Taking Entity Reconciliation Offline

Ryan Shaw
School of Information and Library Science
University of North Carolina at Chapel Hill
ryanshaw@unc.edu

Patrick Golden
School of Information and Library Science
University of North Carolina at Chapel Hill
ptgolden@live.unc.edu

ABSTRACT

Entity reconciliation—linking names or terms to identifiers in external datasets—is a popular method of adding standardized structured data to loosely structured documents. Most approaches to entity reconciliation rely on remote web services, requiring network access during the reconciliation process. For use cases that rely on a "human in the loop" (reconciling entities during the authoring process), this requirement may be a problem. To address this problem, we investigated the feasibility of offline entity reconciliation against the Virtual International Authority File. Offline entity reconciliation was implemented by taking advantage of newly standardized browser storage interfaces to store and query parts of this large dataset locally. We present the results of this investigation and our comparison of the performance, scalability, ease of implementation, and cross-browser compatibility of the various options for storing entity data locally.

Keywords

Linked data, name authorities, web interfaces.

INTRODUCTION

We first review the typical approaches taken to annotate loosely structured text with structured data. We claim that reconciliation against a database of entities is an attractive approach for many use cases. However, most implementations of reconciliation establish a dependency on web services, making some use cases difficult to support. We examine techniques for breaking this dependency by storing and reconciling against entity data locally. We present the results of a study in which we implemented and tested offline reconciliation using several combinations of operating systems, browsers, and local storage technologies. The local storage technologies are compared to one another in terms of ease of use, scalability, and performance. We conclude with a discussion of the implications of our study.

STRUCTURED DATA

Information professionals are well-acquainted with the benefits of adding standardized structured data (e.g. metadata) to loosely structured documents. Standardized structured data can bring consistency and interoperability to otherwise inconsistent and idiosyncratic documents, making them amenable to consumption and manipulation through generic tools. Faceted browsing and visualization are just two specific examples of this.

While structured data can be authored directly using forms, another approach is attractive when authors are willing to re-use another author's description of an entity (as in shared cataloging), or when there is an external source of structured data about the entities that can be exploited. For example, the restaurants a food blogger reviews are likely to be listed in a directory providing structured data. Medical thesauri will have structured data related to the terms a doctor uses in her notes. A place name gazetteer can provide structured data related to a place name. In all these cases an author need not re-enter this data but can simply reconcile the name or term he used with the external data source. Reconciliation involves an author linking a name or term to an external identifier, thereby disambiguating it and allowing him to gather structured data that others have associated with that identifier (Maali et al., 2011).

Adding structured data to documents via reconciliation against an external data source typically introduces a dependency on web access. For use cases that cannot tolerate sparse or dirty data, and which therefore adopt a "human in the loop" model of reconciling entities during the authoring process, the need to be constantly online may be problematic. Consider the doctor making clinical observations in unconnected rural areas, or the historian taking research notes deep in an archive. Can adding structured data via reconciliation during authoring be feasible in these offline scenarios?

LOCAL STORAGE TECHNIQUES

The most basic approach to using browser storage for entity reconciliation is to serialize and store an index structure that is deserialized and loaded fully into memory upon page load. In theory, this method could be used to store a small entity index in cookies, but a better approach would be to use the Web Storage API. The Web Storage API (Hickson, 2011) better known as `localStorage`, enables persistent storage of key-value pairs. It is intended to be used to store data that should persist across browser sessions and are too large to be stored in cookies. Another option is to use the newer File API, which provides FileSaver (Uhrhane, 2012) and FileReader (Ranganathan, 2012) interfaces that can be

ASIST 2013, November 1-6, 2013, Montreal, Quebec, Canada.

Local storage technology	Browser support
Web Storage API	Supported by all major browsers
File API	File writing only supported in Chrome
Web SQL Database API	Not supported by Internet Explorer or Firefox
Indexed Database API	Not yet supported by Safari or iOS Safari; only partially supported by Internet Explorer

Table 1. Browser support for local storage.

used for writing data of arbitrary size to files on disk and reading it back into memory.

