DoCQS: A System for Supporting Data-oriented Querying over Web Content

Mianwei Zhou, Tao Cheng, Kevin Chen-Chuan Chang
Computer Science Department, University of Illinois at Urbana-Champaign, USA
{zhou18, tcheng3, kcchang}@illinois.edu

ABSTRACT
Witnessing the richness of data in document content and many ad-hoc efforts for finding such data, we propose a Data-oriented Content Query System (DoCQS), which is oriented towards fine granularity data of all types by searching directly into document content. DoCQS uses the relational model as the underlying data model, and offers a powerful and flexible Content Query Language (CQL) to adapt to diverse query demands. In this demonstration, we show how to model various search tasks by CQL statements, and how the system architecture efficiently supports the CQL execution. Our online demo of the system is available at http://wisdm.cs.uiuc.edu/demos/docqs/.

Categories and Subject Descriptors
H.3.4 [INFORMATION STORAGE AND RETRIEVAL]: Systems and Software—Question-answering (fact retrieval) systems; H.3.1 [INFORMATION STORAGE AND RETRIEVAL]: Content Analysis and Indexing—Indexing methods

General Terms
Design, Languages, Performance

Keywords
Content Query, Entity Search, Data Retrieval, DoCQS, CQL, Relational Data Model

1. INTRODUCTION
The development of the World Wide Web largely enriches the content of web pages. People nowadays are no longer satisfied with just finding relevant pages, but hope to retrieve finer granularity information inside the documents, e.g., movie release date, book price. Many applications could be built on the Web based on recognizing and retrieving small pieces of information inside the pages. Consider the following emerging Web-based application scenarios:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. SIGMOD ’10, June 6–11, 2010, Indianapolis, Indiana, USA. Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

Web-based Information Extraction (WIE) Information extraction, with the aim to identify information systematically, has naturally turned to Web-based, for harvesting numerous “facts” online—e.g., to assemble a table of all the (country, capital) pairs (say, (France, Paris)). Recent WIE efforts (e.g., [7, 2, 9]) have mostly relied on phrase patterns (e.g., “X is the capital of Y”) for large scale extraction. With the richness and redundancy of the Web, we can use more general patterns to scrape millions of facts from the Web.

Typed-Entity Search (TES) As the Web hosts all sorts of data, as motivated earlier, several works (e.g., [5, 4, 6, 3]) proposed to search for specific types of entities, such as person names near “invent” and “television.” Such techniques often rely on readily available information extraction tools to first extract data types of interest, and then matching the extracted information units with the specified keywords based on some proximity patterns.

Web-based Question Answering (WQA) Many recent efforts (e.g., [1, 8, 10]) exploited the diversity of the Web for virtually any ad-hoc questions, and leveraged the abundance to find answers by simple statistical measures (instead of complex language analysis). Given a question (e.g., “where is the Louvre Museum located?”), WQA needs to find information of certain type (a location) near some keywords (“louvre museum”), and examine numerous evidences (say, counting co-occurrences) to find potential answers.

These applications are interested in data types of all kinds (e.g., noun phrases, person names, address, etc.) embedded in pages. To support these ad-hoc content search tasks, we propose a novel general search system over data embedded in web pages, Data-oriented Content Query System (DoCQS). DoCQS aims at minimizing the efforts people take for querying large scale document content. The previous works on WIE, TES and WQA usually rely on hard-coded extraction and search strategies, which are inextensible and hard to debug. As an alternative, DoCQS describes these content search tasks by the Content Query Language (CQL), which is highly flexible and customizable. More specifically, the proposed DoCQS and CQL have the following features:

Extensible Data Types Currently, most efforts towards finding fine granularity information use hard-coded heuristic rules for extraction, which is static and hard to extend. Recognizing the need for supporting extensible data types, in DoCQS, we support specialized data types over existing basic types (e.g., we can specialize “number” data type into “zipcode”, “price”, “population”, etc.) in an online fashion.

Flexible Contextual Patterns While phrase patterns have shown their usefulness in Web information extraction, they are
Limited to scenarios where clear sequential patterns could be specified. In DoCQS system, we design a series of more expressive contextual patterns, facilitating users to utilize all available information that appears in the context of target information. We further design an indexing framework to efficiently support online matching of these patterns.

**Customizable Scoring** The design of scoring functions for most current content query efforts only fits to their own domain needs. As a general content search system, it should enable users to customize the scoring to measure the confidence of results from various domains. DoCQS embeds scoring computation in every CQL operation, and users can design the scoring function themselves via CQL specification.

The rest of the paper is organized as follows. Section 2 presents an overview of the system. Section 3 introduces the architecture of DoCQS, and we show the system demonstration in Section 4. For detailed discussion of the DoCQS design, refer to [11].

