Building a Self-Contained Search Engine in the Browser

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo
jimmylin@uwaterloo.ca

ABSTRACT

JavaScript engines inside modern web browsers are capable of running sophisticated multi-player games, rendering impressive 3D scenes, and supporting complex, interactive visualizations. Can this processing power be harnessed for information retrieval? This paper explores the feasibility of building a JavaScript search engine that runs completely self-contained on the client side within the browser—this includes building the inverted index, gathering terms statistics for scoring, and performing query evaluation. The design takes advantage of the IndexDB API, which is implemented by the LevelDB key–value store inside Google's Chrome browser. Experiments show that although the performance of the JavaScript prototype falls far short of the open-source Lucene search engine, it is sufficiently responsive for interactive applications. This feasibility demonstration opens the door to interesting applications and architectures.

Categories and Subject Descriptors: H.3.4 [Information Storage and Retrieval]: Systems and Software

Keywords: JavaScript; LevelDB; IndexedDB

1. INTRODUCTION

In nearly all deployments, search engines handle the vast bulk of processing (e.g., document analysis, indexing, query evaluation) on the server; the client is mostly relegated to results rendering. This approach vastly under-utilizes the processing capabilities of clients: web browsers today embed powerful JavaScript engines capable of running real-time collaborative tools [5], powering online multi-player games [3], rendering impressive 3D scenes, supporting complex, interactive visualizations,[1] enabling offline applications [6], and even running first-person shooters.[2] These applications take advantage of HTML5 standards such as WebGL, WebSocket, and IndexedDB, and therefore do not require additional plug-ins (unlike with Flash).

[1] `d3js.org`

[2] `www.quakejs.com`

ICTIR'15, September 27–30, Northampton, MA, USA.

ACM 978-1-4503-3833-2/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2808194.2809478.

Can we apply this processing power for information retrieval in interesting new ways? This paper explores the feasibility of building a JavaScript search engine that runs completely self-contained in the browser—this includes parsing documents, building the inverted index, gathering terms statistics for scoring, and performing query evaluation.

Is such a design merely a curiosity, or does it offer advantages over traditional client–server architectures? Even as a curiosity, this work explores how far browser technologies have advanced in the previous decade or so, where they have emerged as a viable platform for delivering rich user experiences. However, browser-based search engines provide interesting opportunities for information retrieval, both from the perspective of enabling novel applications and opening up the design space of search architectures.

In addition to discussing the implications of a browser-based search engine, this paper—which is a shorter version of a previous report [7]—describes the design and implementation of JScene (pronounced "jay-seen", rhymes with Lucene), an open-source proof of concept that illustrates the feasibility of these ideas. JScene takes advantage of the IndexedDB API, which is supported by a few modern web browsers and implemented using LevelDB in Google's Chrome browser. The result is a completely self-contained search engine that executes entirely on the client side without any external dependencies. As a reference, JScene is compared against the popular open-source Java search engine Lucene; it should not be a surprise that JScene falls short of Lucene in performance, but results nevertheless demonstrate that a pure JavaScript implementation is sufficiently responsive to support interactive search capabilities.

2. DESIGN IMPLICATIONS

Suppose it were possible to build an in-browser JavaScript search engine that is fully self-contained and delivers reasonable performance: so what? More than a technical curiosity, such a design promises to open up many interesting possibilities, detailed below (see more discussion in [7]).

Offline access. One obvious advantage of this design is that the user doesn't need to be connected to the internet, so documents are available for searching offline. One can imagine a background process that continuously ingests web pages that the user has visited in the recent past and updates an index of these documents—search capabilities would then be available even if the computer were disconnected from the network. Previous studies have shown that a significant fraction of users' search behavior on the web consists of "refinding" [9], or searching for pages they had

encountered before. Thus, a reasonably-sized local index might achieve good coverage of many user queries.

