Master Thesis

An Extensible Content Aggregation Engine for the Web

Author(s):
Church, Tim

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-010056897

Rights / License:
In Copyright - Non-Commercial Use Permitted
An Extensible Content Aggregation Engine for the Web

Master Thesis

Tim Church
<tchurch@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Matthias Geel

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

4th January 2012
Abstract

The Web offers a vast repository of information on almost any topic imaginable, but information for a single topic is often dispersed across many different websites and presented in a variety of diverse formats. Some web pages may only provide unstructured text while others may embed various amounts of semantic metadata within the document. This fragmentation and diversity makes it difficult for users to gather and make use of information encountered while browsing. Our aim is to simplify data collection on the web and empower users with complete control over the presentation and use of this information. Liberating data from individual web pages allows it to be combined, repurposed, organized, and personalized according to the individual needs of the user.

In this thesis, we present Sift, a system for collecting and managing structured data from the web. Our approach combines multiple data extraction and integration techniques to provide a flexible and robust information harvesting platform. Our solution helps non-technical end-users discover, gather, organize, and manage structured information found while browsing the web. We provide a browser-integrated interface which enables users to quickly and easily collect individual information items as they browse the web, as well as a web interface which allows users to browse, organize, and manage their collected data.
# Contents

1 Introduction ................................................... 1
   1.1 Goals of the Thesis .................................... 2
   1.2 Structure of the Thesis ................................ 3

2 Background Information and Related Work ............... 5
   2.1 Bookmarking ........................................... 5
   2.2 Web Snippets .......................................... 6
   2.3 Web Feeds ............................................. 6
   2.4 Web Scraping .......................................... 7
   2.5 Semantic Web ......................................... 8
       2.5.1 Schema.org ...................................... 9
   2.6 Data Warehousing .................................... 10
       2.6.1 Data Integration .................................. 11

3 Approach .................................................... 13

4 System Design and Architecture .......................... 19
   4.1 System Architecture .................................. 19
   4.2 Design Decisions ..................................... 21
       4.2.1 Data Model ....................................... 21
       4.2.2 Database .......................................... 22

5 Implementation .............................................. 25
   5.1 Firefox Extension .................................... 25
       5.1.1 Notifications .................................... 26
       5.1.2 Extraction Wizard ................................. 28
   5.2 Content Aggregation Engine .......................... 30
       5.2.1 Data Extraction .................................. 31
       5.2.2 Data Transformation ............................... 33
       5.2.3 Data Validation ................................... 35
       5.2.4 Duplicate Detection ............................... 36
       5.2.5 Revision History .................................. 37
   5.3 Web Interface ......................................... 38
       5.3.1 Organize ........................................... 39
       5.3.2 Browse ............................................ 40
       5.3.3 View ............................................... 42
       5.3.4 Create, Edit, & Delete ............................ 43
# CONTENTS

5.3.5 Manage .................................................. 43

6 Conclusion ................................................. 45
6.1 Contributions ............................................. 45
6.2 Future Work .............................................. 46

A How to Extend the System .......................... 49
A.1 Developing a Custom Extractor ................. 49
A.2 Installing a Custom Extractor ................. 52

B Proposed Date Similarity Metrics ............... 55
Introduction

The Web is the single largest source of information ever assembled throughout history. There are estimated to be over 555 million websites on the Internet [14], and this number is growing rapidly. The amount of data available online is expanding at an incredible pace as organizations increasingly provide online access to existing databases and community websites enable users to generate an abundance of new data. Accordingly, the Web offers a vast repository of information on almost any topic imaginable. However, information on a single topic is often spread across thousands of websites and is not always in a format that supports end-users’ needs. Therefore, the task of collecting structured information from multiple websites in heterogeneous formats is a difficult challenge.

Our work is motivated by the desire to free information from the borders of individual websites in order to give the end-user complete control over its presentation and organization, as well as to enable the data to be repurposed and combined in new ways. For this to be possible, we must extract not only the web content but also the semantic meaning of each piece of information. The vast majority of websites have access to this semantic metadata internally because they store their data in a database with a well-defined schema, but those databases are not publicly available. Instead, they are used to dynamically populate HTML templates that define the web pages which can be accessed online. This process flattens the relational entity for presentation and often results in the loss of most or all of the semantic metadata that is defined in the database schema. Our challenge is to attempt to reconstruct the original database entity based only on the information presented on the web page.

The Semantic Web aims to enrich the content of web pages with machine-readable semantic metadata, in order to create a web of data which can be manipulated directly by computers. The Semantic Web, if fully implemented, would enable us to extract structured data from web pages with ease. However, even though the vision for the Semantic Web dates back to 1994, Tim Berners-Lee and colleagues recently acknowledged that “This simple idea, however, remains largely unrealized” [18]. One reason that it has not yet taken off is that
there is little or no incentive for publishers to make this information freely available as it could reduce traffic to their website and potentially negatively affect advertising revenue. Also, the Semantic Web has suffered from a number of competing initiatives. In addition to the official standards from the W3C, there have also been several commercial and community efforts such as Open Graph protocol, Microformats, and Schema.org. This fragmentation causes different websites to implement different standards and makes it difficult for a single system to support all formats. Most previous web collection tools narrowly focus on a single metadata format such as RDF and ignore the wealth of semantic markup available in alternative formats.

Once data has been collected from the web, there is still the problem of managing that information over time. The dynamic nature of the Internet allows information to be added, removed, or updated at any moment. Data published on a web page can evolve over time and may be updated on a regular basis. Manually checking each website for updates is impractical as it is inefficient and time-consuming for the user. Web feeds attempt to solve this problem by allowing publishers to push updates to subscribers, and web feed aggregators allow users to check for updates on multiple sites from a single location. However, web feeds are limited to specific data formats that usually lack semantics and also have limited availability because they rely on the content provider to publish an appropriate feed. Although web feeds are only a partial solution, we can utilize techniques from web feed aggregators to ensure that collected data in our system is kept up-to-date over time.

In this thesis, we present a system to facilitate the collection and management of structured data from the web called Sift. Our solution creates a ‘personal data warehouse’ in which a user can save and organize information while browsing. This allows data items from diverse websites to be browsed and categorized together, regardless of their original location or format. A browser-integrated interface enables non-technical users to easily gather information items as they browse the web. Extraction from several common semantic markup formats is supported, and custom extraction rules can also be created through the provided extension mechanism. Additionally, the system enforces data integrity and automatically updates collected information when changes are made to the original source web page. Finally, a web interface allows users to browse, organize, and edit collected information online.

1.1 Goals of the Thesis

The main goal of this thesis is to design and implement a system to simplify data collection on the web. This system should facilitate the process of creating and maintaining a structured data repository from heterogeneous online data sources. To this end, the thesis should consist of the following steps:

- Analysis of web content aggregation techniques. Existing methods for extracting and aggregating data from the Internet should be reviewed in order to identify current best practices. A special emphasis should be placed on distinguishing the benefits and shortcomings of both RSS aggregators and the Semantic Web.

- Content aggregation engine. Based on the findings of the first step, a framework for collecting structured data from the web should be designed and implemented. This
framework should enable data collection across multiple channels including syndicated content aggregation, automated content collection, and manual curation. The framework should also utilize data warehousing techniques and best practices to ensure data integrity including data validation, duplicate detection, and revision history.

- **User interface design.** One or more appropriate user interfaces should be designed and implemented for accessing and interacting with the system. These interfaces should take into account the principles of usability and human computer interaction (HCI).

- **Reference Implementation.** A proof-of-concept prototype should be developed to highlight the functionality of the system. This application should serve as an example of a practical use case for the system.

### 1.2 Structure of the Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2** discusses background information and previous work related to this thesis. We also analyze related approaches to dealing with the issues addressed in this thesis and identify their shortcomings.

- **Chapter 3** introduces our own approach to collecting and managing information from the web. We relate our solution to previous research efforts and explain the benefits of our design.

- **Chapter 4** provides a high-level overview of the architecture of our system, including an explanation of all the major components and how they relate to one another. Additionally, we discuss some of the principal design decisions underlying the foundations of our system.

- **Chapter 5** describes the internal details of how we implemented our system.

- **Chapter 6** provides concluding remarks, lists our contributions, and identifies possible directions for future work.

- **Appendix A** explains in technical detail how to extend the system by walking through the required steps to build and install a custom extractor component.

- **Appendix B** discusses extending fuzzy matching to include approximate date comparisons and proposes a set of new similarity metrics especially designed for comparing date and datetime objects.
In this section, we review previous work related to collecting and organizing information from the web. We consider relevant research from the fields of information workspaces, web feed aggregation, information extraction, and the Semantic Web. Additionally, we examine data warehousing practices for data integration and duplicate detection.

2.1 Bookmarking

One of the earliest methods for collecting web content was saving bookmarks. Bookmarking allows a user to save the URLs of web pages of interest and is well integrated into most web browsers. Often, users are only interested in a single piece of information on a given website, but bookmarks consider a complete web page as the artifact of value. Consequently, users are forced to bookmark the entire web page containing the desired information rather than the finer level of granularity they would prefer. To later access this piece of information, the user must repeatedly access the full web page and make the effort of locating that item within the page. This problem is exacerbated when multiple items are being collected from different websites; each containing web page must be bookmarked and the user must access several full web pages in order to find the desired information items. A study by Abrams and colleagues [2] found that bookmarks fail to effectively support the retrieval of information items. Furthermore, bookmarks only record the URL of a web page and do not store any information about the structure of the data contained within that page. Therefore, bookmarks do not allow data to be manipulated or reused in another context.

In order to deal with some of the shortcomings of bookmarks, several early systems including WebBook [5], Data Mountain [15], and TopicShop [3] attempted to create information workspaces for gathering and organizing bookmarked websites. These projects focused mainly
on creating innovative interfaces for interacting with collections of web pages. While these systems offered new mechanisms for presenting and organizing saved web information, they were limited by only considering a complete web page as the smallest unit of consideration.

2.2 Web Snippets

A study by Schraefel and Zhu [16] found that users were interested in creating collections of information found within web pages. In other words, people want the ability to gather information at a finer level of granularity than a full web page. This led to the development of systems such as Hunter Gatherer [17], Internet Scrapbook [19], and myPortal [10] which allow users to save content blocks from various web pages and reassemble them into customized documents. By selecting components from within web pages, users capture only the desired information rather than the full document, which may contain additional, irrelevant material. These web page fragments, often referred to as ‘web snippets’, can then be arranged into collections to provide personalized access to the selected information. This allows information clipped from multiple websites to be consolidated and presented in a single location.

Although collecting only selected components of a web page is an improvement over saving the full document, the main shortcoming of these systems is that they still do not have any knowledge of the underlying semantics of these web page fragments. Without understanding the structure of the data, the use of the collected information is limited and sophisticated interaction with or repurposing of the data is not possible.

2.3 Web Feeds

Information collected from online data sources can quickly become out-of-date as the content may be updated frequently at its original location. Manually checking for updates on a large number of different websites is time-consuming and impractical. This issue led to the emergence of web feeds, standardized data formats for syndicating web content. Content publishers can provide a web feed for the content of a web page which allows users to subscribe to it. The web feed then notifies subscribers whenever content updates occur. The feed contains either a summary or the full content that has been published to the content distributor’s website. Feed entries contain additional metadata such as title and date of publication as well. Web feeds typically contain a fixed number of entries in reverse chronological order, representing the most recently added content to the site.

The two main web feed formats are RSS and Atom. RSS (originally RDF Site Summary, now Really Simple Syndication) is a standardized XML dialect for web content syndication that is available in a number of backward-compatible versions. The latest version of the specification is RSS 2.0.11, which was published in March 2009. Atom is a competing syndication format for web feeds that is also XML-based. Atom was developed as an alternative to RSS in an attempt to improve upon some of its perceived limitations and ambiguities. Both RSS and Atom have been
widely adopted and are available from many websites. Presence of a web feed on a web page in either format is typically represented by the unofficial web feed logo shown in Figure 2.1.

