Webzeitgeist: Design Mining the Web

Ranjitha Kumar[1] **Arvind Satyanarayan**[1] **Cesar Torres**[1] **Maxine Lim**[1] **Salman Ahmad**[2]
Scott R. Klemmer[1] **Jerry O. Talton**[3]
[1]Stanford University [2]Massachusetts Institute of Technology [3]Intel Corporation
{ranju, arvindsatya, ctorres, maxinel}@cs.stanford.edu, saahmad@mit.edu, srk@cs.stanford.edu, jerry.o.talton@intel.com

ABSTRACT
Advances in data mining and knowledge discovery have transformed the way Web sites are designed. However, while visual presentation is an intrinsic part of the Web, traditional data mining techniques ignore render-time page structures and their attributes. This paper introduces *design mining* for the Web: using knowledge discovery techniques to understand design demographics, automate design curation, and support data-driven design tools. This idea is manifest in Webzeitgeist, a platform for large-scale design mining comprising a repository of over 100,000 Web pages and 100 million design elements. This paper describes the principles driving design mining, the implementation of the Webzeitgeist architecture, and the new class of data-driven design applications it enables.

ACM Classification Keywords
H.2.8 Information Systems: Database Applications; D.2.2 Software Engineering: Design Tools and Techniques

General Terms
Design

Author Keywords
Web design; data mining

INTRODUCTION
Web knowledge discovery and data mining [23] have transformed the way people build and evaluate Web sites [15], and the way that consumers interact with them. The information gained from Web mining drives search, e-commerce, interface development, network architectures, online education, social science, and more [16].

Web data mining typically comprises three domains: usage mining, or click analysis [32]; content mining, or text analysis [24]; and structure mining, or link analysis [8]. Together, these techniques mine the *content* contained in a Web page, but ignore that content's *presentation*. In fact, most mining and knowledge discovery systems *discard* style and rendering data [39, 33]. This raises the question: what could we learn from mining design?

CHI 2013, April 27–May 2, 2013, Paris, France.

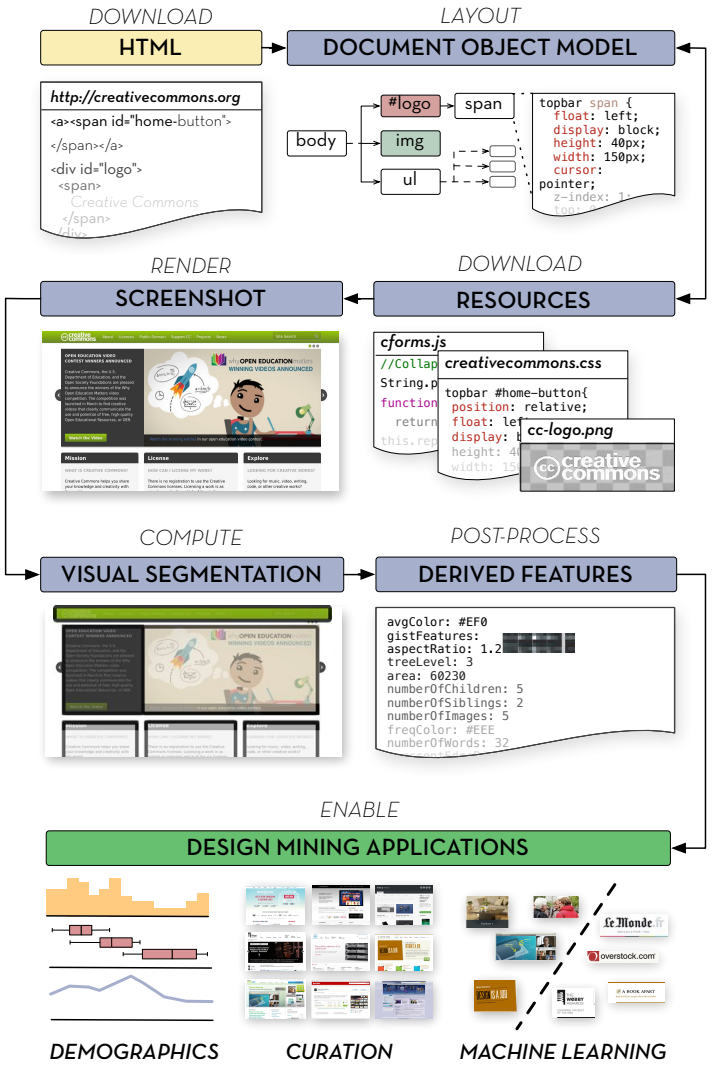


Figure 1. Webzeitgeist, a scalable platform for Web *design mining*, supplements the data used in traditional Web content mining (yellow) with information about the visual appearance and structure of pages (blue) to enable a host of new design applications (green).

This paper introduces *design mining* for the Web (Figure 1). With billions of extant pages—each comprising a concrete example of human creativity and aesthetics—the Web provides an opportunity to learn about design on a truly massive scale [18]. This paper demonstrates that applying knowledge discovery techniques to Web design data can help users understand design demographics, automate design curation, and support new data-driven design interactions.

These ideas are manifest in *Webzeitgeist*,[1] a software platform for mining and machine learning on Web design. Webzeitgeist comprises a repository of Web pages, processed into data structures that facilitate large-scale design knowledge extraction. The Webzeitgeist architecture is based on four underlying principles—scalability, extensibility, completeness, and consistency—and optimized for three common use cases: *direct access* to specific page elements, *query-based access* to identify a set of page elements which share common properties, and *stream-based access* to the repository as a whole for large-scale machine learning and statistical analysis [12].

Webzeitgeist's repository is populated via a bespoke Web crawler, which requests pages through a specialized caching proxy backed by a flexible data store. As each page is crawled and rendered, its resources are versioned and saved, and its Document Object Model (DOM) tree is snapshotted to produce a complete, static record of the page's design. Then, a set of semantic and visual features describing each DOM node are computed in a post-process and stored. Client applications access the repository through a RESTful API [30].

This paper discusses the principles that enable large-scale Web design mining, the implementation of the Webzeitgeist architecture, the repository crawled from the Web, and the API that exposes it. In addition, we demonstrate the utility of the platform by describing several data-driven design applications, including statistical analysis of design patterns, design-based search, and design-driven machine learning.

PRINCIPLES FOR DESIGN MINING

To support design mining applications, the Webzeitgeist architecture is predicated on four underlying principles.

