QWIC: Performance Heuristics for Large Scale Exploratory User Interfaces

Daniel A. Smith, Joe Lambert, mc schraefel
School of Electronics & Computer Science
University of Southampton, UK
{ds, jl2, mc}@ecs.soton.ac.uk

David Bretherton
Music, School of Humanities
University of Southampton, UK
d.bretherton@soton.ac.uk

ABSTRACT
Faceted browsers offer an effective way to explore relationships and build new knowledge across data sets. So far, web-based faceted browsers have been hampered by limited feature performance and scale. QWIC, Quick Web Interface Control, describes a set of design heuristics to address performance speed both at the interface and the backend to operate on large-scale sources.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors, Performance

Keywords: Faceted Browsing, Scalability, performance

INTRODUCTION
Faceted browsers have been shown to be an effective way to explore information spaces by being able to organize searches via metadata attributes [Hearst2009, Huynh2007]. Mainly however, faceted data browsers have been limited in UI features and the size of data sets over which they can operate. Exhibit [Huynh2007] offers multiple facets over only approximately 200 records; Flamenco [Hearst2009] was likewise demonstrated over a small database. The challenges to improve faceted browsers' performance have been at both the front- and back-end. For instance in the mSpace column browser [schraefel2006], one facet instance selection triggers filtering on all facets forward to it, and highlights all related facets behind it, as well as returning a list of results. In order to return these results, 3 separate queries, 1 for each of the 3 operations must be processed live. Where there may be 30+ possible facets to filter over 200k records for arbitrary real-time selection, as in our musicSpace project, performance optimisation is critical.

Our motivation for optimising performance, even for a research prototype is based on experience in field trials: if response times in a search interface is perceived to be slow, users walk away from the tool, regardless of its possible benefits to them. We cannot test the effectiveness of a tool over time, or its impact on tasks, for example, if the user interface performance over that data is not efficient.

In this poster & paper, we describe QWIC, Quick Web Interface Control, a set of three techniques we have developed to accelerate both front-end and back-end performance of web-based faceted browsers over large data sets. The three QWIC heuristics are: Real Separation of client and server; Viewports rather than paging; and Fully Compounded Indexes of facets.

Our approach is based on tuning the mSpace interface service [Bretherton2009] for many datasets, but in particular in the musicSpace project, which comprised over 200k records from commercial and research partners: the British Library Music Collections & Sound Archives, Cecilia, COPAC, Grove Music Online and Naxos.

REAL SEPARATION: CLIENT/SERVER ARCHITECTURE
Our experience combining Web2.0 applications and large-scale datasets has taught us to make a firm separation between the server and client component of the overall system. This sounds obvious but can be tempting to ignore in research prototypes, as Web2.0 applications typically keep the logic in the client. We have learned that each should have a clear role: the client managing user interaction, and the server doing the heavy lifting and data processing.

The Web2.0 application works best as a "thin client" with no data manipulation ability and no direct querying of the underlying data. Forwarding all queries to the server works best. This is important as it ensures the client is both lightweight and generic. The client does not need to be concerned with the context of the data, it must only know how to render it and enable users to manipulate how it is rendered and explored.

We also found, through many variations[1], that a stateless client offers the simplest decoupling from the server. When the client is not responsible for maintaining state between server requests there is less chance the client and server can become de-synchronised. Each interaction and subsequent server request results in an update to the client's internal data structures. This enables all system logic to be handled exclusively by the server application and keeps the client simple and completely generic. We maintain state by creating a new URL for each selection and manipulation that the user makes.

UIST'10, October 3–6, 2010, New York, NY, USA.

[1] http://blog.mspace.fm/2009/09/18/agile-development-in-practice/

These principles are mirrored in the open-source mSpace codebase[2] that we use to visualise the musicSpace data. Such an architecture requires careful consideration to the design of the communication protocol between server and client. To ensure that the network communication is as fast as possible, the structure of the response payload needs to be as lightweight as is manageable. However such a lightweight payload may require more processing on receipt by the client, and as such the computational overhead required might negate the benefits of the smaller payload. We found with musicSpace that the best compromise between small payloads and quick processing came from a reduced-size JSON format, designed to reduce data redundancy and repetition, that could be directly evaluated by the JavaScript client. In summary:

1) Separate the role of the server and client application.