An important feature of web feeds is that they separate content from presentation so that data production is independent from data consumption. This allows innovative user interfaces to be developed on top of existing content and allows content from multiple sources to be combined in a common interface. A web feed aggregator, also called a feed reader, is an application which aggregates syndicated web content into a single location. Feed readers can be implemented as web applications such as Google Reader\(^1\) or as standalone applications like FeedDemon\(^2\).

By providing users with one place to check for updates from a large number of websites, feed readers help users to deal with the issues of data fragmentation and information overload. However, web feeds and feed readers are only a partial solution to the problem of collecting and updating structured data from the web. Feed readers are generally restricted to very specific data formats and rely on the content providers to offer a feed in the required format. The availability of web feeds is limited based on the preferences of individual websites, and the data provided within the feed might not meet the user’s needs. Furthermore, most web feed formats do not provide any means to semantically describe the content they syndicate. The metadata included in both RSS and Atom is minimal and typically limited to title, author, and date of publication. Therefore, web feeds by themselves are insufficient for the purpose of structured data collection, although web feeds and feed aggregation techniques could be useful as part of a larger information collection system.

### 2.4 Web Scraping

Web scraping, also referred to as screen scraping or information extraction, is a technique that uses software programs to extract information from within web pages so that the data may be reused in another context. Scraping emerged early in the history of the web as a mechanism for users to access and repurpose individual information items from websites. There are several possible ways that a web scraper can implement the extraction including rules based on the structure of the document, metadata attributes within the HTML, or the textual content of the web page. Since websites are dynamic in nature, much research \([11, 10]\) has focused on creating robust extraction patterns that will continue to work even after a document has been updated. Methods that use relative path within the Document Object Model (DOM) are generally more robust than those that use absolute paths. Popular methods for defining extraction rules include XPath, XQuery, and CSS selectors, while some systems utilize heuristics, ontologies, natural language processing, or machine learning techniques to identify which pieces of information to extract.

In \([7\) and \([6\), Dontcheva et al. develop an information extraction system to collect, view, and organize personal web content. Users select and label web page elements using a browser-integrated toolbar. The interface allows users to define both structural and content-based ex-

\(^1\)http://www.google.com/reader/
\(^2\)http://www.feedDemon.com/
traction rules. These user-defined extraction patterns are applied to extract individual information items semi-automatically into a local database. Gathered content is presented through pre-defined summary layout templates. In [6] the authors extend the system to allow users to define their own custom templates for personalized presentation. The system created in [7, 6] does not leverage any existing semantic markup within the HTML but instead relies on users to manually define and label each individual data attribute. Also, extracted elements are limited to a small set of predefined labels within a single schema.

2.5 Semantic Web

The Semantic Web is a movement to enrich the content of the web with additional metadata to help machines understand the meaning of the content. The goal is to transform the current Internet of unstructured documents into a ‘web of data’, thus enabling computers to have meaningful interactions with the information items described within web pages. The Semantic Web aims to provide a common framework to allow data to be shared and reused freely between websites and applications. Towards this end, the World Wide Web Consortium (W3C) has defined a number of official standards for the Semantic Web stack including Resource Description Framework (RDF), RDFa, RDF Schema (RDFS), N-Triples, Web Ontology Language (OWL), and SPARQL. These specifications are all intended to work together to provide a comprehensive set of technologies to allow for the description, storage, and querying of semantic metadata. RDF and RDFa are the most relevant formats for our research. RDF is a metadata data model for describing web resources in the form of subject-predicate-object expressions. RDFa, or RDF-in-attributes, provides a standardized mechanism for embedding RDF information into XHTML documents.

There have been a number of efforts by the Semantic Web community to address the problems of extraction and collection of structured web content. Two noteworthy projects are Thresher [8] and Piggy Bank [9], each of which will be discussed in turn.

Thresher [8] is a tool that allows end-users to extract semantic structures from HTML documents on the web. Users label examples of semantic content on a web page to teach the browser how to find similar data items. Extraction is based on tree edit distance between the DOM sub-trees of the user-provided examples. For all matching objects, RDF data wrappers are created which allow for rich interactions inside the Haystack semantic web browser. Haystack is an extensible semantic web browser built on the Eclipse platform that was developed by the Haystack research group at the MIT Laboratory for Computer Science and Artificial Intelligence. Thresher is integrated with the Haystack semantic browser and cannot be used with traditional web browsers. Also, this tool does not extract semantic data into a separate, user-controlled database, but rather automatically annotates web pages with semantic metadata as the user browses.

Piggy Bank [9] is a browser extension that allows users to collect structured content from web pages while browsing and save them in semantic web format. For websites that do not publish information in RDF format, Piggy Bank relies on user-generated JavaScript screen

\[http://groups.csail.mit.edu/haystack/\]
scrapers to extract individual information items from within web pages. The goal of this work is to empower users to build the Semantic Web from the bottom up by creating and sharing custom screen scrapers as well as by sharing collected semantic information. A web interface is also provided which enables faceted browsing of the RDF data that has been harvested.

The usefulness of the Piggy Bank tool is somewhat limited due to certain design decisions. Because Piggy Bank relies on client-side extraction, this process can only be executed when the user has the web browser application open. This design prevents the possibility of automatically updating the extracted data when the user is offline. Also, this project focuses exclusively on RDF data, which is a more complex data format than traditional relational entities. Many users might not be familiar with RDF, and therefore, it may be harder for them to utilize and repurpose their collected data.

2.5.1 Schema.org

In addition to the Semantic Web specifications defined by official standards bodies like the W3C and IETF, there have also been several alternative approaches driven by enterprise including Microformats, Open Graph protocol, and data-vocabulary.org. One commercial initiative in particular that we would like to highlight is Schema.org [1] because, as Section 4.2.1 will later explain, this was chosen as the common data model for our system.

Schema.org is a collaboration between several leading online search engine providers (Google, Microsoft Bing, and Yahoo!) to create a single unified vocabulary for structured data markup on web pages. This initiative was announced in June 2011. Schema.org outlines a comprehensive collection of schemas that encompass hundreds of different entities including commonly used types such as Person, Place, Event, Movie, Book, Product, and Organization. The vocabulary defined was inspired by earlier semantic markup efforts such as Microformats, FOAF, GoodRelations, and OpenCyc. There is also an extension mechanism that allows developers to add additional properties to existing schemas or to define their own custom entity types.

Websites can implement Schema.org using HTML5 Microdata. Microdata allows semantic metadata to be embedded into HTML documents using tag attributes. Listings 2.1 and 2.2 show an example of how to use HTML5 Microdata to mark up a sample movie listing. Schema.org provides the standardized terminology to use as the HTML Microdata attribute values, which allows for interoperability between systems due to a shared vocabulary. Schema.org markup has already been added to several major websites including eBay, IMDB, LinkedIn, and MySpace.
2.6 Data Warehousing

Data warehouses are traditionally found in an enterprise setting where they are used to integrate and aggregate data from heterogeneous sources. Information from various internal operational databases and external data sources is consolidated into a centralized repository with a single, shared schema; this provides users with a unified view over all of the data. Data warehouses are typically optimized for reporting and analysis and are used for online analytical processing (OLAP) and decision making support. Data warehouses usually also support a time dimension which associates temporal information to data items.

Our system borrows several concepts and techniques from traditional data warehousing and adapts them to a new context. Rather than building a corporate data warehouse from operational databases, we are interested in collecting and integrating personal web content from various online resources. Consequently, several fundamental aspects of data warehousing are not applicable to our system including business decision support, OLAP, and support for offline data sources. Instead, we are primarily interested in the areas of ETL processes, data integration patterns, and duplicate entity detection mechanisms.

The idea of combining data warehousing and web technologies has been explored in previous research [20]. Web warehousing relaxes the view that input sources for a data warehouse are limited to operational databases and permits data resources to be located anywhere on the web. Various projects have attempted to find innovative, new methods to incorporate existing information from the web into a data warehouse. For example, Moya et al. [12]...
have investigated how to integrate web feeds into data warehouses. They use sentiment analysis on web feeds in order to include customer opinions in a corporate data warehouse for business intelligence purposes. Meanwhile, Nebot and Berlanga [13] have looked into ways to leverage the Semantic Web within the context of data warehousing. They propose a semi-automatic method for extracting and combining data in OWL format into a multi-dimensional star schema for OLAP analysis. Both of these research projects focus narrowly on a single data format, and both are only interested in converting web data into an OLAP-friendly format that is suitable for business analysis. These limitations make each solution unsuitable for general purpose information collection and less useful outside of the enterprise context.

### 2.6.1 Data Integration

One of the primary challenges in data warehousing is data integration, which refers to the problem of how to effectively merge and reconcile data from separate, heterogeneous sources. This generally involves the process of extracting, transforming, and loading data from each source into a central repository with a common data schema. Data integration also includes the identification and merging of different information records which represent the same real-world entity.

Octopus [4] is a recent example of a system that applies data integration to the web to allow users to create new data sets from those found online. Octopus is focused on extracting structured information from HTML tables and lists. The system is also integrated with web search and attempts to infer missing values through best-effort operators. Octopus is limited by its narrow focus on extracting from only HTML tables and lists, while ignoring all other potential data sources. The system also does not provide any interfaces for managing information after it has been collected and integrated.

**Extract, Transform, Load (ETL)**

ETL is a database pattern for performing data integration that is commonly used for populating data warehouses. The ETL process is composed of three steps. First, data is extracted from its original source location. Next, the extracted data is transformed to match the structure, quality level, and expectations of the target system. The transformation stage applies a series of rules or functions to convert the incoming data into the target system’s data model and to manipulate certain values within the data item itself. Some possible transformations that can be applied to the data in this step include cleansing, validation, reformatting, aggregation, standardization, sorting, or the application of any custom-defined rules. Finally, the transformed data is loaded into the target system, which is the data warehouse.

The ETL process creates and saves a modified copy of the original data in the data warehouse. As a result, it is possible that information in the warehouse could become out of sync with the original data source over time. Whenever information is updated at its source, the saved copy of that data is no longer up-to-date. The ETL process needs to be re-executed in order to synchronize the warehouse with the original data source once again. Therefore, ETL processes are typically scheduled to run at regular intervals in order to make sure existing data stays fresh as well as to discover and load new data items.
Duplicate Detection
Another major challenge in data integration is determining if two information records from different sources refer to the same real world entity. This concept is referred to by many names including duplicate detection, entity resolution, or record linkage. The difficulty lies in reconciling the differences in the formatting, quality, and amount of information in each data record. When two textual representations do not match exactly but still denote the same entity, they are called approximate duplicates.

Information on the web is very likely to be inconsistent, incomplete, or erroneous. There are many reasons for dirty data on the web including user input errors, typos, different abbreviation schemes, and dissimilar formatting. It is therefore likely that approximate duplicates will be encountered when integrating data from the web. In order to identify approximate duplicates, fuzzy matching can be used. Fuzzy matching, or probabilistic record linkage, determines a weighted probability that two records refer to the same entity based on the values of multiple characterizing attributes of the records. Pairs with a probability above a certain threshold are deemed to be a match.

When comparing string values, there are several similarity metrics that can be used to determine the distance between the two strings. Some popular string similarity metrics include Levenshtein distance, Hamming distance, Jaro-Winkler distance, and Jaccard similarity. Each metric has different properties and performs best under specific circumstances. Some metrics are better for comparing single words while others are more suited to comparing larger sets of strings. Also, some metrics can effectively catch typos and swapping of letters within a word, whereas others can help detect rearranged words within sentences. Because of these different strengths and weaknesses, multiple string similarity metrics are often used in combination with one another.
In this chapter, we outline our approach to collecting and managing information from the web. We place our solution in the context of previous related research and explain the benefits of our design decisions.

In the previous chapter, several different approaches were presented for confronting the issue of collecting and managing information from the web. Each of the existing solutions is useful for certain scenarios, but each one on its own is limited in scope and insufficient for our needs. However, many of these diverse strategies and methods are not mutually exclusive. This thesis explores how various techniques can be combined and used together to create a more flexible and robust system for structured web data collection, integration, organization, and management. We propose a hybrid approach, illustrated visually in Figure 3.1, that combines aspects from multiple, previously unrelated areas of research. In so doing, we provide the end-user with a larger toolkit from which to select the most appropriate means for collecting structured data from any given web page.