Scalability. In rich visual domains like Web design, the utility of data mining critically depends on the size of the corpus. In a space with thousands of parameters, millions of examples are necessary to extract meaningful statistics or find relevant examples during search [10, 31]. Webzeitgeist, therefore, is designed to scale to millions of distinct page elements. Visual and semantic features are precomputed for fast access, stored in a relational database to facilitate complex queries, and duplicated in a key-value store for efficient streaming. To eliminate redundant storage of shared page resources, Webzeitgeist employs Rabin fingerprint hashing [29]. Additionally, the Webzeitgeist server uses a large memory pool to minimize disk access, backed by a hardware RAID controller with striping to make disk access fast when it does occur.

Extensibility. Since Webzeitgeist provides a general platform for design mining, not all of its eventual uses can be presently foreseen. Thus, a modular architecture facilitates the addition of new data, features, and functionality. To support transparent updates, two versions of the data store exist at all times: a *production* version that is exposed to external applications, and a *staging* version where new pages are added during crawling and features are computed. To minimize code and data dependencies, individual features are implemented as independent `C++` dynamic plugin libraries. The post-process communicates with the data store through the public API, allowing implementation details to change as long as interfaces are preserved.

Completeness. Most Web mining employs static page analysis: issuing an HTTP `GET` request for a given URL, storing the returned HTML, and parsing it [21]. To mine the *design* of Web pages, Webzeitgeist must identify and capture every resource and DOM property that contributes to a page's visual appearance. Since render-time visual properties cannot be determined through static page source analysis, Webzeitgeist uses a layout engine to process retrieved HTML into a DOM tree, and a proxy server to dynamically intercept all the resource requests made by the engine during this process.

Consistency. Dynamically-generated content poses another complication for design mining. DHTML and client-side scripting allow arbitrary code to modify the DOM based on requests made to external resources. Thus, it is nearly impossible to archive pages in a format that guarantees reproducible rendering in a browser without altering their source [28, 2]. This ephemerality frustrates many machine learning and statistical analysis applications, which expect data to remain consistent between accesses. Webzeitgeist, therefore, renders a canonical view of each page during crawling, and serializes the resultant DOM and all of its properties to the data store. Client applications and feature computations access this static snapshot of a page's design instead of interacting with the layout engine directly, and can query render-time properties without having to re-render the page.

IMPLEMENTATION

The Webzeitgeist architecture comprises five integrated components: the Web crawler, the proxy server, the data store, the post-process, and the API (Figure 2). The crawler loads pages through the proxy, which writes them to the data store. Post-processes then run on the stored pages to compute high-level features and data structures, after which client applications can access the repository through the API.

Web Design Crawler

The Web crawler consists of a set of parallel browser processes, which load pages from the Web to add them to the Webzeitgeist repository. The crawler builds a queue of URLs from a seed list. At each stage of the crawl, a browser process dequeues a URL, checks that it is not already in the repository, and requests the corresponding page. Once the page is downloaded, its HTML is parsed, and all external links are extracted and added to the queue. The Webzeitgeist crawler loads each page in a Webkit browser window [3], computes its DOM tree, and renders it. This rendering and the computed DOM are then saved in the *staging* store.

Caching Proxy Server

To identify and store a page's resources, the system routes all browser requests through a custom Web proxy. The proxy sits between the Web and the crawler, and connects directly to the Webzeitgeist data store. The proxy intercepts each resource request made by a page, downloads it from the Web, and hashes its contents. If the file does not already exist in the store, it is added.

[1] *Webzeitgeist* is a portmanteau intended to evoke the "spirit" of a Web site.

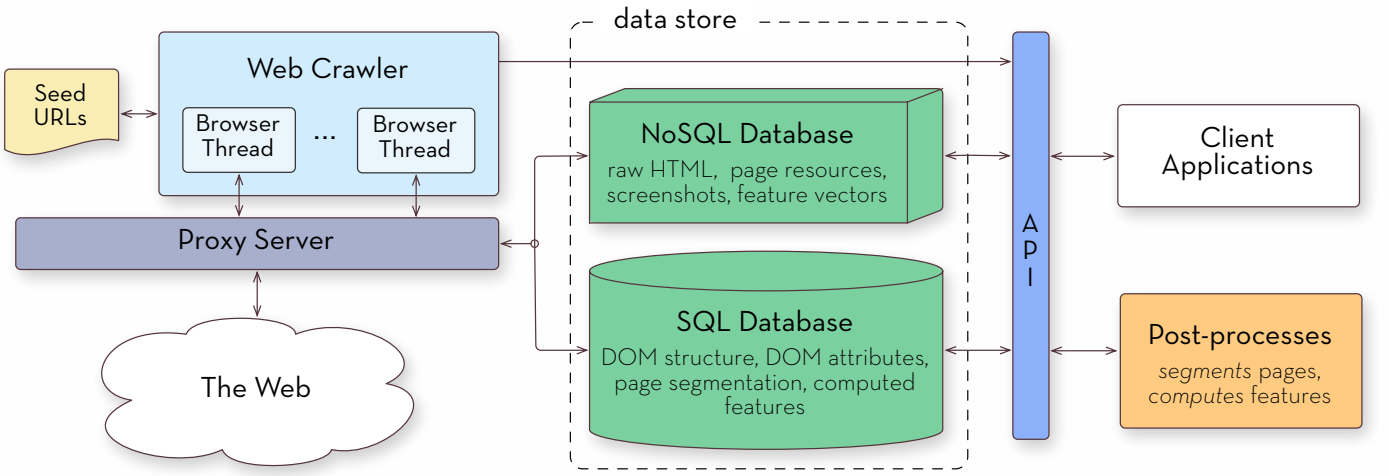


Figure 2. The Webzeitgeist architecture. A bespoke Web crawler requests pages through a caching proxy, and renders them in a set of browser threads. The proxy saves requested resources in a NoSQL key-value store, while the crawler writes the complete DOM tree for each page into a relational SQL database. Then, a set of post-process scripts are run to compute high-level features and data-structures over the stored pages. Client applications access the repository through a RESTful API.

The proxy server is also responsible for storing the graph structure of page-resource relationships. Since HTTP is a stateless protocol, this requires using custom HTTP headers to associate each resource request with the page that originated it. When the crawler requests a page, the data store generates a unique identifier and passes it back in the response header. The crawler then uses this ID to label each subsequent request the page makes to the proxy.

Two additional headers determine how the proxy services requests: adding them to the store during the crawl, or serving them directly from the database during retrieval. In the event that an application tries to retrieve a URL that does not exist in the data store, the proxy server responds with a 404 - PAGE NOT FOUND error.

