IDBIndexRequest` object which mimics the behavior of `IDBRequest`.

```
var index = objStore.index("tweets_text");
var request = index.get("follow friday");
request.onsuccess = function(evt) {
     console.log(request.result);
};
```

The following is an example of generating a cursor over a query on a full-text index. The cursor can be made to play forwards or backwards over the intended order using the parameter value of `next` or `prev`.

```
var index = objStore.index("tweets_text");
var request = index.openCursor("#nowplaying", "next");
request.onsuccess = function(evt) {
     var cursor = event.target.result;
     if(cursor) {
          console.log("Text: " + cursor.value.text +
                      "Score:" cursor.value.score);
          cursor.continue();
     }
};
```

Other functions of `IDBIndex` that are supported include `getKey`, which returns the primary keys on the original object store that are the result of a query, and `count`, which returns the number of relevant objects.

3.4 Natural Language Processing

Queries to our full-text search engine can be words or phrases. In addition, they can contain a special wildcard symbol % at the start or end of any word to search for parts of words. This is possible with our trie indexes, as we explain in the next section. At this time, we do not support full regular expressions nor exact phrase matches.

The processing that we conduct on queries and on the full text is standard for most search engines such as Lucene, MySQL, and PostgreSQL. The first thing we do is tokenize the text into contiguous strings of alphabetic characters, which are generally words. We convert the entire text to lower case and then remove stopwords from the bag of words. The stopword list we employ is taken from MySQL's full-text stopword list[3]. The reason we remove stopwords is because these words are present in much of natural language and including them could grow the result set to be very large but not return very relevant results.

Next, we employ stemming on each individual word using an open source JavaScript implementation of a Snowball stemmer[4]. Stemming can be a useful part of search engines because a user may be interested in text containing a word even if the word is pluralized or has some other form of suffix.

Sometimes however these processing techniques are not desired, given a particular type of text, type of query, or type of index used. For instance, stemming does not make sense when we are building a suffix trie. Our system allows the developer to pass optional parameters to disable stopword removal or stemming when desired. It is also possible for developers to replace the stopwords list for a list that is domain-specific or otherwise more preferred.

Finally, our system offers flexible support for other languages. The stemmer that we employ has support for 14 common languages in addition to English. When creating an index, a developer can specify as an optional parameter the language the full-text strings are in. Then, when it comes time, the stemmer will choose the given language to use for stemming. Also, the stopword list is included as a separate JavaScript file. To swap out one language's stopwords with another's, one only needs to replace the JavaScript file imported in the HTML page. This prevents web pages from having to load stopword lists of all supported languages when using Lucy.js.

3.5 Scoring and Ranking Results

There are many ways to score and rank results of a search query. Some techniques, such as *tf-idf*, require continual recalculation of document metadata. Others require knowing positional information about a word within a document. We define several built-in ranking functions similar to PostgreSQL's built-in `ts_rank` and `ts_rank_cd` functions that take in a `normalization` option, with integer inputs providing different types of document normalizations[5]. PostgreSQL's functions take into account how often the query terms appear in the document as well as how close together the terms are in the document. However, it is possible for a developer to implement their own ranking function and replace ours with theirs within Lucy.js if they so wish.

One function that we provide is `Lucy.calculateWeight()`, which counts the number of words in the search query that appear in the document. The other ranking we provide is `Lucy.calculateCoverDensity()`, which implements Cover Density Ranking [?], an algorithm that takes into account positional information. The result is that documents that contain the terms in the search query closer together are ranked higher than documents that have them farther apart. For this algorithm to work, we need to store at index creation time the positions that the word appears in within the document.

Both functions take in an optional `normalization` integer where the developer can specify whether they are interested in normalizing by document length or unique number of words in the document. If normalization is desired, then at index creation time, we calculate these measures and store them with the document object. Our current implementation stores them in the original object store, but another implementation could store them in a separate object store for document metadata.

3.6 Implementation

We created Lucy.js in JavaScript, as the IndexedDB API is also a JavaScript API. The code consists of four main files - `lucy.js`, `invindex.js`, `trieindex.js`, and `trieindex-hybrid.js`. The user should also additionally add one of the `<language>_stopwords.js` files in order to pick a language to check stopwords on.