Data published on the Internet varies widely across many dimensions including file format, markup scheme, quality, consistency, completeness, and correctness. Some websites may embed semantic metadata in one or more different formats while others may only provide...
information items as unstructured text. We do not want to make any assumptions about the type or amount of details available for the data on any given web page. By utilizing different techniques depending on the specifics of the information provided by a particular website, we are better able to cope with the diversity of the information found on the Internet.

In addition to allowing flexible data collection, we also want to incorporate various data integration techniques to ensure an acceptable level of data integrity and prevent duplicate entries from being added to the local data repository. This continues our hybrid approach by combining the previously discussed data collection methods with data integration techniques from the database and data warehousing communities. By doing so, we are able to increase the quality and reliability of the collected data. The usefulness of the harvested information, and any application that relies upon that data, is limited by the trustworthiness of the incoming data; as the popular expression says, “Garbage in, garbage out.” Our approach attempts to systematically prevent invalid data from ever entering the repository in the first place.

Figure 3.2 shows a conceptual overview of our approach to collecting and integrating data from the web. We aim to enable users to harvest information from any web page on the Internet, regardless of the specifics of how that information happens to be presented to the user. On the left of the diagram, incoming data is represented by a selection of the numerous, diverse file formats and semantic markup specifications found on the web. In addition to existing web data, another potential source of information is user-generated content. Manual data entry allows end-users to create new data items in order to capture information that
may not yet be available online from existing websites. Also, since this list of possible data formats is incomplete and likely to change over time, there is also a placeholder for unknown future data sources. By incorporating extensibility as a foundation of our approach, our solution should be suitably equipped to handle the evolution of web standards and formats over time. The center of the diagram represents the data integration process that all incoming data is subjected to before being saved into the data repository. The data integration steps are intended to prevent invalid information from entering the local database and to maintain an appropriately high level of data quality. This process includes validation, normalization, duplicate entity detection, and revision history tracking. Only after data has successfully passed through all of the data integration stages will it be saved into the local data repository, shown on the right of the diagram.

The final aspect of our approach is a universal way to review, organize, and manage the collected information. This is an important feature because it allows end-users to actually interact with the information they have gathered from various locations on the web. This provides the user with a unified view of the data no matter the original format or location of each particular record. Through this common interface, data items harvested from separate websites can be browsed and organized together in the same collections. Users are empowered to group data into custom collections or albums that are meaningful to themselves in some way. Users are also be able to incorporate feedback into the collection process through the ability to manually correct any potential automated information extraction errors. Finally, tools for keeping gathered information up-to-date and managing that information over time should be provided to the user. In this way, our solution offers a complete solution for managing web information from the initial collection process through the use and continuous upkeep of that data.

As just described, our proposed solution will combine web data collection, integration, organization, and management into a complete end-to-end system. Our system is specifically designed to support the following goals, each of which will be explained in further detail below:

- Lightweight web content collection
- Flexible extraction that leverages any existing semantic metadata
- Complete data integration support
- Tools to organize and manage collected information

**Lightweight Data Collection**

Our first design goal is to create an interface that enables non-technical users to quickly and easily gather data while browsing the web. We want to simplify the information collection process as much as possible. In order to do this, we attempt to automate whatever we can and limit the effort required by the user to the absolute minimum. In so doing, we are able to hide the underlying complexity from the user while still enabling complex interactions. The majority of the burden of data collection is shifted from the end-user to the automated system, and the user can simply guide the process by confirming or correcting the assumptions made by the best-effort predictions of the system. For more advanced and technically-inclined
users, the data collection functionality can also be extended through custom extraction rules.

In order to make the data collection procedure very lightweight, we plan to integrate the system directly with the web browser rather than implement a standalone application. Allowing the user to collect web content within the same application that she already uses to browse the web is a natural choice; otherwise an additional cognitive load would be placed upon the users by forcing them to shift their attention between the web browser and a separate application when saving information from the web. A browser extension lowers the cost of adoption and enables enhanced functionality without degrading the user’s web browsing experience.

Another way we can facilitate web data collection is by alerting the user to potential data sources as they browse the web. These notifications help to subtly remind the user about the possibility of saving information from whatever website they are currently viewing. Our goal was to provide an unobtrusive mechanism to assist users in discovering structured data on the web. In this lightweight manner, we are able to draw the user’s attention to the task of data collection whenever it is most appropriate.

Finally, we plan to use the wizard interface paradigm to guide users through the information gathering procedure. We will identify the most essential elements of the collection process and break it down into a sequence of basic steps. This step-by-step progression is very intuitive and easy for non-technical users to understand and follow. The system requires a minimum of effort from the user by automating the majority of the required work, thus making the task of web data collection both simple and efficient.

**Flexible Extraction Mechanism**

Our next design goal is to provide a mechanism that allows for structured data extraction in an adaptable fashion. A foundation of our approach is the ability to combine multiple information harvesting techniques in order to support any possible data format found on the web. Therefore, the extraction infrastructure must be constructed in flexible way that allows these diverse methods to coexist and work in harmony with one another.

Whenever a website provides its data in a structured format with semantic markup, we want to leverage that metadata when harvesting the information. This is true regardless of which particular format the website publisher has chosen to use to embed that information, whether it is part of the official Semantic Web stack or any of the competing commercial initiatives. If an RSS or Atom feed is available, we want the ability to take advantage of that syndicated content and exploit the dynamic nature of the web feed in a similar fashion to web feed aggregators. If no semantic metadata or syndicated content is supplied, the system should be able to fall back to traditional information extraction practices to harvest structured data based on the structure or content of the web page itself.

It is impossible to predetermine every possible data format on the web or which particular information extraction techniques will be most useful to a particular user for a specific task. Therefore, we rely on the principle of extensibility to support our diverse range of requirements. Extensibility enables the system to adapt and evolve over time, and it prevents
our solution from becoming obsolete whenever the latest semantic markup specifications are released, new web publishing formats are introduced, or innovative information harvesting methods are developed.

Data Integration

Another design goal of our system, which distinguishes it from several previous efforts, is the addition of data integration methodologies. The web contains copious amounts of incomplete, inconsistent, invalid, and duplicate information; this feature helps to filter incoming data in order to maintain acceptable levels of quality and consistency within the data repository. The ultimate value of an information collection system lies in the usefulness and reliability of the resulting dataset. By including support for data integration as a primary concern, our solution systematically improves data quality levels and, therefore, results in more trustworthy output.

Central to the issue of data integration is the problem of mapping attributes between data schemas. In order to define these mappings, we will use an adapted interpretation of the extract, transform, and load (ETL) pattern from data warehousing. Our version allows users to extract information from the web in whichever format is available and then convert that data into a single, shared schema. The resulting database then stores all records under a common data model in order to provide a unified view over all collected information.

Data integrity is enforced by processing all incoming information items through a mandatory validation phase. This step cleans the data by performing normalization, coercing information into the appropriate format, and ensuring certain required attributes are present. For example, a date may be represented textually as numbers or strings, and the order of date components can vary between websites as well. However, whenever a date value is saved as part of a collected data item, that attribute should always conform to the internal standard formatting style for dates.

Since the web contains several different datasets that often overlap, it is also important to be able to detect duplicate records. Information on various websites may differ slightly yet still refer to the same real world entity. These approximate duplicates exist for several reasons including differing amounts of information provided, slight differences in one or more attribute values due to user input errors (e.g. typos), or dissimilar formatting of the same data. Our solution should include entity resolution logic to automatically detect and merge these duplicates to keep our data repository normalized.

The final aspect of data integration that concerns us is the ability to track changes over time, a pattern known as change data capture. Any automatic data extraction process is prone to occasional errors, and there is a possibility that the data collection system could potentially override some existing data with invalid values. To counter balance this, we can log every update to every data record in order to provide the user with the ability to view the full revision history for each item. The user is then able to discover exactly what information was changed, when it was modified, and who made the update. Erroneous data updates can then be easily reverted back to a previous, valid state.
**Information Management**

Our final design goal is to provide tools to allow users to manage the information they have collected. We want to give users the ability to browse, organize, view, and manually edit their data. By including information management capabilities, we offer a complete solution that supports the user through the entire information lifecycle from initial discovery to continued maintenance.

One of the most useful and frequently applied information management tasks is to organize saved items into collections. There are many possible ways to organize data records including hierarchical folder structures and tagging. Tagging applies annotations to each data item and permits information to be categorized in multiple different collections at the same time. Because of this versatility, we want to allow users to organize their collected information by tagging records with custom, user-defined labels.

Another specific information management feature we feel is essential to include is the capability to manually add, edit, or delete data. This incorporates user feedback into the collection process and enables users to correct any errors in the gathered data due to mistakes in the original source website or inconsistencies that may have been introduced by the ETL procedure. Additionally, allowing users to manually create new data items makes it possible to save information that was discovered offline. In other words, this removes the limitation that all data must be found on the web and instead permits users to incorporate data found in other contexts. The combination of automatic and manual data collection techniques continues our hybrid approach theme and makes our platform even more powerful and versatile.

Since this project is focused specifically on data collection from the web, it is a logical choice to provide a web interface for interacting with that data. This way users never have to leave the comfort of their web browser and switch to another application, but rather they are able to view and manipulate their gathered information directly within the familiar environment of their browser.

We want to provide a default web interface for interacting with the collected data so that our solution would provide immediate value to all users and allow non-technical users to make use of the system. However, it is important to note that our interface is just one possible way to interact with the collected data. Once the information has been harvested from the web and saved into the local data repository, it is under the complete control of the end-user. Technically-inclined users can develop their own custom interfaces on top of this collected data repository in order to meet their own specific needs and requirements.
In this chapter, we provide a high-level overview of the architecture of the Sift system, including an explanation of all the major components and how they relate to one another. Additionally, we discuss some of the principal design decisions underlying the foundations of our system.

4.1 System Architecture

This section provides a high-level overview of the architecture of the system we have developed in this thesis, which is illustrated in Figure 4.1. The core of the Sift system is a two-tier Python-based web application that is composed of three primary components. Two of these components, the browser extension and web interface, are user interfaces which interact through web services with the third component, a backend server. Client-server communication between the tiers of the system is implemented via a REST interface over HTTP. In the case of the browser extension, these requests are all made asynchronously via AJAX calls and return JSON objects, while the web interface uses traditional HTTP GET and POST requests to serve HTML documents.

Neither the browser extension nor the web interface communicates directly with the data repository, but rather all database operations are mediated by the web server. This enables the server to manage all data updates in order to ensure proper data integration procedures are executed and all changes are appropriately tracked within the revision history. This architecture also allows for the possibility of including a caching layer within the web server to improve performance; the results of expensive and frequently executed queries could be cached by the server and then served directly without requiring database operations to be re-executed.

The purpose of each component, as well as the flow of data and communication between components, will be explained in turn.
4.1. SYSTEM ARCHITECTURE

Browser Extension
- Discover
- Collect

Web Interface
- Browse
- Organize
- View
- Create
- Edit
- Delete

Server
- Extract
  - HTML5 Microdata
  - RDFa
  - Open Graph
  - RSS
  - Microformats
- Integrate
  - Validation
  - Transformation
  - De-Duplication
  - Revision History
- Store
  - Data Repository
- Update

Figure 4.1: System architecture overview

Browser Extension
The data collection process begins with the browser extension component, which is implemented as an add-on to the Mozilla Firefox web browser. The extension is written in a combination of XUL and JavaScript. As the user browses the web, the extension helps the user to discover potential data sources. It uses client-side detection of selected structured data types to identify candidate sources and, if found, notifies the user through its toolbar button icon. The user can then choose to collect individual data items from the web page. Through a custom interface within the browser, the user can specify which pieces of information to save. This interface communicates with the web server component via AJAX, and the server processes the webpage in the background.

