Not So Creepy Crawler: Crawling the Web with XQuery

Franziska von dem Bussche[1], Klara Weiand[1], Benedikt Linse[1], Tim Furche[1,2], François Bry[1]

[1] Institute for Informatics, University of Munich
Oettingenstr. 67, 80538 München, Germany
[2] Oxford University Computing Laboratories, Parks Rd, Oxford OX1 3QD, UK

Abstract. Web crawlers are increasingly used for focused tasks such as the extraction of data from Wikipedia or the analysis of social networks like last.fm. In these cases, pages are far more uniformly structured than in the general Web and thus crawlers can use the structure of Web pages for more precise data extraction and more expressive analysis.
In this demonstration, we present a focused, structure-based crawler generator, the *"Not so Creepy Crawler"* (NC^2). What sets NC^2 apart, is that all analysis and decision tasks of the crawling process are delegated to an (arbitrary) XML query engine such as XQuery or Xcerpt. Customizing crawlers just means writing (declarative) XML queries that can access the currently crawled document as well as the metadata of the crawl process. We identify four types of queries that together suffice to realize a wide variety of focused crawlers.

1 Introduction

In this demonstration, we present the *"Not so Creepy Crawler"* (NC^2), a novel approach to structure-based crawling that combines crawling with standard Web query technology for data extraction and aggregation. NC^2 differs from previous approaches to crawling in that it allows for high level of customization throughout every step in the crawling process. The crawling process is entirely controlled by a small number of XML queries written in any XML query language: some queries extract data (to be collected), some links (to be followed later), some determine when to stop the crawling, and some how to aggregate the collected data.

This allows easy, but flexible customization through writing XML queries. By virtue of the loose coupling between an XML query engine and the crawl loop, the XML queries can be authored with standard tools, including visual pattern generators [1]. In contrast to data extraction scenarios, these same tools can be used in NC^2 for authoring queries of any of the four types mentioned above.

A demonstration of NC^2, accessible online at `http://pms.ifi.lmu.de/ncc`, showcases two applications: The first extracts data about cities from Wikipedia with a customizable set of attributes for selecting and reporting these cities. It illustrates the power of NC^2 where data extraction from Wiki-style, fairly

homogeneous knowledge sites is required. The second use case demonstrates how easy NC^2 makes even complex analysis tasks on social networking sites, exemplified by last.fm.

NC^2 has already been presented at the 2010 WWW conference [3].

2 Crawling with XML Queries

"Not So Creepy Crawler": Architecture. The basic premise of NC^2 is easy to grasp: A crawler where all the analysis and decision tasks of the crawling process are delegated to an XML query engine. This allows us to leverage the expressiveness and increasing familiarity of XML query languages and provide a highly configurable crawler generator, which can be configured entirely through declarative XML queries.

To this end, we have identified those analysis and decision tasks that make up a focused, structure-based crawler, together with the data each of these tasks requires.

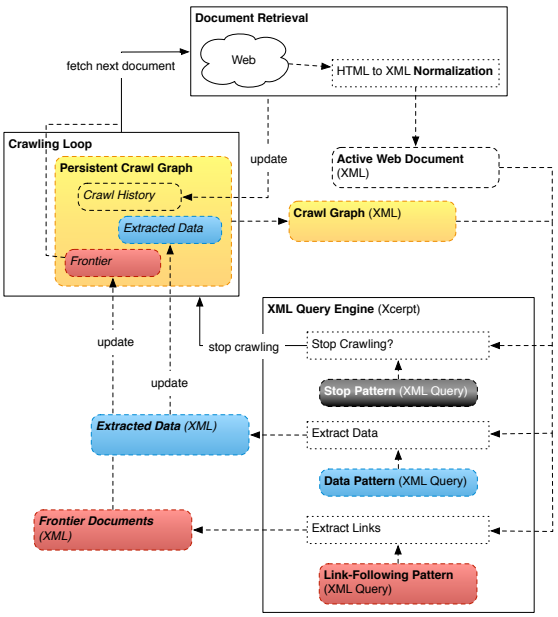

Fig. 1. Architecture NC^2

XML patterns. Central and unique to a NC^2 crawler is uniform access to both object data (such as Web documents or data already extracted from previously crawled Web pages) and metadata about the crawling process (such as the time and order in which pages have been visited, i.e., the crawl history). Our *crawl graph* not only manages the metadata, but also contains references to data extracted from pages visited previously. The tight coupling of the crawling and extraction process allows us to retain only the relevant data from already crawled Web documents.

This data is queried in a NC^2 crawler by three types of XML queries (see Figure 1):

(1) *Data patterns* specify how data is extracted from the current Web page. A typical extraction task is "extract all elements representing events if the current page or a page linking to it is about person X". To implement such an extraction task in a data pattern, one has to find an XML query that characterizes "elements representing events" and "about person X". As argued above, finding such queries is fairly easy if we crawl only Web pages from a specific Web site such as a social network.

(2) *Link-following patterns* extract all links from the current Web document that should be visited in future crawling steps (and thus be added to the crawling frontier). Often these patterns also access the crawl graph, e.g., to limit the

crawling depth or to follow only links in pages directly linked from a Web page that matches a data pattern.

(3) *Stop patterns* are boolean queries that determine when the crawling process should be halted. Typical stop patterns halt the crawling after a given amount of time (i.e., if the time stamp of the first crawled page is long enough in the past), number of visited Web pages, number of extracted data items, or if a specific Web page is encountered.

There is one more type of pattern, the *result pattern*, of which there is usually only a single one: It specifies how the final result document is to be aggregated from the extracted data. Once a stop pattern matches and the crawling is halted, the result pattern is evaluated against the crawl graph and the extracted data, e.g., to further aggregate, order, or group the crawled data into an XML document, the result of the crawling.

All four patterns can be implemented with any XML query language. In this demonstration we use Xcerpt [2] and XQuery.

System components. How are these patterns used to steer the crawling process? Crawling in NC^2 is an iterative process. In each iteration the three main components work together to crawl one more Web document (see Figure 1):

(1) The *crawling loop* initiates and controls the crawling process: It tells the document retrieval component to fetch the next document from the crawling frontier (the list of yet to be crawled documents).

(2) The *document retrieval* component retrieves and normalizes the HTML document and tells the crawling loop to update the crawl history in the crawl graph (e.g., to set the document as crawled and to add a crawling timestamp).

(3) The *XML query engine* (in the demonstrator, Xcerpt) evaluates the stop, data, and link-following patterns on both the active document and the crawl graph (containing the information which data patterns matched on previously crawled pages and the crawl history). Extracted links and data are sent to the crawling loop which updates the crawl graph.

(4a) If none of the stop patterns matches the iteration is finished and crawling starts again with the next document in step (1), if there is any.

(4b) If one of the stop patterns matches in step (3), the crawling loop is signalled to stop the crawling. The XML query engine evaluates the result pattern on the final crawl graph and the created XML result document is returned to the user.

References

1. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *VLDB*, 2001.
2. F. Bry, T. Furche, B. Linse, A. Pohl, A. Weinzierl, and O. Yestekhina. Four lessons in versatility or how query languages adapt to the web. In *Semantic Techniques for the Web, The REWERSE Perspective*, *LNCS* 5500. Springer, 2009.
3. F. von dem Bussche, K. A. Weiand, B. Linse, T. Furche, and F. Bry. Not so creepy crawler: easy crawler generation with standard XML queries. In *WWW*, 2010.