Feature Post-processes

Once a page has been downloaded, converted to a DOM, rendered, and stored, a set of post-processes are run. First, Webzeitgeist computes a visual hierarchy from the DOM, discarding nodes that do not contribute to the page's rendered appearance and re-parenting nodes to ensure that parent-child relationships in the hierarchy correspond to visual containment on the page [17]. Then, the system computes a set of semantic and computer vision features over each element in the hierarchy and stores them (see Figure 3).

Next, the post-process coalesces each node's visual, semantic, and render-time features into a vector descriptor, exposing page properties in a convenient form for design mining applications. Numeric features are normalized to the range $[0, 1]$, and string-based attributes are binarized based on their possible values to generate dictionary-length bit vectors. After this conversion, each page node is associated with a 1679-dimensional descriptor consisting of 691 render-time HTML and CSS properties computed by the DOM, 960 GIST scene descriptors computed on the node's rendering [26], and 28 structural and computer vision properties.

After the feature vectors are computed, the post-process restructures the table in which DOM properties are stored. During the crawl, DOM tree data is added to a wide table which facilitates fast insertions but results in slow retrieval; partitioning this table into a *star schema* reduces retrieval times by an order of magnitude [38]. After this restructuring, the post-process migrates the data store from the staging environment to production.

Data Store

The Webzeitgeist data store comprises two databases: a NoSQL database for page resources, binary data, and the vector descriptors for page nodes; and a relational SQL database for DOM nodes and properties, the visual page hierarchies, and the associated vision and semantic feature values.

The NoSQL database is a MongoDB instance [1], which provides fast access to binary files and structures too large to be efficiently stored in SQL tables, while simultaneously supporting dynamic queries and aggregation. This database con-

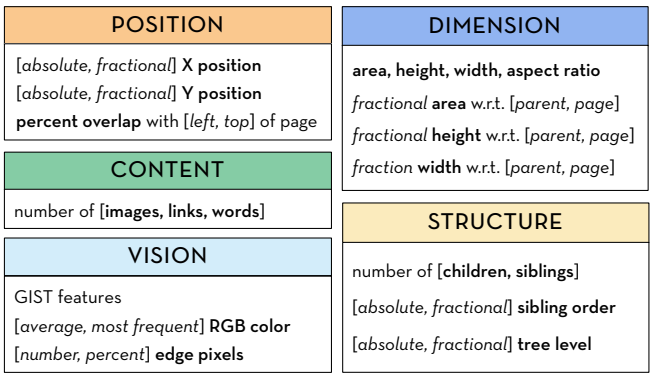


Figure 3. The semantic and computer vision features that Webzeitgeist computes over page elements.

tains the full HTML of every crawled page, its resources, and the high-dimensional feature descriptor for each visual hierarchy node.

The relational database is a MySQL instance [27], comprising five tables: PROXY CONTENT, PROXY LINKS, DOM NODES, VISUAL BLOCKS, and FEATURES (Figure 4). The PROXY CONTENT table contains metadata for every URL requested by pages during crawling, describing where the resource is from, its identifier in the NoSQL store, when it was retrieved, and a Rabin fingerprint hash of its contents. The PROXY LINKS table associates pages with a list of the resources upon which they depend. The DOM NODES table contains a record of each DOM node encountered in the crawl, along with pointers to its parent page and node; its type, name, value, and inner HTML; and all 258 render-time DOM attributes defined by the HTML4 and CSS3 standards [37, 36]. The VISUAL BLOCKS table contains the page elements that result from the visual segmentations performed during post-processing. The FEATURES table stores the visual and semantic features computed for each such block. For fast retrieval, tables are denormalized with replicated Page, DOM, and Block IDs.

API

Clients access the Webzeitgeist repository through a RESTful API, loading the appropriate endpoint URL and receiving JSON data in return [5]. Three modes are available for requesting data. The first allows *direct access* to design properties based on unique identifiers. The second allows clients to *stream* batches of data from the repository with a single request.

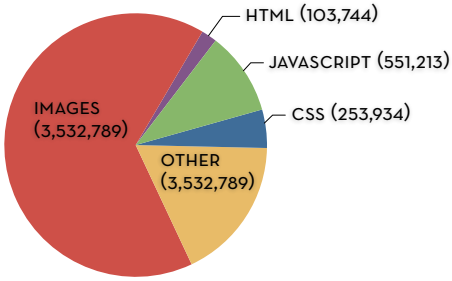
For more complex access patterns, the API also provides a custom JSON-based design query language (DQL). DQL predicates allow for filtering based on combinations of DOM attributes and computed visual features. When issuing queries, the client can also specify a list of properties that should be returned by the API call, keeping result sets succinct. The API transparently converts DQL queries into SQL and Mongo Query Language, sanitizes them to prevent injection attacks, and returns the results in JSON.

Server Hardware

The Webzeitgeist repository is hosted on a twelve-core 2.4GHz Intel Xeon server to allow complex DQL queries to be executed efficiently. The server contains 48 GB of RAM to facilitate caching and ensure that the SQL index fits in memory. The server's 4TB data store consists of fourteen 600GB 15K RPM SAS drives in a RAID 10 configuration, backed by a hardware RAID controller with a 1GB cache. The drive array is capable of sustaining 6GB/s throughput when data is being streamed from disk.

THE WEBZEITGEIST DATASET

The Webzeitgeist crawl was seeded with 20 URLs, including the Alexa Top 500, the Webby Awards gallery, and popular design blogs. To build a diverse repository, pages were crawled in a breadth-first order, self-referential domain links were skipped, and only a single copy of each resolved URL was stored. The resultant dataset contains 103,744 Web pages from 43,743 domains, with 143.2 million DOM nodes and 12.7 million visual blocks. The raw HTML content of these pages and their referenced resources together require 425GB of disk space; the SQL database requires 187GB. The pages reference over 5.3 million HTML, CSS, JavaScript, image, and other resources, including Flash, movie, and audio files (see inset figure).