Server
The web server component is responsible for extracting, integrating, and storing the harvested data. The extraction process is initiated by the browser extension component and uses a multi-pronged approach to extract structured data in any available format. The data integration phase forces the extracted items through a series of steps to clean and conform the data. These steps include transformation, validation, duplicate entity resolution, and tracking of changes. Finally, the extracted and cleaned information is saved in the local data
repository. This user-controlled database stores actual copies of all data items rather than just links to the original data sources. Additionally, the server also contains logic to automatically update saved data by re-executing the extraction and integration process at regular intervals.

Web Interface
After the user has collected some information from the web using the browser extension and that data has been saved to the local database by the web server, the user can then use the web interface to interact with the data. First, the user can organize saved records into custom collections. Next, the user can browse these collections of items in a unified way, regardless of the data type or original source location of the information. The user can also view the details of individual data records as well as edit or delete any existing database entry. Finally, the user also has the ability to manually create new data items directly from the web interface. Before being saved to the local data repository, all information added through the web interface component is subjected to the exact same data integration steps on the server as information items that have been extracted from web pages.

4.2 Design Decisions
This section highlights a few of the important choices made regarding the design of our system as well as the rationale behind each decision.

4.2.1 Data Model
We chose to define a single, common data model to be used for the internal representation of all collected data items in our system. This approach offers less flexibility than if users were allowed to define their own custom data models. However, this tradeoff was chosen because of the potential benefits of sharing a single data model for everyone. The first benefit is that the time required to get started with the system is drastically reduced. Users will see a much faster benefit from this plug and play approach where no data modeling is required. Also, users do not need to worry about designing any custom templates since the system already provides interfaces that are optimized for each of the pre-defined data types. Another major benefit of using a common data model is the potential for sharing of extraction rules between installations of the system. This enables the possibility of creating crowdsourced repositories for data sources and their related data mapping configurations. Finally, since the data schemas are reused by all instances of the system, collected data items themselves could be shared as well. Therefore, communal data repositories could be created to allow groups of people to contribute the information that they have each extracted individually. Through this or other methods, users could exchange structured data and mutually benefit from the work of others in the community.

For the common data model in our system, we have chosen to use Schema.org. As described earlier in Section 2.5.1, Schema.org provides a vocabulary and collection of schemas for defining a vast number of entity types. Schema.org was selected because it is comprehensive, designed for the web, and based on the existing work of several previous semantic markup initiatives. Each of these advantages will be further discussed in turn:
Comprehensive. Although it is not intended to be a global ontology of the world, Schema.org does provide schemas to define hundreds of common data types. These include the objects which are most frequently described in web documents and which are most relevant to the users of our system such as Event, Person, Place, and Product. There is also an extension mechanism provided by Schema.org in order to add attributes to existing schemas or to define new schemas for unsupported data types.

Designed for the Web. The Schema.org initiative was led by leading search engine providers and was created specifically to describe web documents. Since our system is interested in extracting information from the web, this matches our needs and requirements quite well. As an added benefit, many websites have already implemented Schema.org metadata on their web pages. For these sites, the extraction process is simplified because both the web page and the system share the exact same schema definitions.

Distinguished Heritage. By reusing an existing data vocabulary rather than trying to construct our own from scratch, we are able to profit from the accumulated knowledge and expertise of others over a long period of time. There is no need to reinvent the wheel when we can instead stand on the shoulders of giants. Schema.org was conceived by a number of intelligent people from Google, Microsoft Bing, and Yahoo!, and it builds on the work of several previous semantic markup initiatives. The data model for Schema.org was derived from RDF Schema and the vocabulary was inspired by several preceding formats including Microformats, FOAF, GoodRelations, and OpenCyc. In this way, Schema.org inherits from the past and represents the latest culmination of semantic metadata best practices.

4.2.2 Database

Another major design decision was the choice of database management system. Rather than selecting a traditional SQL database such as MySQL or PostgreSQL, we have decided to use MongoDB \(^1\), a schema-less, document-oriented database. MongoDB stores JSON-style documents with dynamic schemas and was designed to be agile and scalable. This unconventional choice was driven by several key advantages provided by MongoDB compared to SQL alternatives.

Schema Evolution. MongoDB is schema free and does not impose any constraints on the structure of the documents it stores. This allows developers to dynamically change schemas at any point without needing any updates to the database structure or to existing data saved in the database. In contrast, SQL databases explicitly define the structure of each table; in order to make changes to an SQL database, the developer must modify the table schema which updates every existing row of data stored in that table. Schema updates on SQL databases are an expensive operation, particularly if there is a large amount of existing data that has to be updated. Therefore, MongoDB is a better choice for agile development with frequent schema changes and for schema evolution on fully populated databases.

Our system does define data schemas in software, which may seem to offset the benefits of using a schema-less database. However, this method allows us to ensure data integrity when

\(^1\)http://www.mongodb.org/
it is needed while allowing the use of unstructured fields within our schemas as well. This
technique also facilitates schema modifications because changes can be made in the code
without requiring any updates to the database.

*Disk Space.* A key difference between MongoDB and SQL databases is which information is
stored on disk. Typical SQL databases are row-oriented and require space to be allocated to
store every column defined in the schema for every row, even if a row is sparsely populated.
On the other hand, MongoDB is document-oriented and only stores key-value pairs for
attributes that hold values. This approach should therefore lead to significant disk space
savings when dealing with sparsely populated databases.

The schemas used in our system each define a large number of attributes but we expect only
a small subset of these properties to contain a value for any given object. Therefore, using
MongoDB is more efficient for our use case because any field without a value does not need
to be written to disk. The actual amount of disk space saved may vary depending on the
efficiency of the internal implementation of MongoDB.

*Performance.* High performance is a primary concern for web applications in order to ensure
responsiveness. Since our system was built to power web applications, speed and scalability
were important design considerations. MongoDB meets these requirements because it is a
web-scale database that is optimized for high performance. MongoDB’s speed is at least
partially due to avoiding the use of SQL and table joins.

*Security.* One of the main threats to web application security is SQL injection. However, by
choosing a non-SQL database, there is no risk of an SQL injection attack on our system. In
this way, MongoDB increases the security of the system by avoiding all SQL-based security
threats. It should however be noted that MongoDB is a relatively immature technology
that may have security vulnerabilities of its own, so this benefit is somewhat offset by the
 possibility of unknown future exploits.
In this chapter, we present the implementation details of the Sift content aggregation engine. The system is composed of three major components: a browser extension, a web interface, and the content aggregation engine. The browser extension provides an interface for content discovery and extraction configuration. The backend infrastructure for the content aggregation engine handles the automated extraction process including data integration. Finally, the web interface allows users to view, organize, and manipulate the collected data items online. The details of the implementation of each of these components will be discussed in turn.

5.1 Firefox Extension

The first component of our system is an interface for discovering and extracting interesting data while browsing the web. We have chosen to implement this interface as a custom extension for the Firefox browser. By integrating our tool with the web browser, we offer a seamless experience that allows a user to collect information directly from a web page without having to leave the browser. This lightweight solution does not require any context switch to change between applications, but rather it unobtrusively enhances the web browsing experience for the user.

Developing for the Firefox browser in particular was chosen because of its popularity as well as the high degree of extensibility offered by the Mozilla platform. Additionally, there are a large number of resources and documentation available for developing Firefox extensions. We initially attempted to develop the extension using the new Firefox Add-on SDK. However, this new SDK does not yet support the full functionality required by our extension; in particular, the Add-on SDK lacks the ability to create overlay panels in the browser. Therefore, we implemented a traditional XUL extension using a combination of JavaScript and XUL (XML User Interface Language).

The main functionality of our extension is to provide notifications to the user about poten-
tial data sources and to allow the user to configure web pages as data sources using a wizard paradigm. To achieve this, the extension integrates with several components of the web browser and creates new components as well. The context menu of the web page has been augmented with an additional entry, as illustrated by Figure 5.2, which opens the extraction wizard when selected. Our extension also includes a simple preferences dialog, as shown in Figure 5.3, which allows users to set the location of the backend server for their installation. Finally, Figure 5.1 demonstrates our custom toolbar button which toggles the extraction wizard panel when clicked. The icon of the toolbar button is also used for user notifications, which will be explained in the next section.

5.1.1 Notifications

Our Firefox extension provides a mechanism for alerting the user to web pages that might be good candidates for data extraction. When certain forms of structured, semantic metadata are detected on the current web page, the icon of the extension’s toolbar button is updated to inform the user of this fact. As shown in Figure 5.1, the normal icon is a dull grey funnel, but when semantic metadata has been detected, the icon is changed to a bright green funnel with a red plus sign overlay. By changing the color and shape of the icon, we are able to subtly and unobtrusively notify the user of potential data sources. This technique conveys the appropriate information to the user without distracting from the web browsing experience. Other notification techniques, such as pop-up alerts, would be much more noticeable but would also frequently disrupt and annoy the user. On the other hand, our subtle icon notification method promotes awareness without user interference. Also, the use of toolbar icons to display additional information conforms to a commonly used design pattern for Firefox extensions and should be familiar to most users of the Firefox browser.

In order to implement these notifications, we had to create an alternative method to detect relevant metadata on the client-side. The normal data extraction process is executed on the server through AJAX interactions, but this process is too slow and unresponsive for these lightweight notifications. The notification icon needs to be updated every time the user visits a new web page, which occurs very frequently in normal, everyday web browsing. Therefore, we chose to move the detection of semantic metadata from the server to the client to improve performance.

Client-side semantic metadata detection is implemented in JavaScript and only supports a limited subset of data formats which are easy to process in real-time. This solution is not as comprehensive as the full server-side extraction, but rather serves to highlight a selection of web pages that contain structured metadata. The user can always click on the toolbar button to start the extraction wizard, regardless of whether or not the notification icon is currently displayed.

Every time a web page is loaded by the browser, our extension checks for the presence of Microformats, HTML5 Microdata, or Open Graph markup. The Firefox browser provides a Microformats API for detecting and accessing common Microformats from within browser extensions. Open Graph protocol and HTML5 Microdata are detected by using JQuery\(^1\) to

---

\(^1\)http://jquery.com/
CHAPTER 5. IMPLEMENTATION

Figure 5.1: Firefox toolbar button and notification icon

Figure 5.2: Firefox context menu integration

Figure 5.3: Firefox extension preferences dialog
search for signature attributes within the HTML document. Notification is triggered if the document contains any of these embedded metadata formats.

5.1.2 Extraction Wizard

Our browser extension also includes a special interface for configuring new data sources. We utilize the software wizard paradigm to guide the user through the process in an easy to understand, step-by-step manner. The wizard is implemented as an XUL overlay panel attached to the bottom of the web browser which splits the viewing area of the browser horizontally. The height of this panel is adjustable and never overlaps the content of the web page itself. The extraction wizard can be opened through the context menu or by clicking on the toolbar button. Subsequent clicks on the toolbar button will toggle the visibility of the wizard panel.

The wizard interface consists of three steps. At any point, the user can navigate between steps either by clicking directly on the tabs or by using the Next/Back buttons at the bottom of the screen. The details for each step in the process are explained in order:

• **Step 1: Extract Metadata** - When the extraction wizard is opened, it triggers an AJAX request to the Sift server. The server then attempts to extract as much metadata as possible from the current web page by executing every known extractor component. The details and implementation of extractor components will be described later in Section 5.2.1. All extracted information is returned to the browser extension and is displayed in the first panel of the wizard. This data is presented in a hierarchical tree structure organized by extractor that can be expanded and collapsed. This interface, which can be seen in Figure 5.4, allows the user to easily browse and view all the available metadata.

• **Step 2: Map to Schema** - The second panel of the extraction wizard, shown in Figure 5.5, allows the user to map extracted metadata to fields in one of the Schema.org data schemas. The interface was designed for simplicity to make the task as straightforward and quick as possible. To assist the user, the system automatically guesses and pre-selects the most appropriate data model for the chosen web page. It also pre-assigns as many default mapping rules as possible. The goal is to allow the system to do as much work as possible up front in order to minimize the effort required of the user.