## 2. SYSTEM OVERVIEW

### 2.1 Data Model

Because the search target of DoCQS is typed data, the traditional IR data models used (e.g., the vector space model which views query and documents as keyword vectors) are no longer appropriate. We come up with a new data model to meet the proposed features of the system. Motivated by the similarity between CQL and relational operations, we take relational model as our data model. Each data type is modeled as a relational table characterized by the following schema:

- **doc**: The document ID where the keyword/data type appears.
- **pos**: The word position of the occurrence.
- **span**: The number of keywords that the occurrence covers. For keyword, span is always 1.
- **val**: The content of the occurrence. For keyword, the content is always itself (and therefore omitted).

---

### 2.2 Content Query Language

With the relational model chosen as our data model, we design the Content Query Language (CQL). CQL takes a special form of SQL with several new constructs to meet the system requirement. It consists of two types of operations: 1) Data Type Definition Operation; 2) Data Retrieval Operation.

#### 2.2.1 Data Type Definition Operation

The specification of the data type definition is

```sql
DEFINE DATATYPE Tnew AS Tbase
WHERE condition
```

where `condition := {boolFunc|fuzzyFunc}[w] [{AND|OR} condition]`

The above CQL defines a new data type `Tnew` based on the basic type `Tbase` satisfying the specified conditions. There are two types of functions in the condition: boolean function and fuzzy function. The difference between them is that the former returns boolean results while the latter changes the `conf` value by how well the condition is matched. The weight `w` behind the condition measures its importance when multiple conditions are involved. One example is shown below:

**Example 1**

```sql
DEFINE DATATYPE #population AS #number
WHERE (pattern(“#{number inhabitants}”)[1.0] OR pattern(“{population of ?(0,1) #number}”)[0.8])
AND ~likeLargeNum(#number)
```

Example 1 defines `#population` based on `#number`. The definition uses the boolean function `pattern()` to indicate that `#number` before “inhabitants” or after “population of” is likely to be `#population`. As a fuzzy function, `~likeLargeNum()` assigns higher confidence to larger numbers to adapt to the property of `#population`. The newly defined data type has the same schema as the basic data types, and therefore can be accessed later in the same way.

The functions in the DoCQS system are highly extensible by supporting a wide range of patterns. In addition to the simple sequential pattern `{X₁ X₂ ... Xₙ}` described above, our system also supports other types of patterns:

- **Window Pattern** `[X₁ X₂ ... Xₙ](m)`: Matched if all `Xᵢ` appear within an `m`-words window;
- **Disjunction Pattern** `(X₁|X₂|...|Xₙ)`: Matched if any `Xᵢ` in the list is matched;
- **Inner Pattern** `(X₀:X₁)`: Matched if `X₀` is matched and `X₁` lies in the scope of `X₀`.

Each pattern can take as input any keywords, data types or other nested patterns. Our system also supports flexible user-defined functions (e.g., `~likeLargeNum()`).
2.2.2 Data Retrieval Operation

A data retrieval operation takes the following specification:

```sql
SELECT T[1][*] [. . . , Tn[*], expr1, . . . , exprm]
FROM T1, ..., Tn
WHERE condition
[GROUP BY T1[. . . , val]]
[ORDER BY expr]
```

With `#population` defined, one concrete example is shown in Example 2 to find the population of China:

Example 2

```sql
SELECT #population, conf()
FROM #population
WHERE pattern("[China #population](10)")
GROUP BY #population
ORDER BY conf()
```

In the data retrieval operation specification, the FROM clause lists the source data types. These relations are naturally-joined by the doc attribute. The WHERE clause refines the retrieved tuples (in Example 2, it only collects `#population` with keyword “China” around in a 10-word window).

The tuples are further aggregated by the GROUP BY clause which assigns a confidence value to each group according to their frequencies. Finally, the ORDER BY clause ranks the result by the `conf()` function. As the default scoring function in the system, `conf()` takes data type confidence and pattern weighting as input, producing the confidence measurement for the final results. Specifically, the default confidence measure has the following forms, where \( P_i . score \) indicates how well pattern \( P_i \) is matched and \( w \) indicates the pattern weighting:

- **AND operator**: Given \( (P_1 \text{ AND } P_2)[w] \), the confidence of the generated result is \( \text{conf}((P_1 \text{ AND } P_2)[w]) = P_1 . score \times P_2 . score \times w \).
- **OR operator**: Given \( (P_1[w_1] \text{ OR } P_2[w_2]) \), the confidence is \( P_1 . score \times w_1 \), if the result tuple comes from matching \( P_1 \), and similar for \( P_2 \).
- **GROUP BY operator**: Given \( n \) tuples with confidence \( \text{conf}_1, . . . , \text{conf}_n \),

\[
\text{conf}(\text{conf}_1 \ldots \text{conf}_n) = 1 - \prod_{i=1}^{n}(1 - \text{conf}_i).
\]

The default `conf()` provides convenience to users who do not want to fine-tune the detailed scoring function. Users are free to customize their own scoring functions to adapt to different domains.