An obvious limitation of in-memory approaches to offline entity reconciliation is that the entire entity index must be loaded into memory before it can be searched. This could become a problem for very large indexes. The alternative is to use a client-side database that can be indexed and queried without loading all of the data into memory. The Web SQL Database API (WebSQL) (Hickson, 2010) provides an interface to an embedded SQLite relational database engine. Thus it is essentially a standardization of the approach already used by browser extensions such as Zotero (Cohen, 2008) that store data using Sqlite. The Indexed Database API (IndexedDB) (Mehta, 2012) provides a low-level interface to a non-relational object store and supports high-performance querying of JavaScript objects (lists of key-value pairs) via indexes. Values may themselves be objects, enabling the storage of hierarchical structures. The various browser implementations of IndexedDB are built on different embedded databases; Internet Explorer uses the Extensible Storage Engine, Firefox uses SQLite, and Chrome uses LevelDB (Powell, 2012).

EVALUATION FRAMEWORK

To evaluate the suitability of these various technologies for implementing offline entity reconciliation, we built a small testing framework.[1] To provide a realistic test of entity reconciliation, we drew upon a popular source of identifiers for persons, places, and organizations: the Virtual International Authority File (Loesch, 2011). The VIAF data is a set of "clusters" of related records from various international authority files. Each cluster represents a single entity such as a personal identity, corporate body, or geographic place. Working from a recent (February 2013) dump of the VIAF data, we produced a JSON file with an identifier, primary name, and array of alternate names for each entity. We ran our evaluations using JSON arrays of varying size consisting of the first 10K, 50K, 100K, and 990K records respectively. To implement reconciliation we tokenized all the names of each entity and added the array of tokens to each JSON object; all matching between queries and entities was done using this array of tokens, and thus all of our various implementations returned identical entity sets for the same query.

To test each reconciliation implementation we issued a series of queries simulating those that would be generated by an autocompletion interface. Each query was executed five times and the average retrieval time was recorded. Retrieval time included not just the time required to identify matching entities, but also the time required to access the matching properties (as would be needed to display feedback in a reconciliation interface).

RESULTS

Because JavaScript engines and implementations of the various APIs vary across browsers, we ran our tests on a variety of combinations of device type, operating system, and browser. Table 1 summarizes current browser support for the various storage technologies we tested.[2] In this section we present a comparison of the APIs along the axes of ease of use, scalability, and performance.

Ease of Use

`localStorage` is straightforward to use; one uses `setItem(key, value)` to store data and `getItem(key)` to retrieve it. Querying (beyond simple exact key matching) must be implemented in JavaScript, but for data small enough to be stored in `localStorage` a simple loop through the data checking for matches is very fast. The File API is similar to file I/O APIs provided by many other standard libraries, but using it is far more complex than using `localStorage`. One must be prepared to monitor progress and handle a variety of errors that might occur when reading or writing files. WebSQL should be familiar to developers who are accustomed to the relational database paradigm and know SQL; whether that familiarity breeds fondness or contempt will depend on the programmer. IndexedDB, on the other hand, has a very different kind of interface that is unfamiliar to many Web developers. This unfamiliarity has resulted in many complaints about its understandability and usability (for a sampling see Caceres, 2013). The designers of IndexedDB have countered this criticism by pointing out that IndexedDB is intended to be a powerful but low-level API and that they expect more user-friendly APIs to be layered on top of it, much as jQuery and other JavaScript libraries emerged to mediate between programmers and the low-level DOM APIs.

[1] Testing tool and full results data at `http://ptgolden.github.io/browser-storage/`, source code at `https://github.com/ptgolden/browser-storage`.

[2] See `http://caniuse.com` for more details.

Scalability

How much data can be stored locally using these methods? The Web Storage specification recommends a “mostly arbitrary limit of five megabytes” (Hickson, 2011), which the browsers we tested enforced. The other APIs do not impose any hard limits on the amount of data that can be stored, but typically the user must give permission to a website wishing to store more than a small amount of data. In our tests, storing the 10K dataset using WebSQL resulted in a 3MB file on disk in Chrome and a 5MB file on disk in Safari. The larger datasets either froze or crashed Chrome and Safari when using WebSQL. IndexedDB fared better: we were able to store the 100K dataset in all the browsers supporting it. In the non-mobile versions of Chrome we were able to store the 990K dataset using IndexedDB (resulting in a 726M file on disk). We were also able to load the 990K dataset into memory using the non-mobile browsers; although this is too much data to be stored using `localStorage` it could conceivably be serialized to and read from a file using the File API. (On the mobile browsers the 990K dataset could not be read into memory due to a lack of sufficient RAM.)