During the AJAX request when the wizard is initialized, the server tries to execute a set of default mapping definitions to determine which are applicable to the current data source. These default settings are defined in a YAML configuration file which can be edited by developers. This allows additional default mappings to be specified as well as allowing changes to the preference order of existing options. All default options that match the current data source are returned to the extraction wizard in the AJAX response. The extraction wizard uses these defaults to determine which schema to pre-select. The schema with the highest number of fields defined by default options is deemed to be the best match.
Figure 5.4: First step of Firefox extension extraction wizard

Figure 5.5: Second step of Firefox extension extraction wizard
If the pre-selected values are incorrect or incomplete, the user can make adjustments in this second step of the wizard. The user has the ability to change both which schema to map to and which mapping definition to use for each field of the schema. To change a mapping for a given field, an alternative can be selected from the dropdown menu associated to that field. If none of the default options are suitable, the user can define a custom mapping. The custom mapping option presents the user with a hierarchical tree interface very similar to that of the first step of the wizard. However, this view allows the user to select any node in the tree as the custom mapping target.

- **Step 3: Review and Save** - The final step provides the user with an overview of all the schema mapping definitions that have been defined. This allows the user to review everything in an easy to read format before saving. The user must then click on the save button to complete the process. The save button triggers another AJAX request to the server which saves both the URL of the data source and the user-defined extraction rules. After the data has been successfully saved on the server, a confirmation will be displayed to the user.

### 5.2 Content Aggregation Engine

The backend infrastructure for the content aggregation engine contains the majority of the logic of the system. This component executes the entire Extract, Transform, and Load (ETL) procedure and is responsible for applying several data warehousing techniques to perform
data integration and ensure data integrity. This section will explain how data enters the sys-
tem and the steps required to process all incoming data including extraction, transformation,
validation, duplicate detection, and revision history tracking. Finally, this section will also
describe the manner in which the system keeps all collected information up to date automat-
ically.

5.2.1 Data Extraction

Extracting structured metadata from web pages is part of the core functionality of our system.
In order to support this task, we have implemented an extensible extraction infrastructure
which is responsible for the full Extract, Transform, and Load (ETL) process. This section
will cover the first step in the ETL process: data extraction.

The data extraction step consists of running a number of extractor components against a
given source and then passing the combined output of these extractors to the next step in the
ETL process. A Sift extractor is a special component which takes a webpage as an input,
applies its unique extraction logic, and then outputs whatever metadata it was able to extract.
The system includes a number of predefined extractors for several common web data formats
and semantic metadata formats. It is also possible for developers to extend the system by
creating their own custom extractor components, which will be explained in the Extensibility
section.

The data extraction process begins by determining which extractors are required for the
current source URL, which depends upon the data mapping that has been defined for that
source. The system then runs each of these extractors in succession. If an extractor is not
applicable to the current source, such as trying to extract RSS metadata from an iCalendar
file, then the extraction will fail silently and no output will be produced for that extractor.
Each extractor which successfully completes its extraction adds its output to the shared
output hash table with its own unique key.

The data extraction process is optimized to make sure that the source web page is only
downloaded a single time. The HTTP response from this initial request is cached locally
and reused by each extractor. This helps performance by preventing duplicate work and
avoiding unnecessary network requests. The current implementation executes each extractor
sequentially. However, this could be further optimized in future versions by allowing
multiple extractors to run in parallel.

Our system implementation includes a default set of nine extractors which support a wide
variety of data formats. A brief overview of each of these extractors follows:

• HTML - The HTML extractor is intended to be used as a base class for creating any
extraction that interacts directly with the HTML markup. This convention allows the
HTML to be parsed only once and reused by all subsequent extractors that require
access to this data.

• HTML 5 Microdata - HTML 5 introduced a mechanism, called Microdata, to al-
allow machine-readable semantic metadata to be embedded within HTML documents.
5.2. CONTENT AGGREGATION ENGINE

HTML elements can be annotated with name-value pairs from a Microdata vocabulary such as data-vocabulary.org or Schema.org. This extractor uses the open-source microdata\(^2\) utility library to extract HTML5 Microdata from the underlying HTML. The output from this initial extraction is then reorganized in order to be grouped according to data type. Also, all relative URLs in the extracted data are automatically converted to absolute URLs.

- **Microformats** - Microformats\(^3\) are a community-driven effort to create a set of open data format standards for annotating HTML and XHTML documents with machine-readable semantic metadata. This extractor utilizes the open-source Microtron\(^4\) package to extract all Microformats metadata embedded within the given HTML. It supports the majority of current Microformats specifications and drafts including hCalendar, hCard, hAtom, hNews, hReview, hListing, hAudio, hProduct, hResume, adr, geo, Votelinks, xFolk, XFN, rel-tag, rel-license, and rel-principles. Note that this extractor intentionally excludes the no-follow specification as this is commonly used but not useful for our use case of extracting data objects. The Microformats extractor also converts all relative URLs to absolute URLs automatically.

- **Open Graph** - The Open Graph protocol\(^5\) is a simplistic semantic markup initiative created by Facebook in 2010. The Open Graph protocol allows web pages to become objects in the social graph by embedding basic metadata in HTTP \(<\text{meta}>\) tags. This extractor parses all Open Graph information from the given web page using the open-source PyOpenGraph\(^6\) library. Since the Open Graph protocol is implemented with HTTP \(<\text{meta}>\) tags, this metadata is only capable of describing a single entity and cannot be used to describe multiple objects on the same web page. Therefore, this extractor never outputs more than one set of metadata attributes for any web page.

- **RDFa** - RDFa, as mentioned in Section 2.5, is a W3C recommendation for adding semantic metadata to XHTML documents. Specifically, RDFa enables RDF subject-predicate-object triples to be embedded within XHTML markup. This specification provides a set of XHTML attributes to augment human-visible text with machine-readable hints without repeating content. The RDFa extractor employs the RdfaDict\(^7\) package to parse and extract all RDFa annotations within the given document.

- **Web Feeds** - This extractor uses the open-source Universal Feed Parser\(^8\) module for parsing syndicated web feeds. It can handle several different data formats including RSS 0.9x, RSS 1.0, RSS 2.0, CDF, Atom 0.3, and Atom 1.0, as well as several popular extension modules such as Dublin Core and Apple’s iTunes extensions.

- **iCalendar** - This component extracts calendar and event metadata from Internet Calendar (iCal) files with help from the open-source Python icalendar\(^9\) package. It also

\(^2\)https://github.com/edsu/microdata  
\(^3\)http://microformats.org/  
\(^4\)https://github.com/amccollum/microtron  
\(^5\)http://ogp.me/  
\(^6\)http://pypi.python.org/pypi/PyOpenGraph  
\(^7\)http://pypi.python.org/pypi/rdfadict  
\(^8\)http://code.google.com/p/feedparser/  
\(^9\)http://pypi.python.org/pypi/icalendar/
ensures that all extracted dates are appropriately converted into \texttt{datetime} objects in order to facilitate further processing by guaranteeing a consistent data format.

- \textit{JSON} - This extractor attempts to read the input data as a JavaScript object. If successful, the entire JSON object is added to the output and made available for data mapping.

- \textit{Content Type} - This extractor returns the Internet media type (or MIME type) of the webpage. It first checks if the Content-Type HTTP header field is set, then tries to guess the MIME type from the filename, and finally resorts to checking for the presence of a magic number file signature. The first valid value encountered is returned.

\section*{Extensibility}

The data extraction process was designed to be extensible to allow developers to extend the functionality of the system. Custom extractor components can be created in order to support the unique requirements of each user. Extensibility enables the system to evolve and keep pace with the emergence of new data formats and semantic markup standards over time.

Developing a custom extractor is designed to be quick and painless for developers by minimizing the amount of work necessary. Extractors are Python modules that extend one of the base classes provided by the Sift framework and implement a single required method. The technical details of how to develop and install a custom extractor component will be described in Appendix A, and an example of a fully implemented extractor component can be seen in Listing A.5.

Custom extractors are not limited to any specific techniques or methodologies but rather have the freedom to process data sources any way they choose. This means that, for example, one extractor can use XPath expressions, while another utilizes semantic markup, while yet another relies on natural language processing. All of these techniques can co-exist because each extractor is completely independent of all others. Also, once a custom extractor is installed, it is treated equally with the default extractor components.

\subsection*{5.2.2 Data Transformation}

The second step in the ETL process is data transformation. Data transformation consists of mapping each data element from its original location in the source system to the appropriate position within the destination system. In our case, this involves mapping values from the extraction output to fields in one of the internally defined data models. These data mappings can be defined using the Firefox Browser extension that was previously described in Section 5.1.

The data transformation step also includes the conversion of data from its original format to the destination system's expected or preferred data format. Because our system deals primarily with data extracted from webpages, most if not all incoming data is initially formatted as a \texttt{string}. However, in our local database we would prefer to store this data in the most appropriate representation, such as a \texttt{datetime} object for dates and \texttt{int} for integer numerical values. Even for string values, we would like to ensure a consistent, universal encoding scheme for all strings stored in our database; for this reason, all strings
are automatically converted into Unicode encoding. These conversions are made possible by the schema definitions which tell the system which data type to expect for each field. The system can then attempt to coerce the extracted value into the expected format. The method used to define the schema definitions will be further explained in the Data Validation section.

**Format of Data Mapping Definitions**

When a new data mapping is defined through the Firefox browser extension, it is stored internally in a separate MongoDB collection in a custom format. The Firefox extension sends the mapping definition to the server as a JSON object, which is then saved directly into the database. This takes advantage of the flexibility and schema-less nature of MongoDB which allows the storage of data objects of arbitrary sizes and structures. When the system accesses this mapping definition from the database, it is converted into a Python multi-dimensional dictionary (i.e. hash table) which can then be processed by the server.

The custom format for data mapping definitions, as a slightly simplified explanation, consists of a two-dimensional hash table. The keys of the hash table define the target schema and attribute, while the value defines the mapping definition for that particular attribute. A sample data mapping definition is illustrated in Figure 5.7, which shows a mapping definition for the ‘genre’ attribute of the ‘Movie’ data type.

![Figure 5.7: Sample data mapping definition](image)

The first level of the hash table contains one entry for each target data type, such as ‘Movie’ or ‘Event’. Currently, the Firefox extension is only able to configure a data mapping for a single data type, but this data format is capable of storing mappings for multiple data types from a single source. The second level of the dictionary contains one entry for each field in the target which has a data mapping defined. If no mapping has been defined for a given field, that field will be excluded from the hash table. Finally, the mapping definition is made up of a list of string values. The first item in the mapping definition list is the “extractor key”. The extractor key uniquely identifies which extractor to use for this attribute. The remainder of the mapping definition consists of a list of string values which comprise the mapping path. The mapping path defines the traversal path through the output of the specified extractor to reach the desired value or values.

**Entity Identification**

A single data source can contain information about one or possibly multiple entities of the same data type. For example, one webpage might describe a single event, while another webpage might list several related events. This ambiguity presents a challenge both in
determining the number of entities that should be extracted from a given source and how to identify the boundaries of each object. This is not a trivial task due to the flexible nature of our mapping definitions, which can allow mappings from multiple different extractors to a single entity. Additionally, if multiple items exist on a single webpage, the amount of semantic metadata associated with each object may vary greatly. For example, one event might be missing an end date while another might have an end date set but no location. This adds further complexity to the scenario because the system must be able to resolve each entity from a combination of multiple sources which each may contain a different amount of metadata for each item.

In order to overcome this challenge, our system uses a heuristic to identify separate entities and ensure that the appropriate information is matched to the appropriate entity. We are able to do this by imposing a design requirement on all extractor components. Each extractor must conform to a standard method of defining the separation of entities in its output. As long as the extractor meets this requirement, the rest of its output is not limited in any way. By utilizing this constraint on extractor output, our system is able to clearly identify the metadata for each separate entity and is then able to assemble the resulting object appropriately.

5.2.3 Data Validation

After the data has been extracted from its source and transformed into the local data model, the framework must perform some additional validation checks before saving the data to ensure an acceptable level of data integrity. In traditional SQL databases, these data checks are typically enforced at the database level through the use of schemas and constraints. However, since our framework is implemented with a schema-less database, MongoDB, all data validation must be defined and performed at the software level.