The Webzeitgeist crawl was a computationally intensive task, requiring more than 35 CPU days of processing. As a representative example, the CHI 2013 homepage `http://chi2013.acm.org` references two style sheets, four JavaScript files, and five images for a total of 480KB of raw content. The crawler downloaded the page on September 13th, 2012, in 3.5 seconds. The DOM, which comprised 251 nodes, was computed in 0.1 seconds and stored via the API in 0.47 seconds. The visual segmentation algorithm ran for 2ms, producing 61

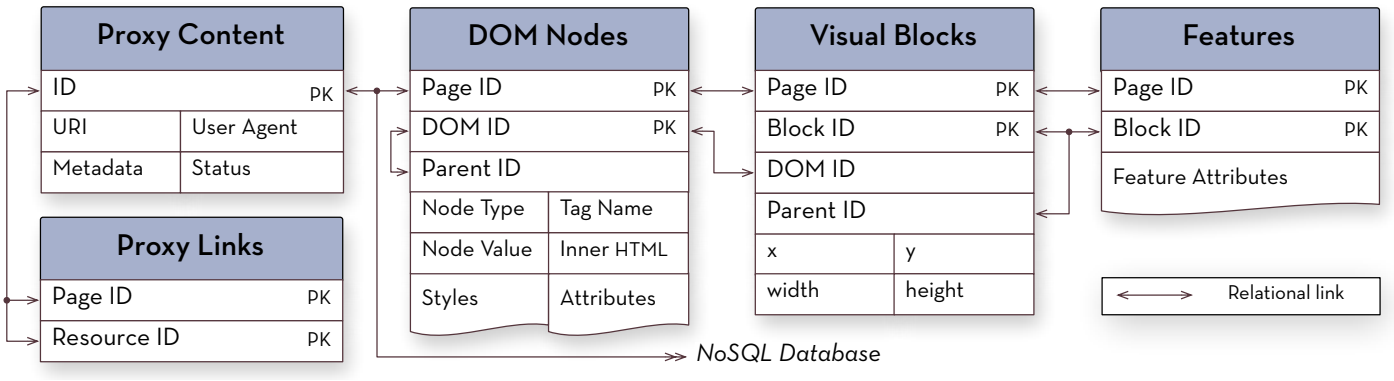


Figure 4. The schema for the Webzeitgeist data store, showing the five tables that comprise the SQL database and their contents. This denormalized structure, with replicated Page, DOM, and Block IDs, facilitates fast retrievals.

Figure 5. The 295 distinct cursors in the Webzeitgeist repository, fetched in 47.8 seconds. This query searches the CSS `cursor` property on DOM nodes, looks up the ID in the PROXY CONTENT table, and fetches the associated file from NoSQL.

visual blocks; visual feature computation ran for 13.53 seconds. Writing this segmentation and the associated features to the database took another 1.04 seconds, for a total processing time of 18.64 seconds.

DESIGN MINING IN ACTION

The Webzeitgeist design mining platform enables content producers to answer questions about design practice and software developers to build next-generation design tools. Designers can query Webzeitgeist to understand design demographics and search for examples of design patterns and trends [7, 11], without relying on manual curation. Application developers can apply machine learning techniques to Web design problems without incurring the overhead of crawling, rendering, and sanitizing Web data. Webzeitgeist significantly lowers the barrier to data-driven Web design, facilitating analysis on a scale 50–300 times larger than prior work [14, 31].

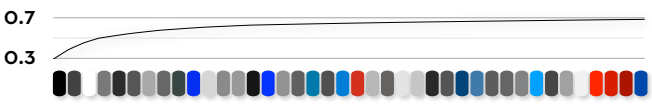
Design Demographics

Designers often seek to understand the space of options along a particular design axis [20]. For instance, a designer who wants to customize the cursor on her Web page might look at a gallery of cursors used on other pages for inspiration. We can query Webzeitgeist to return all the distinct cursors in the repository:

```
POST, /v1/dom.json
query = {
  "$select": [
    {"@styles": {"$distinct": "cursor"}
  ]
}
```

This DQL query—which executed in 47.8 seconds—first finds cursors by examining the CSS `cursor` property across DOM nodes, and then fetches the corresponding files from the NoSQL database. A "cursory" inspection of the 295 results shows that arrows, hands, cartoon characters, and celebrity faces are all popular choices (Figure 5).

Webzeitgeist can answer similar questions about popular text color choices, for instance by computing a cumulative distribution function over the CSS `color` property:



The forty most popular text colors in the database account for nearly 70 percent of all text color; most are shades of grey.

Webzeitgeist affords the ability to examine distributions over both page- and node-level Web properties. For instance, an information architect might use Webzeitgeist to compute statistics on the visual complexity of pages. Figure 6 (*top row*) shows that the mode depth of a page's DOM tree is six, and that most pages contain between 50 and 200 DOM nodes. To investigate the cause of the sharp spike in the latter histogram, the architect can request IDs for pages with only a single DOM node and inspect their HTML: unsurprisingly, these pages are predominantly Flash-based.

Similarly, a designer might wish to inspect common properties for individual page assets to guide the design of new content. Calculating a histogram over the *aspectRatio* of visual nodes reveals that there are many square elements, but that page elements, on average, are wider than they are tall (Figure 6, *bottom left*). Computing a histogram over the CSS `opacity` property and examining values less than 1, reveals sharp peaks at .5, .65, .75, and .8 (Figure 6, *bottom right*).

Since Webzeitgeist stores HTML properties in addition to design data, we can also use the repository to revisit HTML demographics in a new way. In 2005, Google released the results of a large-scale survey of popular HTML `class` names [9]. Webzeitgeist allows us to take this study one step further, and understand the relationship between static HTML properties and dynamic render-time ones. Since Webzeitgeist records the render-time bounding box for each DOM node, we can compute spatial probability distributions for the most popular CSS selectors (Figure 7). The striking patterns that result indicate that the visual positions and semantic roles of some page elements are highly correlated.

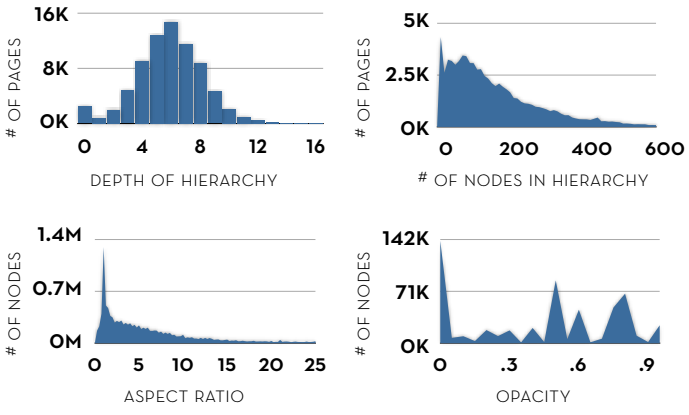


Figure 6. *Top row*: distributions over page-level properties: depth (left) and number of nodes (right) in a page's visual hierarchy. *Bottom row*: distributions over node-level properties: *aspect ratio* feature (left) and CSS `opacity` (right).

Figure 7. Spatial probability distributions for frequently occurring HTML `id` and `class` attributes demonstrate striking visual correlations.

Design Queries

Designers are often interested in understanding the context of particular patterns and trends. Many design blogs maintain small, curated sets of examples showcasing notable Web design techniques.