The file `lucy.js` contains a static class called `Lucy` which holds various text processing functions that the different indexes use. There is also an extension of `IDBRequest` called `IDBIndexRequest` that provides the same behavior, only for our full-text indexes. Finally, there are a set of functions that intercept functions of `IDBObjectStore`. When a user calls `create_index` or `index`, if they are creating or referring to a full-text index we direct the function to one of our index implementations. When a user calls `add`, `delete`, or `put` on an object store that has a full-text index, we additionally add the item to our indexes. In the following section, we explain in more detail our index implementations. The core code as well as examples can be found on Github[6].

4. INDEX IMPLEMENTATIONS

We looked into many existing implementations of search engines and the indexes they use to decide what indexes to build, such as inverted indexes in Lucene and GiN and GiST indexes in PostgreSQL. We initially implemented an inverted index as well as prefix and suffix tries, before also developing a hybrid trie combining elements of both inverted index and tries as a third index.:

4.1 Inverted Indexes

In an inverted or reverse index, tokens taken from the full-text field make the keys of this object store. The value given a token key is a set of document ids that correspond to the documents that contain this token in that field. A example of what a inverted index could look like is given in Figure 2. Our implementation of an inverted index is relatively straightforward, as the structure is suited to a key-value store. When the user wishes to use Cover Density Ranking to compute relevancy scores, we also store positions of tokens along with the document ids. We comma delinate positions, keep them in sorted order, and separate them from the document id with a colon. For example, given the data in Figure 2, we could have `"tweet":{docIds:[3:0,7,10]}`.

When creating an inverted index, we first create a new object store that will hold the index. Then we use an `IDBCursor` over the original object store to iterate through

[3]http://dev.mysql.com/doc/refman/5.1/en/fulltext-stopwords.html

[4]https://github.com/fortnightlabs/snowball-js

[5]http://www.postgresql.org/docs/8.3/static/textsearch-controls.html

[6]https://github.com/amyxzhang/lucy.js

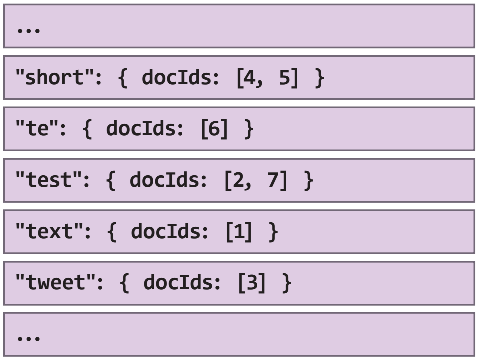


Figure 2: Structure of a simplified inverted index, as it would be stored in IndexedDB. Note that our implementation can also include additional information about the position of the tokens in the document (among other things), for better ranking

all documents in succession. After tokenizing, removing stopwords, and stemming the tokens, we then iterate through all the tokens, inserting them into the inverted index asynchronously. Because we do need to often get and rewrite lists of documents for each new occurrence, a potential better version of this method would be to construct partial inverted indexes in memory and batch write them to the database. Additionally, some tokens may have a very large number of documents, making getting and rewriting even more wasteful, though a stricter stopword list can help to mitigate this problem somewhat.

When a get call is made to an inverted index, we use the same natural language processing we used for the index creation on the search query. Then for each token in the query, we make an asychronous call to the index to search for that token and return the document ids that contain that token. After resolving duplicates and counting frequencies for each document, we go back to the original object store to retrieve the original documents given the document ids. IndexedDB does not provide batch `get` calls, so we grab the documents one-by-one asynchronously. Once all of them are retrieved, we use the designated ranking algorithm to calculate `score` values and sort the documents before returning them to the user via a callback function.

4.2 Prefix and Suffix Tries

Trie indexes are used to perform prefix and suffix searches on text data. While prefix/suffix search is often used for pattern-matching that matches the entire original text, the trie index in Lucy.js uses tokenization to perform finer-grained searches on multi-word text inputs (like tweet data). Phrase search with tries is also supported. This allows users to search by multiple prefixes or suffixes at a time, and get results with the most matches.

Prefix and suffix tries are implemented using the same design. The only difference between them is that words are reversed before insertion into suffix tries, and similarly, search terms are reversed before they are looked up in the suffix trie. Otherwise, they rely on the same implementation as described below.