In order to define our document schema and validation constraints in code, we have chosen to use the open-source MongoEngine\(^{10}\) package. MongoEngine is an Object-Document Mapper for working with MongoDB from Python; this is conceptually similar to an Object-Relational Mapper (ORM) but for document databases instead of traditional relational databases. MongoEngine defines field types for several data types including strings, URLs, email addresses, dates, numbers, booleans. Also, MongoEngine enables the developer to define additional validation checks such as setting a maximum length for strings or identifying mandatory fields.

We have defined document schemas for each of the implemented data types from Schema.org. Due to the wide variability in data available from different online sources, only a single field for each schema is classified as mandatory: the 'name' field. This also helps to minimize user effort when manually entering data and enables quick, lightweight data entry. Also of note is that each document schema is independent and does not contain references to any other documents. The database is intentionally de-normalized in order to provide superior performance.

If incoming data fails the validation step, the data is not saved into the database and an error is

\(^{10}\)http://mongoengine.org/
logged. If this document was being added manually through the web interface, the form will be redisplayed to the user highlighting the validation errors. The user then has the opportunity to correct the given errors and resubmit the data.

5.2.4 Duplicate Detection

After the data validation step has successfully completed, the framework then checks for duplicate entities. Duplicate detection is another data integrity step which attempts to ensure that each real-world object is stored only once in the database. In order to support duplicate detection within our framework, we have defined a set of duplicate identification fields for each of the implemented Schema.org object types. This set of fields acts as a composite primary key that can uniquely identify an entity occurrence. For example, an Event object can be uniquely defined by the combination of its name, start date, end date, and location.

The duplicate detection process consists of two stages. The first stage checks for the presence of exact duplicates. The candidate object (i.e. the data object to be added) is considered to be an exact duplicate if and only if there exists an object in the database which contains the exact same values for each of the candidate object’s duplicate identification fields. Note that a match will not be found if the candidate object contains values for any of the duplicate identification fields in which the existing object has no value. Therefore, the existing object must contain at least the same level of detail as the candidate object to be considered an exact match. This design choice helps to prevent false positive matches. Furthermore, if an exact match is not found, it is still possible to be flagged as a near duplicate in the second stage of duplicate detection.

The second stage of duplicate detection utilizes fuzzy matching techniques to check for the presence of near duplicates. As with the exact duplicate check, only the set of duplicate identification fields are considered and all other fields are ignored. The near duplicate check calculates a weighted score of the similarity of each duplicate identification field. If the resulting score is above the system-defined threshold, then the candidate object is considered to be a duplicate of the existing object.

Similarity comparisons for each field are type dependent and each return a value in the range [0,100]. The only field types that currently support fuzzy matching are strings and dates; all other field types return a score of 100 for a perfect match or 0 otherwise. Also, if either of the objects being compared is missing a value for a given field, the similarity score for that field is 0. We will now describe the similarity scoring techniques used for strings and dates in further detail:

**Approximate String Matching.** As discussed in subsection 2.6.1, there are many different methods for approximate string matching, also known as fuzzy matching. In this project, we use the open-source FuzzyWuzzy\footnote{https://github.com/seatgeek/fuzzywuzzy} package for fuzzy string matching. This package supports multiple approximate matching measures including edit distance, partial string similarity, sorted token ratio, and token set ratio. The edit distance calculation is a wrapper around Python difflib’s SequenceMatcher.ratio() method, and it is useful for
comparing strings of similar length and for identifying typos or spelling errors. Partial string similarity uses a heuristic to compare substrings when two strings are of different lengths. The sorted token ratio is useful for finding strings which contain the same words in different orders. Finally, the token set ratio is a more flexible measure that tokenizes each string and then compares the intersection and the remainder. For each string similarity comparison performed by our framework, all four of these similarity measurements are calculated and the maximum value is returned.

**Approximate Date Matching.** Unlike fuzzy string matching, there has been little or no previous research in approximate date matching. Therefore, we have developed our own similarity metrics for fuzzy date comparisons, which are presented in further detail in Appendix B.

Our framework implements one of our proposed date similarity metrics, the fractional time period, and uses one year as the time period for comparison. This measurement calculates the difference between the dates in days and then divides this amount by the number of days in a year. The output of this calculation is converted from a continuous number in the range $[0,1]$ to a discrete integer in the range $[0,100]$. If the two dates are exactly the same, then a score of 100 will be returned; if the two dates are 365 days apart or more, then a score of 0 will be returned. This metric is effective at finding dates that occur in a similar temporal space even if they occur in different months or years.

### 5.2.5 Revision History

After incoming data has been validated, checked for duplicates, and saved to the database, the final step is to store the revision history for the object. By automatically tracking all changes to every data object, the Sift system provides a built-in audit trail and enables the tracing of data lineage. This allows users to understand exactly where each piece of data originated, and also enables users to roll-back any incorrect or erroneous updates without difficulty.

We implemented revision history tracking through the use of broadcast events and listeners. MongoEngine supports a limited set of broadcast ‘signals’ which are triggered before and after certain events including initialization, save, and delete. We defined custom listeners for both the post_save and post_delete signals. When either of these listeners is called, the newly stored data updates the revision history collection.

All revision history is stored in a separate collection$^{12}$ in the database rather than embedding history information within the object itself. Since revision history is not regularly queried except in special cases, keeping all history in a separate collection ensures that querying performance for the most common use case is not affected as the number of revisions increases. Also, the full object details are stored for each revision. By storing all fields for every revision, a significant amount of duplicate data will be saved within the history collection; however, this design decision was a conscious tradeoff between disk space and performance. This approach was chosen because disk space is relatively cheap and high-performance is a priority.

---

$^{12}$A ‘collection’ in MongoDB is roughly equivalent to an SQL table.
Automatic Updates

Data on the web is not static; rather, it is constantly evolving with frequent edits, additions, and deletions. Our system was built with this principle in mind and therefore supports the evolution of data over time. All data sources can be periodically revisited to check for new content or updates to existing content. Through this automatic maintenance, collected data is kept fresh and up-to-date over time.

The scheduler process handles these automated data updates. The architecture for the update process is illustrated in Figure 5.8. When this process is invoked, it first queries the database for the full list of available sources. A new task for each source is then added to a queue, and multiple worker threads are spawned. These worker threads execute in parallel as they work on each item in the queue. For each task, the worker thread will perform the applicable data extraction followed by the data integration steps, which were described earlier in this chapter. After the data has been extracted and cleaned, it will be saved to the database. In order to fully automate this procedure, the user must create a cron job on the server to re-execute the scheduler process at a fixed time interval.

5.3 Web Interface

The final component of our system is a web interface for browsing, organizing, viewing, editing, and managing collected data items. Since the system is interested in gathering data from the web, implementing a web interface for interacting with the saved information online was a natural choice. This allows users to collect and then also manipulate data items without
ever leaving the web browser. Choosing a web-based application instead of a stand-alone desktop application has the advantage that users of the system are already experienced in web browsing and interacting with web applications so the environment will be familiar and intuitive. Also, the web interface allows world-wide access to the data repository, rather than being limited to local access on a single computer. A final benefit of enabling a web interface to the collected data is the potential for collaborative editing; data entry, maintenance, and organization could be crowdsourced to ensure that the repository is always complete and up-to-date.

The web interface is implemented as a reusable application for the popular Python Django web framework\textsuperscript{13}. This application is capable of functioning as the primary user interface for the collected data, but it can also be used solely for data administration and allow other applications and interfaces to be built on top of the data repository. The design of the interface was built with the help of the Bootstrap toolkit\textsuperscript{14} from Twitter, which includes a CSS framework as well as a number of JavaScript plug-ins.

The web interface was designed to support a number of common tasks including browsing, organizing, viewing, editing, creating, deleting, and managing revision history of items in the data repository. Each of these tasks will be outlined in further detail:

5.3.1 Organize

The Sift system allows users to define their own ‘topics’ and then categorize all data items according to these user-defined topics. A topic is a keyword or phrase that is used to classify a set of items that are related in some way that is meaningful to the user; in some systems, this concept is referred to as a ‘tag’. Each data item can be added to multiple topics, which allows for the existence of several different organizational schemes concurrently. For example, an event could be classified both by location and subject area by tagging it under those two topics. This allows users to annotate data items with several classification keywords so that they have a better chance of later finding this information when they are trying to find it. Also, unlike some other tagging schemes in which tag names are case-sensitive or limited to a single word, our system imposes no restrictions on topic naming.

Topics can be added through the web interface when editing a data item. Our goal was to provide a lightweight method to allow users to both create and assign topics in order to encourage the use of this feature. This functionality is implemented in the edit form as an HTML multi-select box, but we apply the Chosen JavaScript plugin\textsuperscript{15} to transform this into a more user-friendly text input with drop-down suggestions, as shown in Figure 5.9. As the user starts typing in this text input field, a list of matching choices is presented which is dynamically filtered with each keystroke. A suggestion can be selected by pressing the Enter key or with a mouse click. Selected items can easily be removed by clicking on the x next to the tag to be removed. New topics can also be created inline using the button to the right of

\textsuperscript{13}http://www.djangoproject.com/
\textsuperscript{14}http://twitter.github.com/bootstrap/
\textsuperscript{15}http://harvesthq.github.com/chosen/
the text input. New topics are dynamically added to the `<select>` tag and displayed in the text input instantly.

5.3.2 Browse

The web interface offers support for browsing the data repository either by data type or by topic collection. In the first case, the presentation is more straightforward because all items are of a uniform type. By clicking the Types link in the top navigation bar, the user is taken to the type overview page which shows all available data types along with previews of three recently added items for each type. The choice to display these previews on the overview page was based on the assumption that recently added items are likely to be accessed more frequently than older items. Clicking on one of the previews will navigate directly to the details view for that item, while selecting View All for a particular data type will display the list view. The list view, as shown in Figure 5.10, presents all collected items of the selected type in a table format. The columns displayed in the table depend upon the specific data type; only the most relevant fields for sorting and filtering are shown. The contents of the table can be sorted by clicking on any of the column headers. The table also provides direct links to view, edit, or delete any data item shown.

Browsing by topic, on the other hand, is more difficult to support because topic collections can contain a mixture of different data types and designing an interface to display a set of heterogeneous data types in a consistent and logical fashion is a challenge. To solve this problem, we have chosen to utilize the data tile paradigm. This technique uses conformity in size and layout to present diverse objects in a homogeneous visual style. The size and shape of each individual data tile is completely identical, and the same visual layout is reused by each tile as well, even though the data being displayed within the layout may differ between data types. Figure 5.11 shows an example of the preview tiles shown on the overview page when browsing by topic. The design uses the same tile design as the data type overview page, however, in this case the types can be intermingled. For items without an image, a default data type icon is displayed. When the user selects a topic on the overview page, a list of all items categorized in that topic is displayed using larger tiles, which continue the same concept as the preview tiles but with more information displayed in a bigger area. Unlike browsing by data type, a table view does not make sense in this case because different data types have very few overlapping attributes so each column would only be useful for a small subset of the items.
Figure 5.10: List view for event type

Figure 5.11: Web interface for browsing objects of heterogeneous types
### 5.3.3 View

The item details view, as shown in Figure 5.12, provides an interface to review all available information for a single data item. This view is composed of two primary sections: a summary panel followed by a data details panel. Topics are displayed separately in the right sidebar, along with a set of buttons to edit, delete, or view the revision history of the current item.

The summary panel highlights the most important attributes for the item’s specific data type, allowing the user to quickly identify and understand meaningful information at a glance. The summary panel displays type-dependent templates which are uniquely designed to showcase the underlying structured data through different visualizations and interactions. For example, the summary panel for an event features a Google Map to interactively present the event’s location, while the summary panel for a person will dynamically display that person’s most recent posts on Twitter.

Below the summary panel, the data details panel displays a full listing of all object attributes sorted alphabetically. This view shows the raw data that is stored in the database. The presentation for this panel is uniform for all data types.

