DoCQS: A Prototype System for Supporting Data-oriented Content Query

Mianwei Zhou, Tao Cheng, Kevin Chen-Chuan Chang

Computer Science Department, University of Illinois at Urbana-Champaign
{zhou18, tcheng3, kcchang}@uiuc.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://falcon4.cs.uiuc.edu:8080/DoCQS.

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:

Web-based Information Extraction (WIE) Information extraction, with the aim to identify information systematically, has also 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 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 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.

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 and TES usually rely on hard-coded extraction and search strategies, which is 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 pattern has shown its usefulness in Web information extraction, it is 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 the 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 WIE and TES 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. Referring to confidence management in uncertain database, DoCQS embeds the scoring computation in every CQL operation, and users can design the scoring function themselves via CQL specification.

2. SYSTEM OVERVIEW

2.1 Data Model
Because the search target of DoCQS is typed data which is different from traditional IR, 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: 1) doc, the ID of the document where the data type appears; 2) pos, the word position of data type in document; 3) span, the number of keywords that each data type occurrence covers; 4) val, the instance value of data type; 5) conf, the confidence which measures the possibility that the data occurrence belongs to its data type. Without loss of generality, keyword can be regarded as a special data type. Figure 1 shows how the text corpus is transformed to our data model.

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 is composed 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

\[
\text{DEFINE DATATYPE } T_{\text{new}} \text{ AS } T_{\text{ori}} [\text{expr AS conf}] \text{ WHERE } \text{condition}
\]

where \text{condition} :=

\{ \text{bool Func|fuzzy Func} \} \text{[w]} \{ [\text{AND|OR} \text{ condition}] \}

The above CQL defines a new data type \( T_{\text{new}} \) based on the basic type \( T_{\text{ori}} \) 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

\[
\text{DEFINE DATATYPE } \#\text{Population AS } \#\text{Number WHERE } \{ \text{pattern("\#\text{Number inhabitants}\}])[1.0] OR \text{pattern("\{population of ?(0,1) \#\text{Number}\}])[0.8] \text{AND ~likeLargeNum(#Number)} \}
\]

Example 1 defines Population data type based on Number. The definition uses the boolean function \text{pattern}() to indicate that Number data type before “inhabitants” or after “population of” is likely to be Population. Considering the property of Population data type, ~likeLargeNum(), as a fuzzy function, assigns higher confidence to larger numbers. 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 DoCQS system are highly extensible by supporting a wide range of patterns. In addition to the simple sequential pattern\((P_1; P_2 \ldots P_n)\) described above, our system also supports other types of patterns: Window Pattern\([P_1; P_2 \ldots P_n]([m])\), Disjunction Pattern\([P_1; P_2 \ldots P_n]\) and Inner Pattern\(P_1; P_2\). Each pattern can take as input any keywords, data types or other nested patterns. Our system also supports flexible user-defined functions (e.g., \( \sim\text{likeLargeNum}() \)).

2.2.2 Data Retrieval Operation

The data retrieval operation takes the following specification:

\[
\text{SELECT } T_{\text{t}}[., . . . T_{n}.\text{expr1}, . . . \text{exprn}] \text{ FROM } T_{\text{1}}, . . . , T_{n} \text{ WHERE condition } \{ \text{GROUP BY } T_{\text{k}.\text{val}} \} \{ \text{ORDER BY expr} \}
\]

With Population data type defined, one concrete example is shown in Example 2 to find the population of China:

Example 2

\[
\text{SELECT } \#\text{Population, conf()} \text{ FROM } \#\text{Population WHERE pattern("\{\#\text{Population}\}])[10]" \text{ GROUP BY } \#\text{Population ORDER BY conf()}
\]

In the data retrieval operation specification, the FROM clause lists the source data types. They are natural joined by the doc attribute. For conciseness we use “,” instead of \( \bowtie \text{doc} \). The WHERE clause refines the retrieved tuples (in Example 2, it only collects Population data type with keyword “China” around in a 10-word window). They are further aggregated by the GROUP BY clause which assigns high confidence to ones with high frequency. Finally, the ORDER BY clause ranks the result by the given expression.

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 Named Entity Recognition(NER) tool 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. That’s because the most common table operation in CQL
4. Demonstration

The system is built on a PC with Dual Xeon 2.8GHz CPU, is to transverse the keyword or data type tables, and efficiently supporting the transversal operation is the distinct advantage of inverted index.

However, in the experiment, we find that some keywords or data types have very long index lists (e.g., Number data type is everywhere in any data corpus), a basic inverted index structure could not satisfy the performance requirement of online query. 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 transversal on the inverted list of highly frequent keywords or data types. Refer to [1] 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-basic data types, and combine the original condition with their definitions. For instance, with the Population data type defined by Example 1, the query in Example 2 will be rewritten as

```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 [1].

The execution layer collects the parsing result and the indexing plan, constructing an execution tree for query implementation. The leaves of the trees are nodes interfering with the Index Layer; while the intermediate tree nodes are a series of operations (e.g., filtering, aggregation, ordering, etc.). 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 Command Line Interface which allows expert users to define new data types and execute CQL; another is the application layer beyond 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 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. REFERENCES