A trie index in Lucy.js (prefix or suffix trie) is represented by two underlying entities in IndexedDB: an object store to save nodes in the trie, and an index on that object store to make trie traversal faster. A reference to the input object store to be indexed, which contains the full-text to be searched, is also saved.

4.2.1 *Trie Node Object Store*

Each entry in the object store represents a different node in the trie. The structure of entries in the object store is as follows:

```
node_id:{parent_id:a, char:b, docIds:[x,y,z]}
```

The key to the object store is the `node_id`, and the corresponding value object has three attributes: `parent_id`, `char`, and `docIds`. The `parent_id` attribute is the node id of the parent node in the trie, and the `char` attribute is the character value for the trie node. The final attribute is `doc_ids`, which is a list of keys into the input text object store. The keys in this list are like foreign keys that allow the trie index to retrieve the actual text from the input object store when answering queries. Each path, and therefore each node, in the trie represents a unique prefix/suffix. The `docIds` for a node contains the ids of all documents with that given prefix/suffix. Inserting a word from a given document therefore involves appending that document's id into the `docIds` of each node along the path corresponding to the word. The structure of the trie is shown in Figure 3. It is important to note that this approach is not very scalable; since each node stores a list of matching document ids, interior nodes closer to the root of the trie could store large lists of these ids. However, the benefit is that searches in the trie do not require traversal all the way down to the leaves. Matches can be returned as soon as the prefix is traced in the trie, and no further traversal along subtrees is needed. This is useful for when the dataset in question is reasonably small and speed is of great importance. An alternative design to the trie index is the Hybrid Trie discussed in the next section.

4.2.2 *Trie Node Index*

While the structure of the trie node object store optimizes lookups by node id, this is not particularly useful for trie traversal. When traversing down a trie, we would instead like to look up nodes by both their character value and their parent id. It would have been possible to change the design of the trie object store so that each node stored a list of children node ids instead of a single parent id. However, this approach requires each node to store more information, since a list of children ids is saved per node instead of a single parent id. Additionally, we would then need to iterate over this list of children ids in order to find a particular child node with a given character value. With the current object store structure, we can optimize lookups by `parent_id` and `char` by building an index on both these attributes. Overall, the trie index is backed by an object store for node storage and an index on that object store with a compound key of both `parent_id` and `char`.

4.2.3 *Lookups*

Lookups in the trie index are performed by traversing the trie downward, character by character in the search pattern. Partial matches are not accepted; the entire prefix (or suffix) must be present in the trie or no results are returned. If the entire pattern is found, the `docIds` are extracted from the trie node corresponding to the last letter of the search

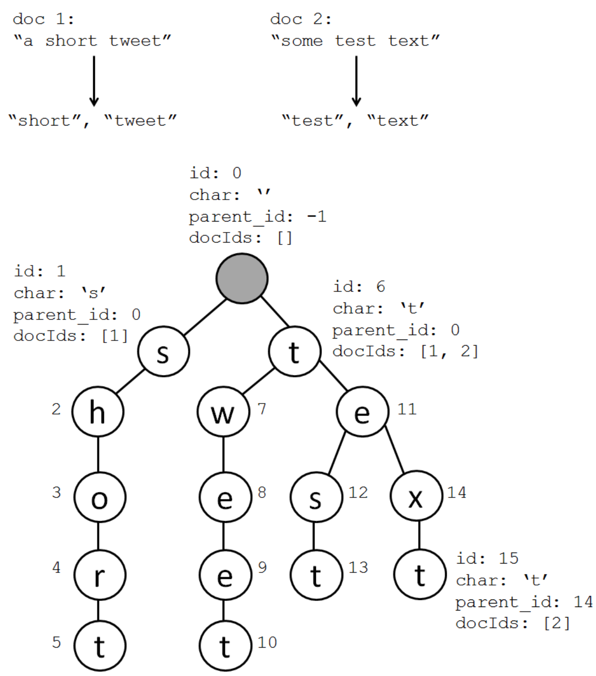


Figure 3: Structure of a trie index with two input tweets. The tweets are first tokenized and stop words are removed. Note that document ids are stored in every node. An artificial root node is created as the root of the trie.

pattern. These ids are used to key into the input text object store and retrieve the original text data.