Private search. Another advantage of a search engine that resides completely self-contained within the browser is that there is no third party logging queries, clicks, and other interactions. This is particularly useful when a user has a collection of documents she wishes to search privately—for example, when researching a medical condition, some stigmatized activity, or other sensitive topics. This scenario would be operationalized by coupling the in-browser search engine with a focused crawler: the user would, for example, direct the crawler at a collection of interest (e.g., a website with medical information), and the search engine would then ingest documents according to crawl settings.

Load shedding. From the perspective of a commercial search engine company, which needs to continuously invest billions in building datacenters, in-browser search capabilities are appealing from the perspective of reducing server load. However, "dispatching" queries for local execution on the client's machine may eliminate the opportunity to generate revenue (i.e., via ad targeting), but this is an optimization problem that search engine companies can solve. Although the coverage of a local index would be miniscule compared to the centralized index of a commercial search engine, for particular classes of queries (such as the "refinding" queries discussed above), the local collection might be adequate (and it is possible that refinding queries provide fewer ad targeting opportunities anyway).

Split execution. Instead of purely server-side or client-slide query execution, there are possibilities for split execution where query evaluation is performed cooperatively. One possibility is search personalization, where generic search results are tailored to a user's interests on the client side.

Distributed search marketplace. Synthesizing the ideas discussed above, one possible future scenario is the emergence of a marketplace for hybrid models of centralized and distributed search where optimization decisions are arrived at jointly by rational economic actors driven by incentives.

For example, a commercial search engine company might offer an incentive for a user to execute all or part of a search locally, in the form of a micropayment or the promise of privacy (e.g., not storing the queries and interactions). From the search engine company perspective, the value of the incentive can be computed from the costs of datacenters, revenue opportunities from ad targeting, etc. Commercial search engine companies have a clear sense of which searches are "money-makers" (rich ad targeting opportunities) and which aren't. Yet, all searches currently cost the companies money. From the users' perspective, they can control in a fine-grained manner their preferences for privacy and resource usage. The marketplace determines when the incentives on both ends align. To the extent that "money-losing" queries overlap with the types of queries that can be handled locally, such transactions are mutually beneficial.

3. FEASIBILITY STUDY

Having discussed the interesting applications and architectural possibilities of self-contained, in-browser search engines, it is now time to address the practical question: Is such a design actually feasible? Note that the goal of JScene, the proof of concept described in this paper, is to show that it is possible to build a pure JavaScript in-browser search engine with *reasonable* performance—within users' latency tolerance for interactive search. Of course, the performance of the system will not come close to a custom-built search engine (e.g., Lucene), but that's not the point; a feasibility demonstration confirms that this general concept warrants further exploration. To facilitate follow-on work, the JScene prototype is released under an open-source license and available to anyone interested.[3]

At the storage layer, JScene depends on LevelDB, an on-disk key–value store built on the same basic design as the Bigtable tablet stack [2]. It is implemented in C++ and was open-sourced by Google in 2011. The key–value store provides the Chrome implementation of the Indexed Database (IndexedDB) API, which is formally a W3C Recommendation.[4] LevelDB supports basic put, get, and delete operations on collections called "stores". Keys are maintained in sorted order, and the API supports forward and backward iteration over keys (i.e., to support range queries). Data are automatically compressed using the Snappy compression library, which is optimized for speed as opposed to maximum compression. The upshot is that inside every Chrome browser, there is a modern key–value store accessible via JavaScript. The JScene prototype takes advantage of LevelDB, as exposed via the IndexedDB API, to store all index structures.

3.1 Index Construction

The biggest challenge of building JScene is implementing an inverted index using the provided APIs. The IndexedDB API is built around key–value pairs, where values can be complex JavaScript objects and keys can be JavaScript primitives, a field inside the value object, or auto generated. In JScene, the postings are held in a store called `postings`, where the key is a concatenation of the term and the docid containing the term, and the value is the term frequency. For example, if the term "hadoop" were found twice in document 2842, the key would be "hadoop+2842" (with "+" as the delimiter) with a value of 2. In the tweet search demo application (see below), tweet ids can be used directly as docids, but in the general case, docids can be sequentially assigned as documents are ingested. A postings list corresponds to a range of keys in the `postings` store, and thus query evaluation can be translated into range scans, which are supported by IndexDB.