To give designers more powerful search and collection capabilities, Webzeitgeist introduces the ability to quickly create dynamic collections that exhibit particular design characteristics. For instance, one distinctive technique discussed on design blogs is the use of long, scrolling horizontal layouts.

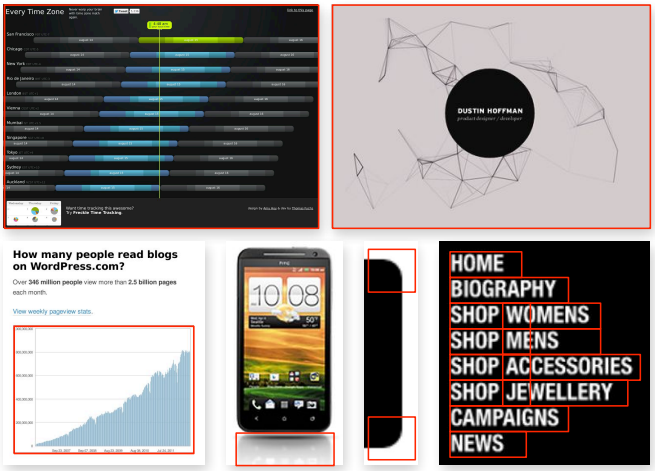


Figure 9. Selections from some of the 4943 pages containing <CANVAS> elements in the Webzeitgeist repository, demonstrating interactive interfaces, animations, graphs, reflections, rounded corners, and custom fonts.

To find such pages, we queried Webzeitgeist for pages with *aspectRatio* greater than 10.0:

```
POST, /v1/pages.json
query = {
  "$select": [{"$distinct": "page_id"}],
  "$where" : [
    {"@visual": {"aspectRatio": {"$gt": 10} } }
  ]
}
```

This query produced 68 horizontally-scrolling pages in 1.1 seconds. Figure 8 shows a few representative results.

Querying Webzeitgeist with constraints based on HTML markup can also shed light on design trends. The W3C describes the <CANVAS> element—introduced in HTML5—as a scriptable graphics container. The specification, however, gives little insight into how the tag is actually used. Webzeitgeist returns all 201,658 <CANVAS> elements in the database in 2.4 minutes. Figure 9 shows representative uses.

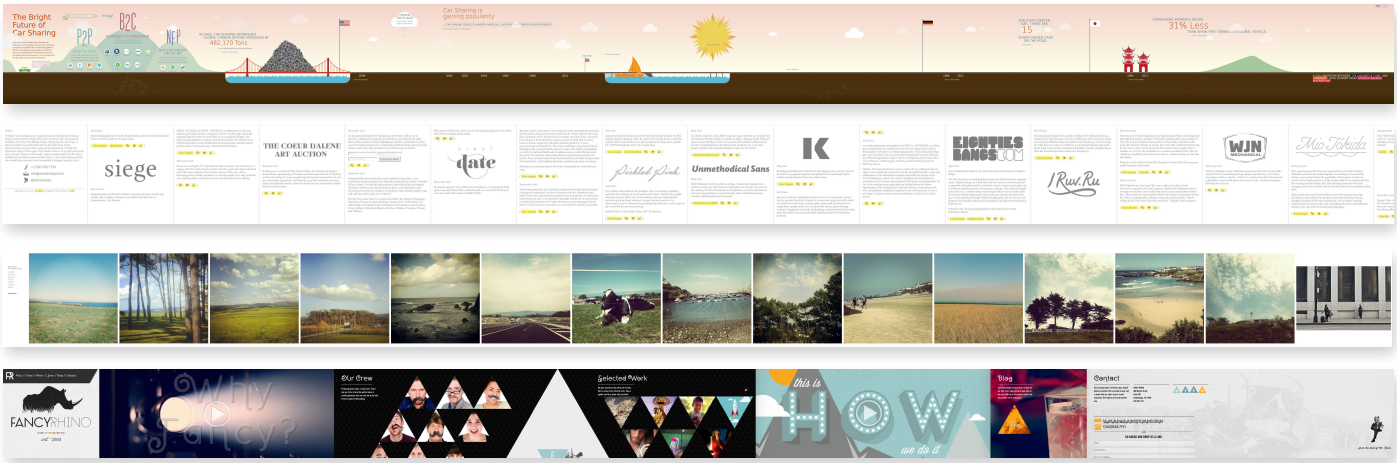


Figure 8. Four of the 68 query results for pages with horizontal layouts. Blogs, image/photo galleries, and vector art pages are a few of the representative styles in the results set.

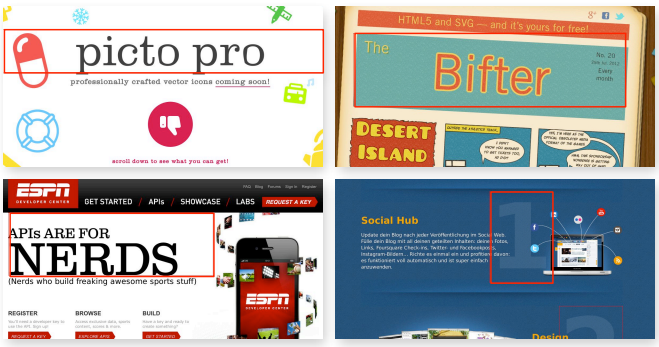


Figure 10. Query results for nodes containing large typography, demonstrating large text in logos, site titles, hero graphics, and background effects. The query identified 6856 DOM nodes from 1657 distinct pages, and executed in 56 seconds.

Webzeitgeist allows us to investigate another problematic aspect of Web design: typography. Although the CSS `@font-face` rule was introduced in 1998 [6], technical and licensing issues with embedding custom Web fonts have traditionally relegated complex typographic effects to images. We can use Webzeitgeist to search for examples of prominent Web font typography, querying for nodes with a CSS `font-size` property greater than 100 pixels. Figure 10 shows a few of the 6856 results, which occur in only 1657 distinct pages.

We can build more complex queries by specifying more constraints. To learn the different ways in which semi-transparent overlays are used, we queried for nodes with a solid background color where the *alpha* value of the CSS `background-color` property is between 0 and 1 (Figure 11).