Performance

IndexedDB also outperformed the other storage methods in terms of query processing time; here we just present some general findings. Figure 1 compares the retrieval times among different mobile browser storage methods for queries of various lengths (result sets of various sizes) on the 10K dataset. All the methods performed well on this small dataset, returning results in under a second. Overall, WebSQL retrieval was the slowest (and particularly slow in Chrome on Android). In-memory data access was consistently fast. The two IndexedDB implementations were the fastest—but only for the smaller result sets. Iterating over the retrieved results using the IndexedDB cursor is slow since the result value objects are created lazily as they are accessed. This can negatively impact retrieval performance when there are lots of results. We observed a similar pattern of performance among non-mobile browsers; see Figure 2 for a comparison of browsers running on a Macbook Pro. (We did not observe significantly different performance by Chrome and Firefox on Linux or Internet Explorer on Windows.) IndexedDB's performance advantage over looping through an in-memory array was particularly clear with the 990K dataset.

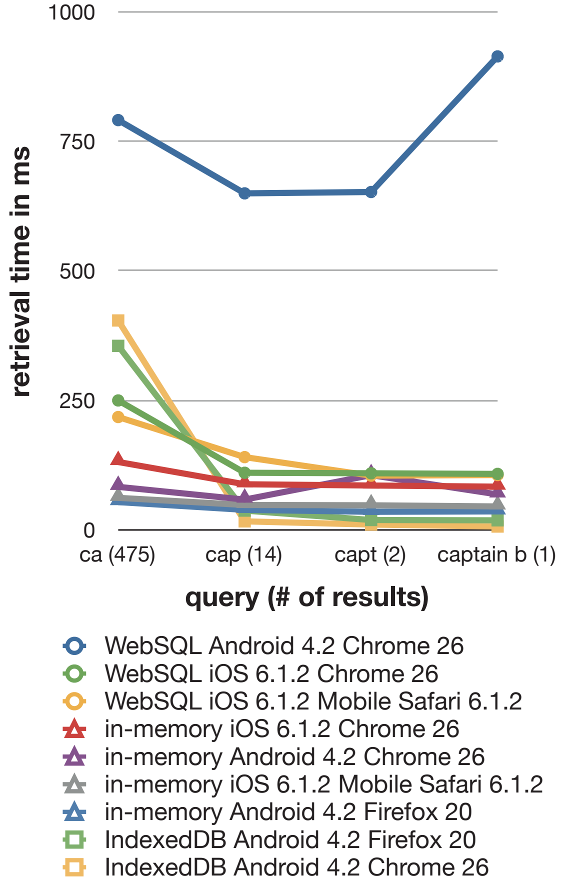

Figure 2: Retrieval times on mobile browsers.

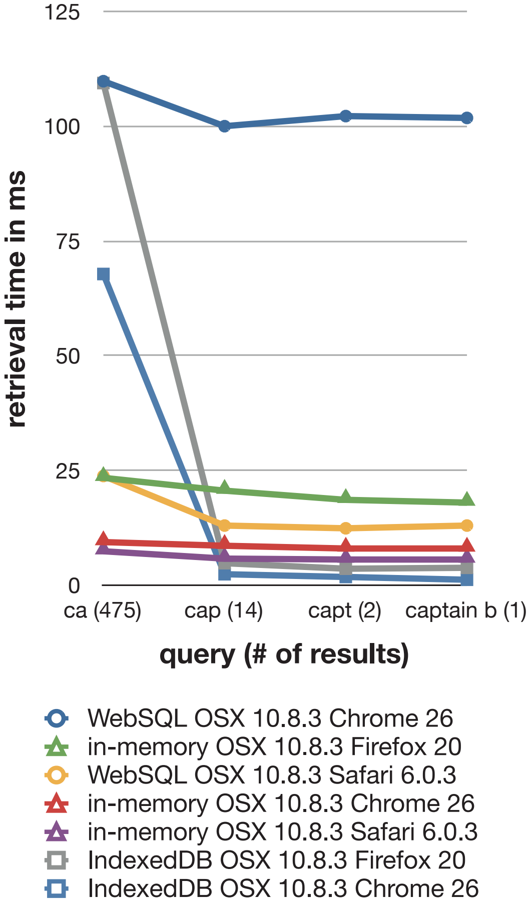

Figure 2: Retrieval times on a MacBook Pro.