3. SYSTEM ARCHITECTURE

Figure 2 shows the architecture of DoCQS. In this section, we zoom into the function of each component in the architecture.

3.1 Offline Processing

The offline stage prepares data for efficient online access. The data source is a list of web pages with all html tags removed. We utilize off-the-shelf Named Entity Recognition (NER) tools to extract the basic data types from pages, and index them together with keywords. In the index layer, we choose inverted index instead of the database indexing framework. Because the most common table operation in CQL is to traverse the keyword or data type tables, we use inverted index for its distinctive advantage of efficiently supporting the traversal operation.

However, in the experiment, we find that some keywords or data types have very long index lists (e.g., `#number` everywhere in any data corpus), a basic inverted index structure could not satisfy the performance requirement of online querying. As a complement, we design another advanced inverted index in the system. The design principle is to use selective keywords to quickly zoom into a small part of a long list by joining selective keywords with common data types. This avoids traversal on the inverted list of highly frequent keywords or data types. Refer to [11] for the detailed discussion.

3.2 Online Query

The online stage is responsible for executing the CQL query. Once the parsing layer receives a CQL statement, it first justifies whether it belongs to a data type definition. If it is, this CQL statement will be simply stored in the data type repository for future reference; otherwise, for data retrieval operation, the layer will detect all referenced non-data type repository and data types have very long index lists (e.g., `#number` everywhere in any data corpus), a basic inverted index structure could not satisfy the performance requirement of online querying. As a complement, we design another advanced inverted index in the system. The design principle is to use selective keywords to quickly zoom into a small part of a long list by joining selective keywords with common data types. This avoids traversal on the inverted list of highly frequent keywords or data types. Refer to [11] for the detailed discussion.

Example 3

```sql
SELECT #number, conf() FROM #number
WHERE (pattern("{#number inhabitants}{1.0]"
 OR pattern("{population of "}{0.1] #number}{0.8])
AND ~likeLargeNum(#number)
AND pattern("{China #number}{10}"
)
GROUP BY #number
ORDER BY conf()
```

After rewriting and parsing the query, the index selection module works to generate an optimal query strategy. Due to the special index layer design, the system has multiple choices to implement the query with different time costs. We model the strategy selection problem as a graph coverage problem, the details of which is described in [11].

The execution layer collects the parsing result and the indexing plan, constructing an execution tree for query implementation. The leaves of the tree are nodes interfering with the index layer; while the intermediate tree nodes are a series of operations (e.g., filtering, aggregation, ordering,
During the execution, tuples are “pulled” from leaves, processed through the tree, finally outputted at the root.

3.3 User Interface

The DoCQS system provides users with two interfaces. One is a command line interface which allows expert users to define new data types and execute CQL; another is the application layer based on which we can build some interesting applications. Currently, we build an entity search application for common users to input arbitrary keywords and predefined data types as query (e.g., “USA #president”). These queries will be first translated to a CQL statement by embedding the keyword and data type into a CQL template. For example, if a user inputs “China #population” as query, it will be translated to the CQL statement in Example 2. One can extend the template to adapt to more complex queries.

4. DEMONSTRATION

The system is built on a PC with Dual Xeon 2.8GHz CPU, 2Gb of RAM. The data corpus is 7Gb Wikipedia data including 3 million pages. On the corpus, we pre-extract four basic data types: #number, #person, #location and #organization, based on which other advanced data types could be defined. Figure 3 shows the system interfaces.

4.1 Command Line Interface

Figure 3(a) and 3(b) show the command interfaces for expert users. Using the CQL in Example 1 as input, user can define #population based on #number, shown in Figure 3(a). After saving the definition, the name of the data type will be displayed on the left menu.

With #population defined, users can then execute the CQL query in Example 2 to find the population of China. Figure 3(b) shows the result in a sorted table where the answer “1.3 billion” is ranked at the top with highest confidence.

4.2 Application: Entity Search

Figure 3(c) shows an example application—Entity Search—for common users which is built upon DoCQS system. This search engine receives arbitrary keywords and a specified data type as input. The example query could be “Google #CEO”, “China #population”, “USA #president”, “computer science #professor” etc., where the data type in the query comes from the pre-defined data types. Figure 3(c) shows the result for query “DELL #CEO”, with top result “Michael Dell” being the correct answer. In [11], we evaluated a set of benchmark queries on three data types: #CEO, #capital and #population, the precisions are 65.5%, 90.0% and 67.0% respectively. Due to the small degree of data redundancy of Wikipedia, we expect to achieve higher accuracy if a larger web corpus is used.

5. REFERENCES