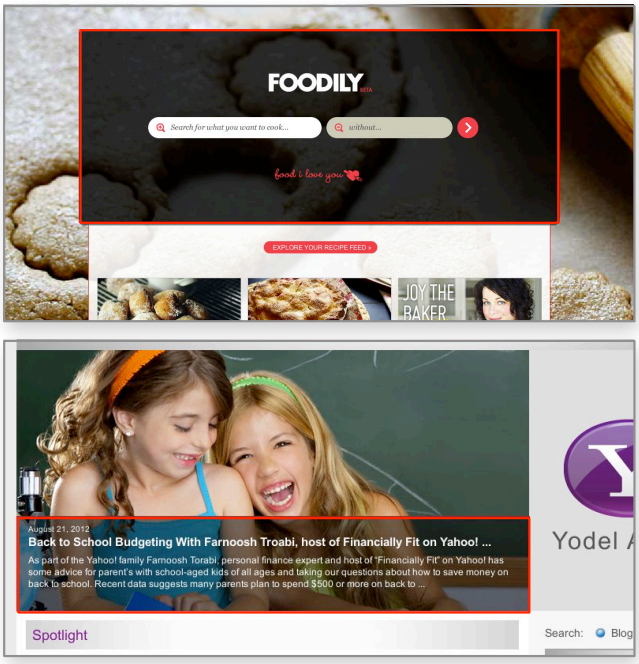


Figure 11. Query results for semi-transparent overlays. The query identified 9878 DOM nodes from 3435 distinct pages, and executed in 48.1 seconds. Semi-transparent overlays are often used on top of photographs, either as a way to display content over of a busy photographic background (top) or to frame captions (bottom).

We can also use Webzeitgeist to construct queries over high-level design concepts. Suppose a designer wants to browse "search engine-like" pages. This concept is loaded with design constraints, but we can approximate it in DQL as a query for pages with a centered <INPUT> element and low visual complexity:

```
POST, /v1/pages.json
query = {
  "$select": [{"page_id": 1}],
  "$where": [
    {
      "@dom": {
        "tagName": "INPUT",
        "type": "text",
      "@visual": {
        "leftSidedness": {"$or": {"$gte": 0.4}, {"$lte":0.6}},
        "topSidedness":  {"$or": {"$gte": 0.4}, {"$lte":0.6}}
      }
    },
    {"@visual": {"$cnt": {"$lt": 50} } }
  ]
}
```

Figure 12 shows a few results from this query.

Similarly, attribute queries can be composed to search for pages with specific visual layouts. Figure 13 shows a sample layout with a large header, a top navigation bar, and a large body text node. We can encode this layout in a DQL query that searches for pages with a header that takes up more than 20 percent of the page's area, a navigation element that is positioned in the top 10 percent of the page's height, and a text node that contains more than 50 words. This example illustrates the kinds of applications that Webzeitgeist might engender: imagine a search interface that automatically formulates queries from sketches like the one shown in the figure.

Figure 12. Query results for "search engine" pages: roughly centered (vertically and horizontally) text `INPUT` elements, and fewer than 50 visual elements on the page. This query produced 209 pages and executed in 3.9 minutes. Some login and signup pages are also returned (bottom right).

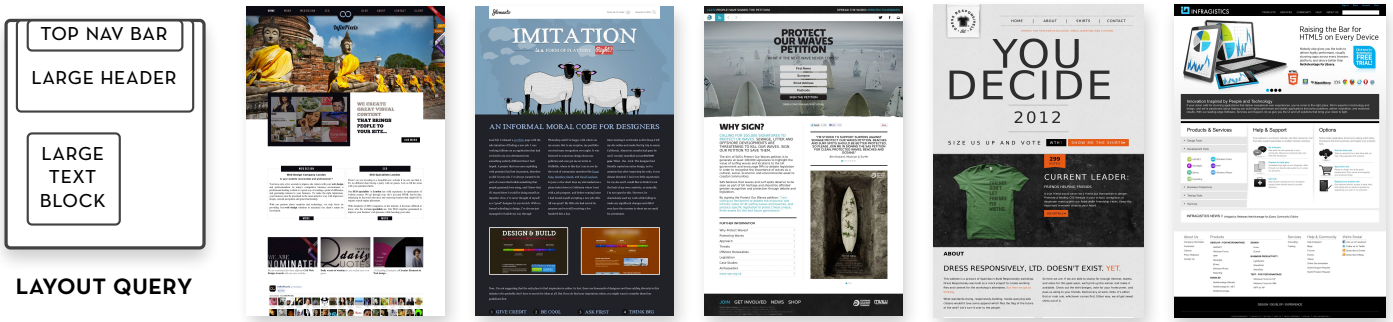


Figure 13. Five of the 20 search results for the three-part DQL layout query visualized on the left. The query, which executed in 2.1 minutes, returns pages that share a common high-level layout, but exhibit different designs.

Machine Learning

Webzeitgeist also enables a new kind of design-based machine learning. For the first time, applications can stream structured visual descriptors for page elements from a central repository. Moreover, Webzeitgeist's extensible architecture allows new data to be collected and integrated with the repository for supervised learning applications, for instance via crowdsourcing.

Classification. Lim et al. [22] used Webzeitgeist as a backend to train structural semantic classifiers for concepts like ARTICLE TITLE, ADVERTISEMENT, and PRODUCT IMAGE. In an online study, they collected a set of more than 20,000 semantic labels over more than 1000 distinct pages. They then used the descriptors associated with page elements to train 40 binary Support Vector Machine classifiers, reporting an average test accuracy of 77 percent. In the future, these and other similar classifiers could be used to support Web accessibility, guide attempts to "semantify" the HTML standard, and allow designers to search for pages that match a given visual "feel."

Metric Learning. Machine learning techniques can also be used to enable example-based search over the repository. Using Lim et al.'s label data, we induced a distance metric in the 1679-dimensional descriptor space using OASIS, a metric-learning algorithm originally developed for large-scale image comparison [4]. The method takes as input sets of identically-labeled page elements, and attempts to learn a symmetric positive-definite matrix that minimizes interset distances. Once learned, this metric can be used to perform query-by-example searches on page regions via a nearest-neighbor search in the metric space. These nearest-neighbor computations can be performed in realtime via locality sensitive hashing [13].

Example-based search provides a powerful mechanism for navigating complex design spaces like the Web [31]. Figure 14 shows three example queries and their top results, demonstrating how Webzeitgeist can be used to search for alternatives for a given design artifact, and to identify template reuse between pages. The utility of this search interaction critically depends on the full Webzeitgeist feature space. For comparison, Figure 15 shows nearest-neighbor results for the top query in Figure 14 using only the vision-based GIST descriptors. While these elements are visually reminiscent of the query, they bear little structural or semantic relation to it.

DISCUSSION AND FUTURE WORK

This paper demonstrates—for the first time—the value of large-scale mining of design data, and offers a new class of data-driven problem-solving techniques to the design community. While the paper showcases several concrete design interactions, we imagine that the applications that eventually arise from design mining will greatly outstrip our power to predict them.