4.2.4 Tokenization

Unlike with the inverted index, words are not stemmed before insertion or lookups. This is especially important for suffix tries because many suffix searches will not succeed due to stemming of the original text being searched. Tokenization is still used to break up input into words; it is used during both insertion (so that prefix/suffix searches can be done by word), and during lookups (making phrase search possible). Phrase search in tries involves performing prefix/suffix search on each word in the input search phrase, and ranking results by the number of matching patterns in the original text.

4.2.5 Synchronization

Due to the manner in which letters in a word need to be inserted sequentially in a trie, much of IndexedDB's built-in asynchronous behavior cannot be used to improve performance. In fact, the parallelization that IndexedDB introduces can cause serious problems due to collisions between different threads. When testing on tweet data, even inserting two tweets concurrently posed problems because the tweets contained words that started with the same letter. As a result, concurrently-running threads would try to insert duplicate elements, resulting in IndexedDB transaction errors. To avoid this, text is added sequentially to the trie index during creation, and insertion of words occurs sequentially by letter. While this avoids transaction errors, it greatly increases index creation time.

4.2.6 Depth-Limiting

To limit the size of trie indexes, an optional maximum depth can be set. If set, words longer than the depth are cut off so that the remainder substrings are stored at the leaves. Figure 4 shows the structure of the same trie as before, but with a maximum depth of 4. Leaf nodes store this substring data as a mapping from substring to document ids containing that substring. During lookups, if the maximum depth of the trie is reached, this map is searched for matches. So while depth-limiting trie indexes can limit the number of trie nodes that need to be traversed upon lookups, they incur the cost of searching these maps at the leaves. If the max depth parameter is too small, these maps can be large and slow down search significantly. If set too large, the max depth parameter has no effect.

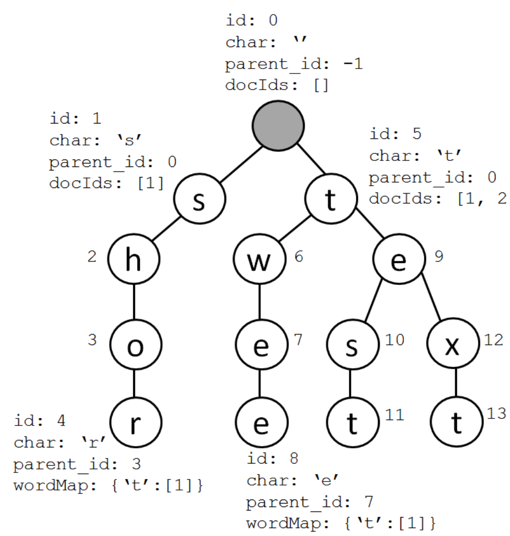


Figure 4: Structure of a trie index with a maximum depth of 4. If a word gets cut off by the depth limit, the substring remainder is stored at the leaves. Each substring remainder maps to a list of document ids with that remainder.

4.3 Hybrid Tries

We also implemented a novel approach, which we call a hybrid trie: A hybrid between an inverted index and a prefix (or suffix) trie. This approach takes advantage of the fact that every object store in IndexedDB is essentially a hash index with very fast key lookups.

Every node in our hybrid trie has its entire path as its key. Therefore, when faced with a prefix query, we can directly jump to the position in the trie that corresponds to that prefix instead of doing $O(n)$ lookups on the size of the prefix.

Then, every node includes an array of characters that are the children of the node in the trie. The key of each child is determined by concatenating the key of the parent node with each of the strings in the children array.

Every node can also potentially include a list of document ids, for documents that contain its prefix as a token (i.e. not a prefix of some longer token). Nodes without children must contain a list of document ids, but internal nodes can also contain document ids. For example, the node for cat would

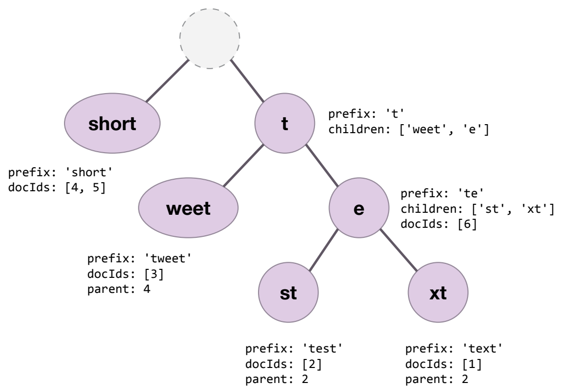


Figure 5: Structure of a hybrid trie with leaf and path shrink.