A few alternative designs were considered, but then rejected (at least for this prototype): it seemed more natural to map each individual posting onto a key–value pair as opposed to accumulating a list as the value of a single key (the term), since in that case the indexer would need to rewrite the value every time a term was encountered. Of course, it is possible to batch data and perform term–document inversion in memory, but this adds complexity that is perhaps not necessary for a proof of concept. In many retrieval engines, terms are mapped to unique integer ids, which allows the postings to be more compactly encoded. Since IndexDB keys are strings, this doesn't seem like much help.

Given this design, the indexer operation is straightforward. Each input document is represented as a JSON object and the entire collection is stored in an array. The indexer processes each document in turn and generates key–value pair insertions corresponding to the inverted index design

[3] `jscene.io`

[4] `www.w3.org/TR/IndexedDB/`

described above. All transactions in IndexDB are asynchronous, where the caller supplies an `onsuccess` callback function which is executed once the transaction completes. Thus, a naïve indexer implementation based on a for loop that iterates over the documents would simply queue potentially millions of transactions, completely overwhelming the underlying store. Instead, the indexer is implemented using a chained callback pattern—the `onsuccess` callback function of a transaction to insert a key–value pair initiates the next insertion, iterating through all tokens in a document and then proceeding to the next document, until the entire collection has been processed. This style of programming, although foreign in languages such as C/C++ or Java, is common in JavaScript.

Separately, the index also needs to store document frequencies for scoring purposes. These statistics are held in a separate store, aptly named `df`. The document frequencies are first computed by iterating over the entire collection and keeping track of term statistics in a JavaScript object (i.e., used essentially as a hash map). Once all documents have been processed, all entries in the object are inserted into the store, with the term as the key and the document frequency as the value. Once again, the chained callback pattern described above is used for these operations.

3.2 Query Evaluation

With an inverted index, query evaluation algorithms traverse postings in response to user queries to generate a top k ranking of results. Query evaluation for keyword search, of course, is a topic that has been extensively studied (see [10] for a survey). In the context of this work, the goal is to explore the feasibility of in-browser query evaluation using JavaScript, not raw performance per se. Thus, experiments in this paper used a simple approach based on *tf-idf* scoring that requires the first query term to be present in any result document. There are several reasons for this choice: First, previous work has shown that this scoring model works reasonably well in practice [1]. Second, terms in a user's query are often (implicitly) sorted by importance, and so it makes sense to treat the first query term in a distinguished manner. Third, this approach serves as a nice middle ground between pure conjunctive (AND) and pure disjunctive (OR) query evaluation. Finally, this approach lends itself to a very natural implementation in JavaScript described below.

The prototype query evaluation algorithm uses a very simple hash-based approach in which a JavaScript object is used as the accumulator to store current document scores, with the document id as the property and the score as the value (essentially, a hash map). In the initialization step, the document frequencies of all query terms are first fetched from the `df` store. Next, a range query corresponding to the first query term is executed, and all postings are scanned. The accumulator hash map is initialized with scores of all documents that contain the term. After the first query term is processed, the query evaluation algorithm proceeds to the next query term, which results in another range scan; for each posting, the accumulator structure is probed, and if the key (document) is found, the value (document score) is updated. All query terms are processed in this manner. At the end, the contents of the accumulator are sorted by value to arrive at the top k.