There are a number of directions for future work. Scaling the database by several orders of magnitude would increase the accuracy and utility of many design-mining applications. While the current indexing strategy for Webzeitgeist should scale to about five million pages, crawling a more substantial portion of the Web would require porting the infrastructure to a distributed computing and storage platform. More flexible and powerful backing stores (for instance, graph databases [25]) may also make it easier to formulate complex queries that span multiple levels of page hierarchy (*e.g.,* "Find all the nodes whose children are all <IMAGE> elements").

In addition to crawling more pages, altering the crawl's selection policy to capture additional information from each visited site could provide a more holistic view of Web design practice. By spoofing USER-AGENT headers and requesting pages with browser windows of varying sizes, the repository could detect responsive Web designs, or pages with layouts that adapt to the viewing environment. This type of mining would help users understand design patterns across different form factors (*e.g.*, mobile, tablet, and desktop). Expanding Webzeitgeist to support site-level mining by sampling several pages from each visited domain could help designers analyze how individual page elements are reused and adapted. Aggregating multiple versions of pages over time could allow users to build data-driven models of Web design evolution [2, 35].

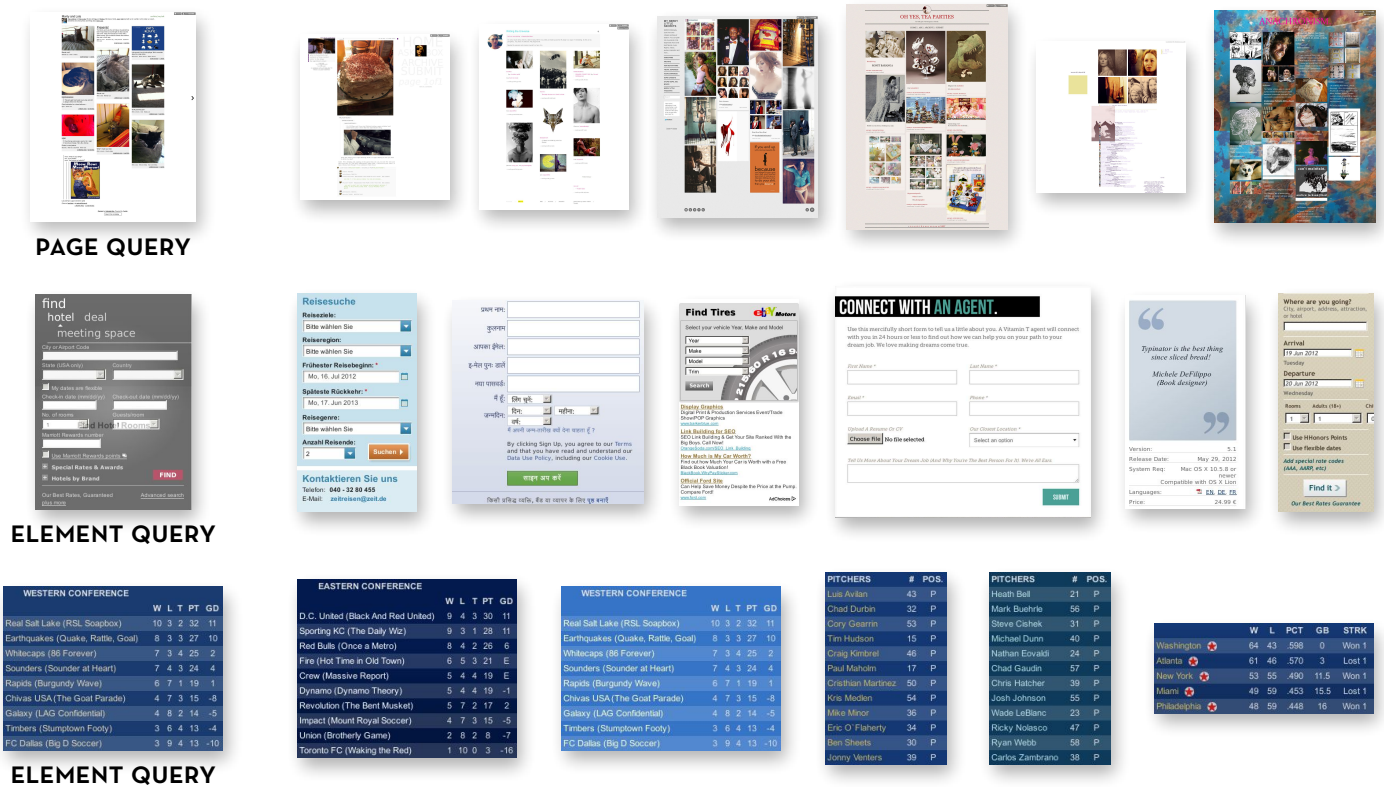


Figure 14. Top search results from querying the repository using the learned distance metric and three query elements. These results demonstrate how Webzeitgeist can be used to search for design alternatives (top, middle), and to identify template re-use between pages (bottom).

Perhaps the most exciting avenue for future work is using the repository to realize new machine learning applications, or reimplementing existing methods at scale [14, 31, 34]. Exploiting model and hardware parallelism has made it feasible to train models with billions of parameters on Web-scale datasets with millions of examples, leading to a number of breakthroughs in unsupervised learning [19]. In addition, using the Webzeitgeist platform to bootstrap crowdsourced data collection may enable a host of new supervised learning applications.

We hope that Webzeitgeist will lower the barrier to building data-driven design applications and engender a new class of Web design tools. For more information, please visit **`http://hci.stanford.edu/research/webzeitgeist`**.

ACKNOWLEDGMENTS

We thank Dan Fike, Andreas Paepcke, Richard Socher, Tom Yeh, David Karger, the Stanford HCI Group, and the anonymous reviewers for their helpful comments and suggestions. This research was supported by the National Science Foundation under Grant No. 0846167, a Google PhD Fellowship, and an SAP Stanford Graduate Fellowship.

REFERENCES

[1] 10gen, I. *MongoDB*. 2012. `http://www.mongodb.org`.

[2] Adar, E., Dontcheva, M., Fogarty, J., and Weld, D. S. Zoetrope: interacting with the ephemeral web. In: *Proc. UIST*. 2008, 239–248.

[3] Apple Inc. *The WebKit open source project*. 2012. `http://www.webkit.org/`.