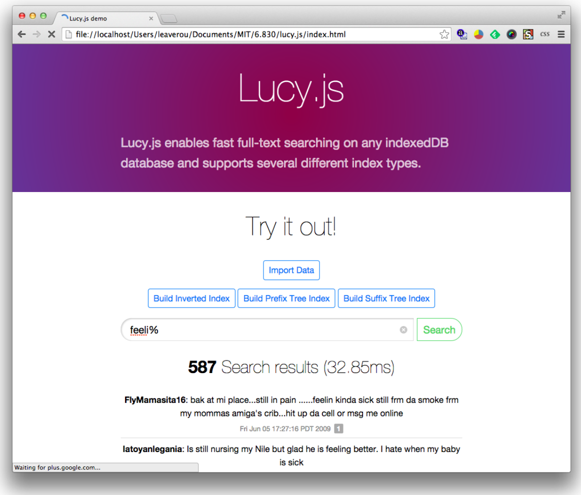


Figure 6: Our local full-text search demo that we used to take the measurements described in this experiment.

contain document ids of documents with the word "cat" itself, as well as children, for terms like "catsitting", "caturday" or "catnip". For these examples, the list of children could be `['n', 's', 'u']`.

We can see that if we discard children and child-less nodes, our index basically becomes a simple inverted index with no stemming. However, traversing the subtree for prefix searches also makes it function like a prefix trie, hence the name Hybrid (prefix/suffix) trie.

4.3.1 *Leaf and path Shrink*

To eliminate wasteful lookups, we implement leaf and path shrinkage. In tries, there are often long sequences of nodes with only one child. If those nodes have no document ids, we can combine them to a single node without any loss of functionality.

Shrinkage is a post-processing step after index generation, since we cannot know which nodes can be combined until all documents have been added to the index. Then our shrinkage algorithm works as follows:

```
for each node:
   if node.children = 1:
      for each parent's children:
         if children[i] points to node:
          children[i] += child.prefix
          exit loop
     child.parent++ (# of chars to reach parent)
     delete node
```

This results in a small speed increase, but since lookups by key in indexedDB are already quite fast, the increase is not significant. However, it also results in space savings, which is important because space on the client-side is very limited: after only a few Megabytes, browsers will ask for confirmation by the user, which is disruptive and can lead to non-technical users declining merely out of confusion.

4.3.2 *Asynchronous depth-first search*

Since IndexedDB is multi-threaded, every step of a single transaction is asynchronous. This brings several challenges to writing transactions that are not present in most other DBMS', where concurrency conundrums only arise in the context of multiple concurrent transactions.

Even something as simple as a depth-first search on our trie subtree becomes nontrivial when asynchronicity is added in the mix. Individual statements (reads or writes) are atomic, but reads and writes can be interleaved in any possible way (unless special care is taken to enforce a specific interleaving, at the cost of parallelism).

For the DFS case, we maintain a counter of pending paths that we pass by reference to the recursive method responsible for traversing the tree, decrement it by 1 when we reach a leaf and increment it by max(0, n) when we encounter a node with n+1 children. When the counter becomes zero at a leaf, we conclude the search.

5. PERFORMANCE AND COMPARISON

5.1 Data set

For the performance comparisons in this section we used a CSV dataset of a one million tweet sample originally intended for sentiment analysis[7].

We first stripped off unneeded information such as sentiment flags to reduce the size of the dataset, as we are only interested in textual content. We then wrote an ad-hoc Python script to select a random subset and convert it to JSON, which is what the generic IndexedDB importer we wrote expected.

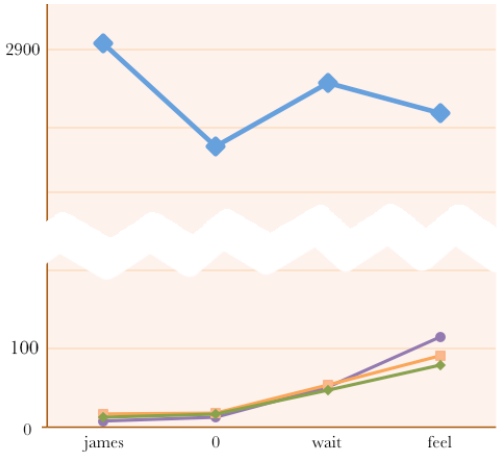
After several tests with different sample sizes, we settled for about 5% of the original dataset (44771 tweets), as our target browser (Google Chrome) was having difficulty with larger datasets.