Two details are worth discussing. First, this query evaluation algorithm bears resemblance to the so-called SvS algorithm for postings intersection (i.e., AND-ing of all query terms) that cyclically intersects the next postings list with the current partial results [4]. However, the standard implementation takes advantage of binary search, skip lists, and other techniques—given the limitations of the LevelDB API, it is not entirely clear how such optimizations can be implemented in JavaScript. Second, the design of the IndexedDB API makes the JScene query evaluation code somewhat convoluted. A range scan begins by acquiring a cursor, which is an asynchronous operation with an associated `onsuccess` callback. An object passed into the callback provides a method that advances the cursor. Thus, the entire query evaluation algorithm is implemented (somewhat awkwardly) as chained callbacks: when the sequence of callbacks corresponding to the processing of the first query term completes, it triggers the range query for the second term and the series of callbacks associated with that, and so on. As with indexing, this style of programming is foreign to developers used to building systems in C/C++ or Java.

4. EXPERIMENTS

Experiments were conducted on a 2012-generation Macbook Pro, with a quad-core Intel Core i7 processor running at 2.7 GHz with 16 GB RAM and a 750 GB SSD. The machine ran Mac OS X 10.9.2 with Google Chrome version 33.0.1750.146. Experiments used the Tweets2011 collection from the TREC 2011 Microblog track [8], which consists of 16 million tweets. Initial trials indicated that JScene would not be able to index the entire Tweets2011 collection within a reasonable amount of time. Thus, a smaller collection comprising 1.12m tweets was created by random sampling. In total, the documents contain 13.9m tokens with 1.74m unique terms, occupying 140 MB on disk uncompressed.

For evaluation, JScene was compared to the Lucene search engine (version 4.7.0). To provide a fair comparison, the Lucene queries were formulated to specify the same constraints as in JScene. To ensure that both systems were processing the same content, for JScene the collection was first tokenized with the Lucene tools provided as a reference implementation in the TREC Microblog evaluations[5] and the resulting tokens were then re-materialized as strings to create the JSON documents used by JScene.

Evaluations used 109 queries from the Microblog tracks at TREC 2011 and 2012 (ignoring the query timestamps). The relevant metrics in these experiments are indexing and query evaluation speed. No effectiveness evaluation was conducted, which is saved for future work.

It took JScene 644 minutes (∼10.7 hours) to build the inverted index for 1.12m tweets and another 152 minutes (∼2.5 hours) to construct the document frequency table (both averaged over two trials). While building the inverted index, the Mac OS X Activity Monitor showed CPU usage oscillating roughly between 15% and 25%, where the peaks correspond to LevelDB compaction events. These utilization levels suggest that the process is IO bound (even though the machine is equipped with an SSD). The LevelDB data for the postings occupy approximately 1.6 GiB on disk, and the document frequency table another 0.2 GiB.

Indexing results translate into a sustained write throughput of around 360 postings per second. However, these figures are not directly comparable with other performance

[5] `twittertools.cc`

System	mean	median	P90	max
JScene	146	106	311	1058
Lucene	1.4	0.8	2.8	9.8

Table 1: Query evaluation performance comparing JScene and Lucene; all values in milliseconds.

evaluations of LevelDB because of at least two reasons: first, it is unclear how much overhead JavaScript and the IndexDB API introduce, and second, our chained callback implementation means that the insertions were performed sequentially (i.e., synchronously), which is known to be much slower than the standard asynchronous write mode.[6]

For reference, Lucene took 27 minutes to index the same collection on a single thread (averaged over two trials). The on-disk index size is just 154 MiB. It is quite clear that JScene indexing throughput falls far short of Lucene, and that Snappy compression is far less effective than special-purpose compression schemes designed specifically for search. This should not be surprising.

Table 1 compares query latency of JScene and Lucene for the 109 queries from TREC 2011 and 2012: figures show mean, median, 90^{th}-percentile, and max values (averaged over three trials). Results for both are with a warm cache and Lucene ran in a single thread. In terms of the mean latency, JScene is roughly two orders of magnitude slower than Lucene; the performance gap is about the same based on the other metrics. This is of course not surprising since Lucene uses specialized data structures and has received much attention from the open-source community. However, JScene is reasonably responsive, with query latencies within the range that users would expect for interactive systems.