![Details view of an event](image_url)
5.3.4 Create, Edit, & Delete

Data items can be added, removed, or updated through the web interface. The same HTML form is used for both creating new data items and editing existing items. The content of the form is dependent on the data type, but the presentation style of the form is consistent for all types. Since many of the Schema.org data models contain a large number of attributes and long forms have a tendency to overwhelm and intimidate users, we decided to create both a basic and extended version of each form, an example of which can be seen in Figure 5.13. Initially, the basic form is shown to the user, which only displays the most essential fields while hiding the full complexity. To enter values for additional attributes, the user can expand the form to display all fields. This dynamically expands the form using JavaScript and briefly highlights the newly displayed fields. When the form is saved, all new or updated information must pass through the data integrity steps described in Section 5.2 including data validation and duplicate checking. If the user has entered invalid data, the form will be redisplayed and the errors will be highlighted.

Item deletion can be triggered by clicking the delete button which appears in both the list view and item details view. Whenever a delete is requested, a modal dialog is presented to the user to confirm the action. This confirmation step prevents accidental deletion of data from the database.

5.3.5 Manage

A number of administrative tasks are also supported by the web interface including management of revision history and data sources. For revision history, a user can view any previously saved version of any data item. From the item detail view, users can click the
View Revision History button in the right sidebar to see a list of all revisions. From here the user can select any individual revision to see what data was stored at that time.

The web interface also handles administration of data sources from which information is being automatically extracted. Users can view a list of all source URLs by clicking on the link in the top navigation bar. This overview helps one to understand the origins of the data in the system and allows the user to make any adjustments necessary. Existing sources can also be changed or deleted through the web interface. It is also possible to create new sources online, although we anticipate most data sources to be added to the system via the Firefox extension.
Conclusion

In this thesis we have designed and implemented a system for collecting and managing structured data from the web. Our work connects several disparate areas of research including web feeds, information extraction, data warehousing, and the Semantic Web. We have demonstrated how multiple data extraction and integration techniques can be used in combination to provide a more flexible and robust information harvesting environment. Our end-to-end solution supports the user throughout the entire information lifecycle from the initial gathering phase through on-going management and maintenance support.

Additionally, we have developed two innovative interfaces for interacting with our system. First, we provide a browser-integrated interface that assists non-technical end-users to discover and collect structured information while browsing the web. This browser extension simplifies and streamlines the information gathering task. Second, our web interface provides a unified view over the collected data that allows information of heterogeneous types and from diverse origins to be browsed and organized in a universal way. Our web interface also provides a mechanism to incorporate user feedback as well as to capture additional data from offline sources.

6.1 Contributions

This section presents a summary of the contributions that were developed as a part of this research project:

- **Extensible structured data collection system for the web.** Our primary contribution is the complete Sift system which provides an end-to-end solution to structured data collection and information management for the web. Our system is innovative in its multipronged approach to data harvesting and is therefore more flexible and robust than existing solutions. The included extensibility mechanism allows developers to easily extend the system and adapt it to their unique needs.
6.2 Future Work

This section proposes some topics of interest that might be relevant for the further development of the Sift data collection system.

- **Enhanced data cleansing and transformation.** The data integration phase could be improved by providing more options and further control over the cleaning and conforming process. The system currently handles data validation, schema mapping, and duplicate entity resolution. However, there are several additional transformations that could be implemented such as support for reformatting, aggregation, standardization, and allowing users to create custom defined rules. Improvements in cleaning and normalizing the incoming data would also help the system to identify duplicate entities more accurately.

An example can demonstrate the potential advantages of these proposed data cleansing enhancements. Imagine three event listings on separate websites which occur in the same city yet are each presented with a different syntax: “San Francisco, CA, United States”, “San Francisco, California”, and “USA / CA / San Francisco”. A human can easily discern that all three strings refer to the same location, but this is much more difficult for a computer to determine automatically. By applying additional data cleansing and transformation rules, each of these location strings could be converted into a common format with a standardized abbreviation scheme. As one potential option, the user could choose to store all locations as “City, State, Country” with country name abbreviated using the two-letter country code defined by the ISO 3166 standard. By reformatting and standardizing this field, the system will then be able to more accurately group, sort, and compare the saved values.
• **Communal data sources and extractor repository.** In the style of popular crowdsourced systems, an architecture could be devised that enables the sharing of both extractor components and data sources. First, developers could distribute their custom-built extractors so that they could be reused by the rest of the community. For example, if Alice extends the system by creating a new extractor that handles the extraction of Dublin Core metadata, then she can add this extractor to the communal repository. If Bob later wants to extract Dublin Core metadata, then he can benefit from Alice’s previous work by installing and using that component from the shared repository. In this way, duplicate work can be avoided and people can instead spend their time working on new challenges.

Non-technical end-users can also benefit from a communal repository. First, they could extend the functionality of their systems by easily installing additional extractor components that had been previously developed and shared by other, more technically-inclined users. Also, all users could share their data sources and the extraction rules necessary for those sources. A website would only need to be identified once as a data source for the Sift system with the extraction rules defined by the initial user. Subsequent users would then be able to collect information from that previously defined data source with ease. Over time the repository would contain a wealth of knowledge regarding where to find information on a certain topic or of a specific object type.

• **Advanced browsing and search capabilities.** Our system is specifically designed to collect structured data, which includes semantic metadata in addition to the content itself. Our web interface highlights a few uses of this structured metadata to enhance the user experience, but this is just a small example of the rich interactions that are possible with this information. One possibility is to develop a faceted browsing interface that leverages the structured data to explore the information across multiple dimensions. Faceted browsing, also called faceted search, helps users to discover insights about their data through context-dependent navigation based on schema attributes.

Another possibility is to enable structured querying of the underlying database. Third-party search engines like Google do not have access to the databases behind the websites they index, and, therefore, they can only provide full text search over the data presented in the HTML documents. Since our system is able to reconstruct the structured data from web pages, there is an opportunity to give the user the ability to issue more complex, structured queries based on the schema of the database.

• **Multi-user support.** Our system is designed with the intention of allowing a single user to collect and manage a personal collection of information. However, this concept could easily be extended to support a group of users that wish to collectively build and manage a data set together. Opening the platform to support crowdsourcing of data entry, feedback, and maintenance could help to create more complete and accurate data sets. An entire community could work together on the problem instead of just one single person, which should help in discovering a wider range of online data sources to collect from and in checking for errors or inaccuracies in the existing data more
quickly and thoroughly.

We have implemented a default web interface which already allows a user to manually create or edit saved items online, but the current implementation is not properly designed to support crowdsourcing as it lacks some basic functionality such as user authentication and administration. Additionally, the system would need to be prepared to properly deal with spam and malicious users. This might include moderation support so that all updates would need to be approved by a trusted administrator before being published.

- **Multi-page extraction.** The current system implementation is only capable of extracting information from a single web page at a time. However, the details of a particular data item often span across multiple pages. For instance, a web page displaying the details of a movie may contain a list of cast members as links to separate web pages which contain the details for each person. Ideally, our system would be able to follow each of these links and embed the relational information found on the linked pages. This enhancement would make the collected data records more thorough and complete.

Also, many websites display lists of related data items that often span across multiple pages; rather than displaying a long list of 100 items on a single page, websites will often break this up into 10 pages of 10 items each. Making the system aware of instances of paged lists and how to navigate between pages would enable more information to be collected automatically with less effort by the user.
A

How to Extend the System

This appendix will explain the technical details of implementing and installing a custom extractor component for the Sift system. We will walk through the process of developing a new extractor using the example of the Open Graph extractor described in Section 5.2.1 as a model.

A.1 Developing a Custom Extractor

Extractor components are Python classes that extend a base class provided by the Sift framework. Developing a custom extractor is a simple process that involves three steps:

1. Extend a base class (either Extractor or HTMLExtractor)
2. Define a unique extractor key
3. Implement one mandatory method (depends on base class)

1. Extend base class
In order to define a new Extractor, a new Python class should be declared that extends the sift.components.extractors.Extractor base class. Alternatively, one could instead extend the sift.components.extractors.HTMLExtractor class if the extractor plans to use the lxml package to parse the input document. The HTMLExtractor option is offered as an alternative in order to provide enhanced performance by ensuring that the common task of (X)HTML parsing only occurs once for all extractors. For all custom extractors that do not need to parse (X)HTML, and for those that would like to use an alternative method of parsing (X)HTML inputs, the default Extractor class should be used as the base for extension.

For our example Open Graph extractor, we do not need access to the lxml parser because we rely on an alternative method to parse the HTML. Therefore, we will declare our class using the default Extractor class as a base as shown in Listing A.1.
A.1. DEVELOPING A CUSTOM EXTRACTOR

Listing A.1: Extending the base Extractor class

```python
from sift.components.extractors import Extractor
class OpenGraphExtractor(Extractor):
    ...
```

2. Define unique extractor key
Every extractor must define a string value that can be used to uniquely identify the extractor component. This value must be stored in the class’s `key` attribute and is not allowed to contain the following characters: `.,[]"'`. This key value should be meaningful because it will be displayed to users within the Firefox browser extension and used to define data mappings. Listing A.2 shows the key definition for the example Open Graph extractor.

Listing A.2: Defining the unique key value

```python
class OpenGraphExtractor(Extractor):
    # Unique identifier for this extractor
    key = 'Open Graph'
```

3. Implement required method
Custom extractor classes always need to implement one single method. Extractors that extend the `Extractor` class must implement `extract(self, input_)`, while those that extend `HTMLExtractor` must implement `extract_HTML(self, input_)`. The only difference between these two methods is that the `input_dictionary` for `extract_HTML()` contains one extra key, `html`, that contains the parsed `lxml` etree output.

Listing A.3: Method interface definition

```python
def _extract(self, input_):
    ""
    This method must be implemented by all subclasses.
    Args:
    input_ (dictionary): Contains at least the following:
    input_['source'] ['url ']: Final URL after all redirects
    input_['source'] ['original_url ']: Original URL
    input_['source'] ['type ']: Content-type from HTTP header
    input_['source'] ['buffer ']: String-like object containing the downloaded document
    Returns: Dictionary. Contains at least all values from
    input_ dictionary argument.
    Raises: None.
    ""
    raise NotImplementedError("Extractor subclasses must implement this method")
```
Listing A.3 shows the interface for the \_extract() method that all Extractor subclasses must implement. This method is passed one argument, a dictionary which contains information about the data source as well as the output from previously executed extractors. The order of execution is not assured so extractors should never rely on the output of another extractor. The ‘source’ key is guaranteed to be set before this method is executed. This key contains both the URL and the downloaded content of the data source. Extractors should always process the downloaded copy of the document rather than using the URL to re-fetch the document over the network.

Extractor components are executed one after another in a chain-like sequence similar to a Unix pipe; the output of one extractor is used as the input for the next extractor. Therefore, this method returns a dictionary which contains all information from the original dictionary that was passed into the function. If the extractor completes successfully, then it adds an additional key to the output dictionary which contains the values that it has extracted. The key must match the extractor’s unique key attribute value. Also, extractors should never raise any exceptions. Instead, they should fail silently and always at least return the same dictionary that was passed into the method.

Listing A.4: Implementing the required method

```python
def _extract(self, input_):
    try:
        url = input_['source']['url']
        buffer_ = input_['source']['buffer']

        # Process the input to extract some data
        og = PyOpenGraph(url, buffer_)

        # Clean up the extracted data, if necessary
        for key in og.metadata.keys():
            if key.startswith('/ '):
                del og.metadata[key]

        # If success, add to dictionary with unique key
        if og.metadata:
            input_[self.key] = og.metadata
    except:
        # Always catch exceptions and fail silently
        pass

    # Always return the input_ dictionary
    return input_
```

Listing A.4 shows the completed version of the \_extract method for our example Open Graph extractor with inline comments. Note that we use the PyOpenGraph library for parsing the HTML and extracting the Open Graph metadata, and we post-process that output
to remove unnecessary values.

By combining the output of the three previous steps, we now have a complete extractor component that is ready to be installed and used. The final result can be seen in Listing A.5.