5.2 Comparing Different Index Types

5.2.1 *Index Creation*

For our 44771 tweets dataset, it took (average of 3 index creations) 3 minutes and 21 seconds to build the inverted index, 47 minutes and 24 seconds to build the prefix trie,

[7]http://help.sentiment140.com/for-students



	james	0	wait	feel
Seq. scan	2908.57	2776.22	2857.82	2818.78
Inv. Index	7.86	12.93	50.89	115.11
Trie	16.89	18.34	53.48	91.45
Hybrid trie	12.71	16.26	47.19	78.85

Figure 7: Comparison of query times for various queries with different result set sizes (left to right: smaller to larger) between our three main types of indexes and sequential search (the only way to search strings that indexedDB offers).

48 minutes 11 seconds to build the suffix trie, 2 minutes 53 seconds to build a hybrid prefix trie and 3 minutes 2 seconds to build a hybrid suffix tree.

For a dataset of 5000 tweets, it takes 20 seconds to build the inverted index and the hybrid indices, but 3 minutes to build the prefix/suffix tries, which indicates that the prefix/suffix tries would be more suited to smaller datasets.

5.2.2 *Index Lookups*

We tried four different queries with different result set sizes (from 25 to 2398) on index type, as well as plain sequential search, which is the only type of string search that IndexedDB provides out of the box. The results can be seen in Figure 7.

While all of our indexes have comparable performance, they outperform sequential search by orders of magnitude. Sequential search for all four of our queries seems to be taking close to 3 seconds, whereas our index-optimized queries take only about 10-100ms.

5.3 Depth-Limiting for Tries

As seen in Figure 8, the maximum depth parameter has an effect on lookup times in trie indexes. The graph shows the results for a trie index prefix search, but similar results would be expected on suffix searches. A small set of input tweet data was used (367 KB data, about 2100 tweets), and the same three words were looked up in indexes created with differing maximum depths. The lookup time tends to decrease as the maximum depth increases; this is because depth cutoffs can result in slow searches through word mappings stored on leaf nodes (see Section 4.2.6 for details). The difference in times is more pronounced for the term “feel”, with a drop from 26.78ms to 18.05ms. For the term “james”, lookup times are much closer together and not as affected by depth limits. This is due to the fact that “feel” yielded many more results (85 results) than “james” (4 results), suggesting that more potential matches had to be searched at the leaf nodes for “feel” than for “james”. Results for “wait” are between the others, and also show a moderate drop in lookup time as the maximum depth increases. This is most likely because “wait” produced a moderate 36 results, and therefore a moderate number of terms were searched at the leaves.

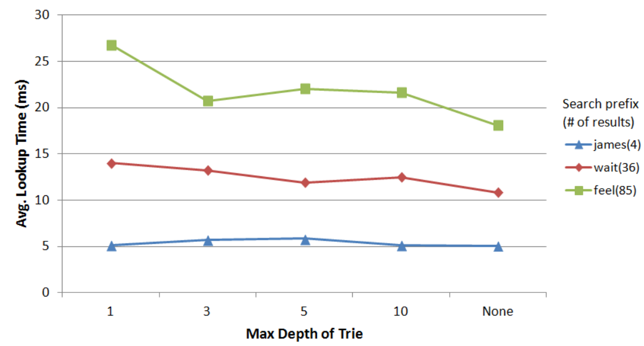


Figure 8: Maximum depth vs lookup times for a prefix trie index. A depth of “None” indicates that there is no limit. The results for three different search terms are shown. The dataset size was roughly 2100 tweets.

The data also shows that the number of results has an impact on lookup time. The three graphs are tiered such that ”feel”, with the most results, is also the slowest query. While the lookup logic is parallelized so that the text for multiple matching document ids can be retrieved concurrently, the number of lookups that are done to retrieve this text has an impact on performance.