2) Ensure the client is stateless.

VIEWPORT BASED UI QUERYING

As datasets referenced in the UI grow, the number of items associated with a facet also increase. For musicSpace, starting with the Copac dataset there were roughly 8000 items in the "Musical Works" facet. By also including data from RISM this number grew to almost 40,000. Assuming the UI is to render these items as a list (HTML <ul>), the naïve approach is to render each item as a separate list item element (<li>). While JavaScript engines have improved dramatically over the last few years, using this approach will still very rapidly cause the client application, and often the host browser, to freeze or crash. This is because of the high number of DOM elements required, which becomes intractable for even state of the art browsers to render.

To ensure scalability we constrain the view into the facet via a viewport. To the user, the interface is unchanged but technically the implementation is much more efficient. Instead of rendering one list item for each item in the facet, only enough DOM elements are created to fill the viewports visible space. When the user scrolls, the Viewport calculates what ought to be visible if this were a regular list and updates the visible DOM elements. Although the same small number of list elements are being recycled, the illusion of scrolling a large list is given to the user.

This approach also allows for data to be loaded from the server as a just in time (JIT) service, as used by large-scale sites such as Twitter and Facebook. For example a facet may have 2000 items, however the Viewport would only load the first 300 items, dynamically loading more when the user scrolls near to the end of the known data. This differs from typical pagination in that the requests are made transparently to the user, starting before the user requires the data. This reduces the response size of each request and prevents sending items that are never displayed to the user.

For short lists, such as 100 or less items, this approach is not optimal as browsers are able to handle the relatively low number of DOM elements required. The native scrolling list is therefore preferable in these cases as the programmatic overhead of our solution may yield no benefit. In summary:

1) Only transmit data which will be visible to the user.

2) Use viewports to more efficiently render long lists.

FULLY COMPOUNDED INDEXES

When creating a user interface over an existing database, it may seem obvious to use the existing table as they are. However, when databases schemas are designed, data integrity is often the main requirement, and these schemas are not always the optimal way to structure data for exploratory browsing. Database performance can be improved by avoiding inlcuding unnecessary information in a search space. By partitioning fields into multiple tables, less memory may be required when querying with multiple facets (table joins), resulting in better query performance. Similarly, if all fields used in a query are included in a compound index, tables may not need to be loaded into memory at all, thus solving queries by index lookup alone.

For example, if a user makes a selection in the facets of "Composer", "Track" and requests the "Album" facet, then a compound index of ("Composer", "Track", "Album") will result in optimal performance. Configuring the particular permutations for different database schemas can be tedious, and there are tools available to create compound indexes, such as the mSpace Data Picker. In summary:

1) Create compound indexes over possible paths through fields that can be selected or ordered by.

2) Create redundant, separately normalised database tables for the UI to use, that complement any in use.

CONCLUSION

Through the use of QWIC, faceted browsing over large scale datasets can be achieved rapidly and efficiently. By following the guidelines outlined here we were able to provide musicologists with previously unobtainable querying power and speed through the musicSpace browser, enabling us evaluate it over a real use, three month field trial.

REFERENCES

1. Hearst, M. Search User Interfaces. Cambridge University Press. 2007.
2. Huynh, D.F., Karger, D.R., Miller, R.C. Exhibit: lightweight structured data publishing. WWW, 2007.
3. schraefel mc, Wilson, M.L., Russell, A., Smith, D.A. mSpace: improving information access to multimedia domains with multimodal exploratory search. Communications of the ACM, 2006.
4. Smith, D., Popov, I., schraefel, mc. Data Picking Linked Data: Enabling Users to create Faceted Browsers. Web Science Conference, 2010.
5. Bretherton, D. et al. Integrating musicology's heterogeneous data sources for better exploration. ISMIR. 2009.

[2] mSpace open-source code: http://mspace.fm/