Listing A.5: Example of a complete extractor definition with inline comments

```python
from sift.components.extractors import Extractor
from PyOpenGraph import PyOpenGraph

class OpenGraphExtractor(Extractor):
    """Extracts all Open Graph protocol metadata""

    # Define the unique key to identify this extractor
    key = 'Open Graph'

    # This method must be implemented by all extractors
    def __extract(self, input_):
        try:
            url = input_['source']['url']
            buffer_ = input_['source']['buffer']

            # Process the input to extract some data
            og = PyOpenGraph(url, buffer_)

            # Clean up the extracted data, if necessary
            for key in og.metadata.keys():
                if key.startswith('/ '):
                    del og.metadata[key]

            # If success, add to dictionary with unique key
            if og.metadata:
                input_[self.key] = og.metadata

        except:
            # Always catch exceptions and fail silently
            pass

        # Always return the input_ dictionary
        return input_
```

A.2 Installing a Custom Extractor

After a new extractor has been developed, it must be installed before it can be used with the Sift system. Installing a custom extractor is a simple process that consists of four steps, each of which will be explained in turn.

1. Save file under extractors directory
2. Update extractors list

3. Update default mappings (optional)

4. Restart server

1. **Save file**

   Each custom extractor should be saved in its own file under the `sift/components/extractors` directory. The exact location of this folder will depend on the installation path for the Sift package. For our example extractor, we create a new file named `opengraph.py` and save it under the appropriate directory for our Sift installation.

2. **Update extractors list**

   After the file has been saved, the new extractor class must be explicitly added to the list of known extractors in `sift/components/extractors/__init__.py`. For our example, we add ‘OpenGraphExtractor’ to the `__all__` list as well as the associated import statement, which is shown in Listing A.6.

   ```python
   ... from sift.components.extractors.opengraph import OpenGraphExtractor
   __all__ = [ 'Extractor', 'ContentTypeExtractor', 'FeedExtractor', 'HTMLExtractor', 'HTML5MicrodataExtractor', 'iCalendarExtractor', 'JSONExtractor', 'HTMLMicroformatsExtractor', 'RDFaExtractor', 'OpenGraphExtractor']
   ```

3. **Update default mappings (optional)**

   This optional but recommended step allows the developer to specify which output fields from their custom extractor map to which schema attributes for each of the internal data types, as well as the preference order between different extractors for each individual field. These default mappings are executed every time a new extraction is requested by the Sift Firefox browser extension, and they allow the system to guess the most appropriate schema mappings for each data source.

   Default mappings are defined in `mappings.yaml`, a YAML\(^1\) configuration file located in the Sift package directory. The file is organized by Schema.org data type and then attribute. For each attribute a list of values can be defined which indicate the order of preference with the first mapping listed being the most preferred. Listing A.7 shows a section of the default mapping definition file that has been updated to include values from our new Open Graph extractor where appropriate.

   \(^1\)http://www.yaml.org/
Listing A.7: Snippet from default mappings configuration file

```plaintext
... 
Event:

  name:
    - [iCalendar, VEVENT, SUMMARY]
    - [Open Graph, title]
    - [Microformats, vevent, summary]
    - [HTML Microdata, event, summary]

  url:
    - [iCalendar, VEVENT, URL]
    - [Open Graph, url]
    - [Microformats, vevent, url]
    - [Microformats, vevent, url, href]
    - [HTML Microdata, event, url]

  ...
```

4. **Restart server**

The final step is to restart the server in order for the system to recognize the new extractor. After that, the installation will be complete and the custom extractor can be used. The extractor will be executed automatically for all future extraction requests from the Firefox browser extension.
Proposed Date Similarity Metrics

Approximate string matching, also known as fuzzy string matching, is a well-studied research problem that attempts to detect the similarity or edit distance between strings. There have been many string similarity metrics and distance functions proposed over the years including, but not limited to, Levenshtein distance, Damerau-Levenshtein distance, Hamming distance, Jaro-Winkler distance, Jaccard coefficient, and Cosine similarity. Although much effort has gone into the problem of string comparison, there have not yet been any similarity metrics specifically designed to support approximate date matching. This area has so far been overlooked by researchers even though input errors and formatting differences can occur nearly as easily with date objects as with strings.

There are many potential causes for two date representations of the same date-in-time to contain differences. Perhaps the most obvious cause is errors in user input such as inadvertent typos. It is possible that a user may accidentally hit a wrong key while manually entering the digits of a date. Some user interfaces, such as those on mobile devices, are especially prone to this type of mistake. The user may also enter the date information in a different order than the system is expecting; formatting conventions of date values can frequently cause issues when inputting data or when converting data between different systems. For example, dates are typically entered in the MM-DD-YYYY format in some cultures, while people in other parts of the world commonly expect dates to be formatted as DD-MM-YYYY. Another possible source of approximate duplicates in dates is errors that result from OCR (Optical Character Recognition) scanning. These automated input systems are prone to mistakes and could cause problems with date values as well as strings. Finally, date discrepancies may be due to different levels of specificity. Some dates may include an exact time down to the microsecond while others may exclude the time component altogether. Also, the full specificity of the information might not be known when a date is first entered into a system, yet most systems commonly require values for all components of a date before that data can be saved. In this case, end-users often enter arbitrary data for the missing attributes as a placeholder in order to meet the requirements of the system.
Fuzzy date matching can be a useful technique in several different scenarios. One example use case is for duplicate entity resolution of events. The start and end dates are two of the most important characterizing attributes of an event. For example, a recurring event may repeat multiple times with the exact same name and location. In this scenario, the only distinguishing features between event instances are the start and end dates. Current fuzzy matching techniques can help detect typos in string values such as the name attribute but would be unable to detect input errors in the date fields. For events or any other object types that rely on date attributes, the ability to detect the similarity of date values would help to improve duplicate entity resolution. Other potential use cases for approximate date matching include input validation of dates, input suggestions such as a spell check equivalent for dates, and data integration between different systems.

In order to support approximate date matching, we propose five similarity metrics that are specific to date and date/time data types:

1. **Date difference** or *Days between dates*: This basic metric calculates the absolute distance between two dates. Days are used as a constant unit of time for this calculation. The order of parameters is not significant as we are only interested in the absolute value of the distance, not the direction. The output of this calculation is a discrete integer value in the range \([0, \infty)\). A lower value indicates a shorter distance and therefore dates that are more similar. This metric is useful for identifying dates that are close together in time, even if they occur in separate months or years.

   **Listing B.1: Sample date difference calculations**

   ```plaintext
   # Sample function definition
   DateDiff(date1, date2)

   # Comparison of dates across month and year changes
   DateDiff(31/12/2011, 01/01/2012) = 1

   # Order of parameters is not significant
   DateDiff(01/01/2011, 01/01/2012) = 365
   DateDiff(01/01/2012, 01/01/2011) = 365
   ```

2. **Fixed date range** or *Date radius*: This metric returns a binary value that indicates whether two dates are within a fixed distance of each other. For example, this calculation could be used to determine if one date is plus or minus 10 days of another date. The unit of time used to define the radius distance can be adjusted based on the needs of the user and the scale of the data. There are many possible units of time that could be used including second, minute, hour, day, week, month, year, decade, etc. The output of this function is always either 1 or 0; a value of 1 is returned if the dates are within the given radius of each other, otherwise 0 is returned.
3. **Fractional time period**: This metric compares the distance between dates as a fraction of a fixed time interval. The fractional time period function is a normalized version of the previously defined date difference metric. The time period can be adjusted based on the needs of the user but must be a larger unit of time than a day because that is the unit of measure for the date difference calculation. The output of this function is a continuous value in the range $[0, 1]$. If the date difference is larger than the time period, a value of 1 is always returned.

Listing B.3: Sample fractional time period calculations

```python
# Sample function definition
FractionTime(time interval, unit of measure, date1, date2)

# Fraction of a year calculation
FractionTime(1, 'year', 31/12/2011, 01/01/2012) = 1/365

# If difference is more than the time period,
# then the maximum value is returned
FractionTime(1, 'year', 01/01/2009, 01/01/2012) = 1
```

4. **Date component overlap**: This metric breaks down each date value into separate components and then compares the amount of overlap between these components. For date objects these components include day, month, and year, while datetime objects also include components for hour, minute, and second. Each component is compared against the same component of the other date, and the output is the fraction of components that match. For example, the date overlap between January 2nd, 2011 and February 1st, 2011 is 1/3 because the only date component that matches for both dates is the year.
Listing B.4: Sample date component overlap calculations

```c
#define Sample function definition
def DateOverlap(date1, date2)

#define Breakdown of a example comparison:
#define Day Component: 01 != 02
#define Month Component: 02 != 01
#define Year Component: 2011 == 2011

DateOverlap(01/02/2011, 02/01/2011) = 1/3

#define Comparing exact duplicates
DateOverlap(01/02/2011, 01/02/2011) = 3/3 = 1
```

5. **Jaccard similarity for dates**: This metric adapts the Jaccard coefficient for comparing dates. It breaks down each date value into sets of four components and then compares the similarity of these sets. The component set consists of the two-digit day, two-digit month, first two digits of the four-digit year, and the last two digits of the four-digit year:

\[
\text{Date}(DDMMYYyy) \Rightarrow \{DD, MM, YY, yy\}
\]

The Jaccard similarity between dates \(a\) and \(b\) is the ratio of the intersection of the component sets for those dates, \(A\) and \(B\) respectively, divided by the union of those same sets.

\[
\text{JaccardSim}(a, b) = Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}
\]

The following example shows how the Jaccard similarity can be calculated for two date values, and it also illustrates this metric’s usefulness for determining when parts of a date have been rearranged, which can easily occur due to formatting conflicts.

\[
\text{JaccardSim}(01/02/2011, 02/01/2011) \Rightarrow
a = 01/02/2011 \Rightarrow A = \{01, 02, 20, 11\}
b = 02/01/2011 \Rightarrow B = \{02, 01, 20, 11\}
A \cap B = \{01, 02, 11, 20\} \Rightarrow |A \cap B| = 4
A \cup B = \{01, 02, 11, 20\} \Rightarrow |A \cup B| = 4

\text{JaccardSim}(a, b) = \frac{|A \cap B|}{|A \cup B|} = \frac{4}{4} = 1
\]

Listing B.5: Sample Jaccard date similarity calculations

```c
#define Sample function definition
def JaccardSim(date1, date2)

#define Example date comparisons
JaccardSim(01/02/2011, 02/01/2011) = 4/4 = 1
JaccardSim(31/12/2011, 01/01/2012) = 2/5
```
# List of Figures

2.1 Web feed icon ........................................... 6

3.1 Venn diagram illustrating hybrid approach .................. 13
3.2 Conceptual overview of the content aggregation engine .... 14

4.1 System architecture overview .............................. 20

5.1 Firefox toolbar button and notification icon .............. 27
5.2 Firefox context menu integration .......................... 27
5.3 Firefox extension preferences dialog ......................... 27
5.4 First step of Firefox extension extraction wizard ......... 29
5.5 Second step of Firefox extension extraction wizard ....... 29
5.6 Final step of Firefox extension extraction wizard ......... 30
5.7 Sample data mapping definition ............................ 34
5.8 Automated update architecture .............................. 38
5.9 Interface for topic assignment with suggestions .......... 40
5.10 Web interface - list view .................................. 41
5.11 Web interface - browse by topic .......................... 41
5.12 Web interface - details view ............................... 42
5.13 Web Interface - basic and extended edit forms .......... 43
Acknowledgements

I would like to thank my adviser Matthias Geel for his support and valuable feedback throughout the course of the thesis. I would also like to thank Professor Norrie for allowing me to propose my own thesis topic and for the freedom to work with the programming languages and technologies of my own choosing. Finally, I want to thank Maximilian Speicher and Michael Nebeling for their encouragement, advice, and good company throughout the year.

This work is dedicated to my family and especially to my wonderful wife, Emily. I am eternally grateful to my parents for everything they have done for me throughout my life, and I would like to thank them for giving me all of the opportunities that I have today. Lastly, and most importantly, I want to thank my wife for her constant love and support throughout the course of my studies at ETH and always.