5.4 Comparison

As we have seen earlier in this section, prefix and suffix tries are very intensive to create in a key-value store, and thus, are better suited to smaller datasets. This is what initially motivated us to build the hybrid approach, which, as we have demonstrated, is orders of magnitude faster in index creation time and slightly faster in query time. However, the hybrid approach is less space efficient than tries (in the general case), since the entire path to the node needs to be stored as its key, which makes it better suited for shorter tokens. The inverted index is slightly (within the margin of error) slower for lookups than both tries, but this is likely due to the overhead of stemming.

6. EVALUATION

As seen previously, our implemented indexes perform much faster than a naive sequential scan over all full-text values. Given the current state of the IndexedDB API, this constitutes a great improvement in new functionality for fully client-side applications. However, most web applications are not client-side only and have a server component that can then query a server-side search engine such as Lucene, MySQL, or PostgreSQL. Thus in our evaluation, we seek to determine whether the client-side approach is faster than the server-side approach. While Lucene may generally be faster than Lucy.js for query evaluation, which is likely given that it has special purpose data structures and many years of development, a fully client-side approach may still beat a server-side approach due to network latency.

The server we built is implemented in Python and uses Django, Apache, and mod_wsgi. It is hooked up to a MySQL database which has a full-text index on the tweet text. Both are hosted on the OpenStack cloud in a virtual machine, with the database residing in the virtual machine's local storage. The full-text search bar on the client side uses our inverted index to search for the query, while the typeahead search bar uses our prefix trie as it is more suited to finding prefixes.

We conduct our evaluation on two different real-life applications: a full-text search bar and a typeahead search bar that makes queries for every character as they are typed. The dataset that we test on is the same Twitter dataset used previously. We test each application on two different size datasets of 100 and 2000 tweets picked randomly. For the search queries, because we do not have access to typical search queries in real life, we simply take a random sample of 20 non-stopword words from the full tweet dataset for each set of experiments. We conduct 8 trials for each condition and delete and reinsert and recreate indexes before each trial. For the full-text search bar, we simply search for the 20 words and record the time taken. For the typeahead bar, we search for each of the 20 words as well as all prefixes of each word, in order as if one were typing into the search bar, and record the time.

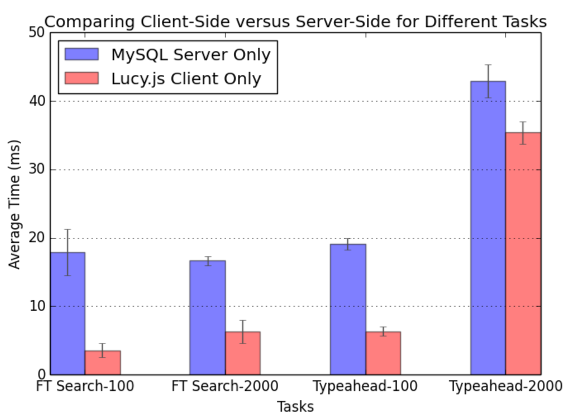


Figure 9: Comparison of client-side versus server-side indexes for full-text search. Tested on a full-text search bar (FT Search) and a typeahead search bar (Typeahead) and with Twitter dataset sizes of 100 and 2000 tweets.

6.1 Results

In Figure 9, we report the average time taken for each of the approaches (client versus server) on the two applications and dataset sizes. As can be seen, overall the client-side only approach is faster than than the server-side approach. When looking at full-text search, even when the dataset size grows over a magnitude larger, the server-side average time varies little. This suggests that the time taken is mostly taken up by network latency. We do see a small increase in the time taken for the client-side database. This suggests that as datasets grow larger, it may be a better decision to store the majority of the data on the server side and do queries on smaller subsets in the client.

When looking at the typeahead search bar, we see that when the dataset gets larger, the average time taken for both client and server increases, though the client-side increase is greater. When looking more closely within trials at the times for all the queries made, we notice that there is a great deal of variation because a very short prefix will return a much larger result set than a longer prefix. The client side had greater variation overall, with a standard deviation of 56.477ms, maximum time of 256ms, and minimum time of 1ms. The server side had a standard deviation of 49.116ms, maximum time of 220ms, and minimum time of 14ms. Given these findings, a best approach may be employ a mixed approach and send shorter prefixes to the server and longer prefixes to the client. Generally, queries that will return a small result set would be faster if they are served from the client, as the network latency always tacks on extra time for the query to the server. Queries that will return a very large result set may have better performance if they are sent to the server.