DISCUSSION

Our experiments demonstrated that offline reconciliation against a non-trivial entity set is feasible. Even on a mobile device, tens of thousands of entity records can be locally stored and queried quickly enough to provide an autocompletion interface. On higher-powered devices such as laptops, this can scale up to hundreds of thousand of entity records. That is insufficient to store massive datasets, such as the full VIAF dataset which includes approximately 27 million entities. But it is not necessary or even necessarily desirable to locally cache that much data. The majority of use cases involve interaction with only a subset of entities. For example, one scholarly editing project with which we are collaborating on this research has been in operation for over three decades. During that time they have recorded approximately 7,500 unique personal names in their research database. Reconciliation against those names could easily be provided offline. As another example, the Medical Subject Headings (MeSH) thesaurus contains 26,853 descriptors and 214,000 supplementary concept records. These too could be reconciled against offline.

For relatively small datasets (less than 5MB of data), we recommend simply using `localStorage` to store a JavaScript array that can be queried in memory. We found that we could store identifiers, primary names, and alternate names of 10,000 entities this way, and that simply looping through the array provided fast querying even on mobile devices. For larger datasets, we recommend using IndexedDB. IndexedDB was the fastest of the methods we investigated, and there are no hard limits on the amount of data that can be stored. However very ambiguous reconciliation queries (returning more than a thousand matching entities) should be avoided when using IndexedDB, due to the performance implications of accessing that many records through the cursor. For example in the case of an autocompletion interface, this may mean not offering completion suggestions until the user has typed at least three characters.

The performance comparison we conducted is, of course, contingent on our implementations. It is possible that WebSQL performance could be improved through better indexing of the database and optimization of the query used. In-memory querying might outperform IndexedDB if we implemented a more efficient data structure such as a trie rather than simply looping through an array. However, it was not our intent to compare the most efficient possible implementation of each method; rather we wanted to compare the performance of the simplest and most straightforward implementations. By that measure `localStorage` for small amounts of data and IndexedDB for larger amounts are the clear winners.

`localStorage` and IndexedDB also appear to be the safest bets for cross-platform compatibility. `localStorage` is already available in all modern browsers. IndexedDB is not yet available in Safari, but the code is in the main WebKit repository meaning that it is likely to be available soon. And while the Internet Explorer implementation was not yet complete enough for us to test it, we do not expect that situation to persist given Microsoft's support of the standard. WebSQL, on the other hand, will never be a cross-platform solution, and it does not seem likely that the file-writings parts of the File API will be either.

CONCLUSIONS

Reconciliation against a database of entities is an attractive way to add structured data to text. Using local storage techniques now available in web browsers, it is feasible to store and reconcile against large collections of entity data. When reconciliation does not require an always-on network connection, it can be deployed in more scenarios, such as when authors are writing offline. There is much to be said for this decentralized approach, and we hope that the work presented here helps to articulate it as an alternative to ever-greater centralization in “the cloud.”

ACKNOWLEDGMENTS

We are grateful to the Andrew W. Mellon Foundation for funding “Editorial Practices and the Web” and to Coleman Fung for helping fund Patrick Golden.

REFERENCES

Caceres, M. (2013). IndexedDB, what were the issues? http://lists.w3.org/Archives/Public/www-tag/2013Feb/0003.html

Cohen, D. J. (2008). Creating scholarly tools and resources for the digital ecosystem: Building connections in the Zotero project. *First Monday*, 13(8).

Hickson, I. (2010). Web SQL Database. http://www.w3.org/TR/2010/NOTE-webdatabase-20101118/

Hickson, I. (2011). Web Storage. http: //www.w3.org/TR/2011/CR-webstorage-20111208/

Loesch, M. F. (2011). The Virtual International Authority File. *Technical Services Quarterly*, 28(2):255–256.

Maali, F., Cyganiak, R. & Peristeras, V. (2011). Re-using cool URIs: Entity reconciliation against LOD hubs. In C. Bizer, T. Heath, T. Berners-Lee, and M. Hausenblas, (Ed.), *WWW2011 Workshop on Linked Data on the Web*.

Mehta, N., Sicking, J., Graff, E., Popescu, A. & Orlow, J. (2012). Indexed Database API. http://www.w3.org/TR/2012/WD-IndexedDB-20120524/

Powell, A. (2012). How the browsers store IndexedDB data. http://www.aaron-powell.com/web/indexeddb-storage

Ranganathan, A. & Sicking, J. (2012). File API. http://www.w3.org/TR/2012/WD-FileAPI-20121025/

Uhrhane, E. (2012). File API: Writer. http://www.w3.org/TR/2012/WD-file-writer-api-20120417/