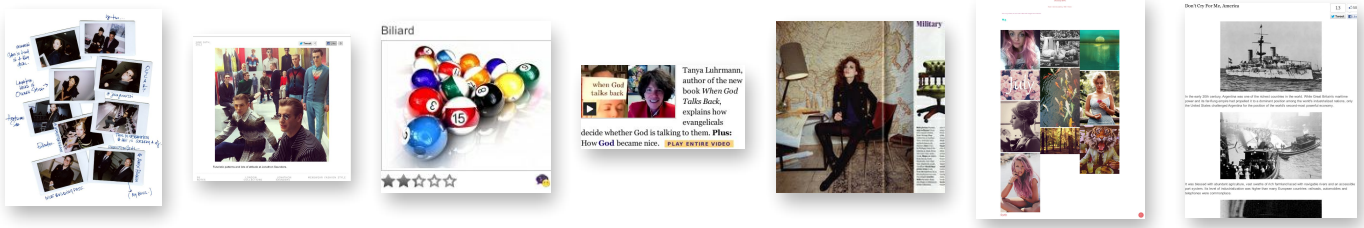


Figure 15. The top matches for the first (page) query in Figure 14 using *only* the vision-based GIST descriptors. While these elements are visually reminiscent of the query, most of them bear little structural or semantic relation to it.

[4] Chechik, G., Sharma, V., Shalit, U., and Bengio, S. An online algorithm for large scale image similarity learning. In: *Proc. NIPS*. 2009.

[5] Crockford, D. *The application/json Media Type for JavaScript Object Notation (JSON)*. Tech. rep. RFC 4627. IETF, July 2006.

[6] CSS Working Group. *Cascading style sheets, level 2*. May 1998. http://www.w3.org/TR/2008/REC-CSS2-20080411/.

[7] Duyne, D. K. van, Landay, J. A., and Hong, J. I. *The Design of Sites*. Addison-Wesley, 2002.

[8] Getoor, L. and Diehl, C. P. Link mining: a survey. *SIGKDD Explorations Newsletter* 7, 2 (2005), 3–12.

[9] Google Inc. *Web Authoring Statistics*. 2005. https://developers.google.com/webmasters/state-of-the-web/.

[10] Hays, J. and Efros, A. A. Scene completion using millions of photographs. In: *Proc. SIGGRAPH*. ACM, 2007.

[11] Herring, S. R., Chang, C.-C., Krantzler, J., and Bailey, B. P. Getting inspired!: understanding how and why examples are used in creative design practice. In: *Proc. CHI*. 2009.

[12] Hirai, J., Raghavan, S., Garcia-Molina, H., and Paepcke, A. WebBase: a repository of Web pages. *Computer Networks* 33, 1–6 (2000), 277–293.

[13] Indyk, P. and Motwani, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In: *Proc. STOC*. 1998.

[14] Ivory, M. Y. and Hearst, M. A. Statistical profiles of highly-rated Web sites. In: *Proc. CHI*. 2002, 367–374.

[15] Kohavi, R., Henne, R. M., and Sommerfield, D. Practical guide to controlled experiments on the web: listen to your customers not to the HiPPO. In: *KDD*. 2007.

[16] Kosala, R. Web mining research: a survey. *SIGKDD Explorations* 2 (2000).

[17] Kumar, R., Talton, J. O., Ahmad, S., and Klemmer, S. R. Bricolage: example-based retargeting for Web design. In: *Proc. CHI*. ACM, 2011.

[18] Kumar, R., Talton, J. O., Ahmad, S., and Klemmer, S. R. Data-driven Web design. In: *Proc. ICML*. 2012.

[19] Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J., and Ng, A. Building high-level features using large scale unsupervised learning. In: *Proc. ICML*. 2012.

[20] Lee, B., Srivastava, S., Kumar, R., Brafman, R., and Klemmer, S. R. Designing with interactive example galleries. In: *Proc. CHI*. 2010.

[21] Lee, H.-T., Leonard, D., Wang, X., and Loguinov, D. IRLbot: scaling to 6 billion pages and beyond. In: *Proc. WWW*. ACM, Beijing, China, 2008, 427–436.

[22] Lim, M., Kumar, R., Satyanarayan, A., Torres, C., Talton, J. O., and Klemmer, S. R. *Learning Structural Semantics for the Web*. Tech. rep. CSTR 2012-03. Stanford University, 2012.

[23] Liu, B. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Springer-Verlag, 2011.

[24] Liu, B. and Chen-Chuan-Chang, K. Editorial: special issue on web content mining. *SIGKDD Explorations Newsletter* 6, 2 (2004), 1–4.

[25] Neo Technology. *Neo4j*. 2012. http://neo4j.org/.

[26] Oliva, A. and Torralba, A. Modeling the shape of the scene: a holistic representation of the spatial envelope. *International Journal of Computer Vision* 42, 3 (2001), 145–175.

[27] Oracle Corporation. *MySQL*. 2012. http://www.mysql.com.

[28] Pollak, B. and Gatterbauer, W. Creating permanent test collections of Web pages for information extraction research. In: *Proc. SOFSEM*. 2. 2007, 103–115.

[29] Rabin, M. O. *Fingerprinting by random polynomials*. Tech. rep. Center for Research in Computing Technology, Harvard University, 1981.

[30] Richardson, L. and Ruby, S. *Restful Web services*. First edition. O'Reilly, 2007.

[31] Ritchie, D., Kejriwal, A., and Klemmer, S. R. d. tour: style-based exploration of design example galleries. In: *Proc. UIST*. 2011.

[32] Srivastava, J., Cooley, R., Deshpande, M., and Tan, P.-N. Web usage mining: discovery and applications of usage patterns from Web data. *SIGKDD Explorations Newsletter* 1, 2 (2000), 12–23.

[33] Sun, F., Song, D., and Liao, L. DOM based content extraction via text density. In: *Proc. SIGIR*. ACM, Beijing, China, 2011, 245–254.

[34] Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N. D., and Měch, R. Learning design patterns with Bayesian grammar induction. In: *Proc. UIST*. 2012.

[35] Teevan, J., Dumais, S. T., Liebling, D. J., and Hughes, R. L. Changing how people view changes on the Web. In: *Proc. UIST*. 2009.

[36] W3C Working Group. *Cascading style sheets snapshot 2010*. May 2011. http://www.w3.org/TR/CSS/.

[37] W3C Working Group. *HTML 4.01 specification*. Dec. 1999. http://www.w3.org/TR/html401/.

[38] Wikipedia. *Star schema — Wikipedia, the free encyclopedia*. [Online; accessed 19-September-2012]. 2012. http://en.wikipedia.org/wiki/Star_schema.

[39] Yi, L., Liu, B., and Li, X. Eliminating noisy information in Web pages for data mining. In: *Proc. SIGKDD*. ACM, 2003, 296–305.