To further explore the performance of JScene, a set of terms were randomly sampled from the `df` store and treated as single-term queries. The performance of these queries are shown as solid squares in Figure 1. The figure focuses on terms with *df* less than 1000, but the linear relationship extends to all sampled terms. The solid squares give a sense of the lower bound on query latency, since any document-at-a-time query evaluation algorithm will need to scan all postings for the single query term. For comparison, the TREC queries are plotted as circles based on the *df* of their first query term. This plot illustrates two points: First, there remains much room for improvement in JScene. Second, even in the limit, query evaluation with IndexedDB (via LevelDB), at least with the current storage layout, will still be measured in tens of milliseconds.

5. FUTURE WORK AND CONCLUSION

What can we conclude from these experiments? Results suggest that although a self-contained, in-browser JavaScript search engine is much slower than a custom native application (big surprise), the JavaScript implementation is sufficiently responsive for interactive querying. The current prototype is sufficiently performant to be deployed for searching (most) users' timelines, i.e., *all* tweets that a user has ever read. From this perspective, the design is most definitely feasible and worthy of further exploration.

These experimental results, however, reflect only a first attempt at realizing the general concept. The prototype reflects a straightforward (i.e., dumb) technical implementation, without applying any of the standard efficiency tricks

[6] `leveldb.googlecode.com/svn/trunk/doc/benchmark.html`

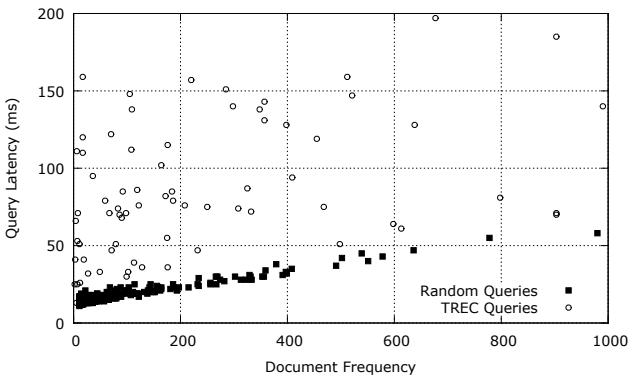

Figure 1: Latency vs. document frequency of query term for randomly-generated and TREC queries.

that are available in every researcher's toolbox. These include various types of compression, alternative schemas and storage layouts, optimizing data access patterns for better locality, etc. These techniques, coupled with future improvements in the IndexedDB implementation, will narrow the gap between in-browser and native search applications.

Given this feasibility demonstration, it would not be premature to start exploring some of the applications and architectures discussed in Section 2. Performance and scalability will continue to improve and become less and less of an issue: in the limit, local indexes have an inherent performance advantage in eliminating network latencies.

6. ACKNOWLEDGMENTS

This research was supported by NSF awards IIS-1218043 and CNS-1405688 while the author was at the University of Maryland. Any opinions, findings, conclusions, or recommendations expressed are the author's and do not necessarily reflect the views of the sponsor.

7. REFERENCES

[1] N. Asadi and J. Lin. Fast candidate generation for real-time tweet search with Bloom filter chains. *ACM TOIS*, 31, 2013.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *OSDI*, 2006.

[3] B. Chen and Z. Xu. A framework for browser-based multiplayer online games using WebGL and WebSocket. *ICMT*, 2011.

[4] J. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM TOIS*, 29(1), 2010.

[5] C. Gutwin, M. Lippold, and T. Graham. Real-time groupware in the browser: Testing the performance of web-based networking. *CSCW*, 2011.

[6] R. Leblon. Building advanced, offline web applications with HTML 5. Master's thesis, Universiteit Gent, 2010.

[7] J. Lin. On the feasibility and implications of self-contained search engines in the browser. *arXiv:1410.4500*, 2014.

[8] I. Ounis, C. Macdonald, J. Lin, and I. Soboroff. Overview of the TREC-2011 Microblog Track. *TREC*, 2011.

[9] S. K. Tyler and J. Teevan. Large scale query log analysis of re-finding. *WSDM*, 2010.

[10] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(6):1–56, 2006.