6.2 Live Demos

Demonstrations of the two applications are accessible at `http://128.52.184.3/`. Figure 10 has a screenshot of the typeahead search bar demo. For both demos, the user first deletes all data on the server as well as on the client. Then they choose whether to import the data, existing as a JSON file on the server, into the server-side MySQL database or into the client-side IndexedDB database. At that point the typeahead bar will work, querying the respective database, returning results, and outputting the time taken for each query. The user can additionally do a time test, which runs a selection of queries against the respective database and outputs the average time. Since IndexedDB does not yet support a synchronous API, the time test for the client-side database requires the user to continue clicking the button to run more queries and see an updating average time.

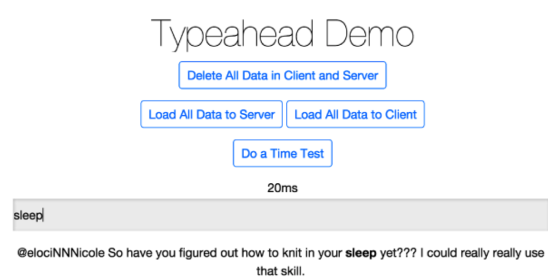


Figure 10: Screenshot of typeahead search bar demo.

7. DISCUSSION

From our findings, we determine that client-side search engines are indeed feasible to build, can be easy to use, and have improved performance over server-side search engines for certain applications and data sizes. Thus, our system Lucy.js opens up many possibilities for new applications and new data processing and storage architectures.

First, as stated earlier, there are many reasons why we would want to build applications that store and process more data on the client side. One issue that has had a great deal of attention lately is privacy and the question of the ownership of data. As a pushback against companies and web applications collecting personal and sensitive data to sell or give to others, many people have worked on applications that are more distributed than centralized, such as the Diaspora social network. Newer peer-to-peer standards such as WebRTC for browsers may make more client-heavy social and interactive applications feasible in the future. Additionally, another benefit of having data stored in the client specifically for full-text searching is that the queries themselves need not be sent to a server and logged. As shown previously, search queries can reveal a great deal about a person [**?**, **?**] and potentially some users would be interested in having private search.

We have also postulated from our findings that a mixed approach combining server and client search queries may be beneficial for performance reasons. For instance, as we saw with the typeahead search bar, longer prefixes are much faster served from the client. Another interesting application that may call for a mixed approach combining server and client queries is personalization. As we found in the previous section, queries that return a large result set might be better sent to the server. This could be used for personalization by having a server return generic results for a query and letting the client side personalize the results through re-ranking and other means. This is additionally useful because this offloads client-specific processing and client-specific data to the client and frees up resources on the server.

Finally, building out client-side storage and processing capabilities makes applications more robust to downtimes, which may be common on smartphone browsers. In addition to caching web pages locally for re-finding, a web application can now also provide searching over these web pages offline.

8. FUTURE WORK

The functionality of Lucy.js can be expanded to perform more complex queries and to improve the quality of returned results. While prefix and suffix searches are currently handled, arbitrary substring or regex searches are not supported. With these new features, users could write more customized queries and get better results. Additional forms of natural language processing could also improve the quality of results returned on searches. For example, synonym replacement could be implemented similarly to Postgres , allowing Lucy.js to find text results with similar themes or meaning. To improve performance, the existing indexes – particularly the trie index – could be optimized during index creation.

Another avenue for future research would be to build and evaluate applications that do collaborative searches between a client and server. A client could store partial data and use Lucy.js to search locally, while the server could respond with additional results not stored on the client. This could be useful to perform client-side caching of data for a web application, so that communication with the server is not always required.

9. CONCLUSION

Client-side search engines offer new functionality to client-side applications and potential speed improvements for applications that query a server-side search engine. We implemented a client-side search engine as an extension of the IndexedDB API, building several different types of index structures including inverted indexes, prefix and suffix tries, and a hybrid approach combining the two. We compare the performance of our indexes against each other and find that our hybrid approach is overall the fastest. We additionally evaluate our system by implementing a full-text search bar and a typeahead search bar and compare performance of a client-side only approach with a server-side only approach using MySQL. We find that for small to moderate datasets, the network latency incurred by trips to the server make client-size querying faster. Finally, we imagine potential applications that could be enhanced using our client-side search engine.