Data-oriented Content Query System: Searching for Data into Text on the Web

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

ABSTRACT

As the Web provides rich data embedded in the immense contents inside pages, we witness many ad-hoc efforts for exploiting fine granularity information across Web text, such as Web information extraction, typed-entity search, and question answering. To unify and generalize these efforts, this paper proposes a general search system—*Data-oriented Content Query System* (DoCQS)—to search directly into document contents for finding relevant values of desired data types. Motivated by the current limitations, we start by distilling the essential capabilities needed by such content querying. The capabilities call for a conceptually relational model, upon which we design a powerful *Content Query Language* (CQL) . For efficient processing, we design novel index structures and query processing algorithms. We evaluate our proposal over two concrete domains of realistic Web corpora, demonstrating that our query language is rather flexible and expressive, and our query processing is efficient with reasonable index overhead.

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, Algorithms, Performance, Experimentation

Keywords

content query, data oriented, content query language, inverted index, joint index, contextual index

1. INTRODUCTION

With the ever growing richness of the Web, people nowadays are no longer satisfied with finding interesting documents to read. Instead, we are becoming increasingly interested in the various fine granularity information units, *e.g.*, movie release date, book price, the typical linguistic usages of certain phrases, *etc.*, which appear within the content of Web documents. Indeed, often our information needs boil down to looking for small pieces of information embedded in documents. Towards exploiting such rich data on the Web, we witness several emerging Web-based search applications:

Web-based Information Extraction (WIE) Information extraction, with the aim to identify information systematically, has also naturally turned to Web-based, for harvesting the numerous "facts" online—*e.g.*, to assemble a table of *all* the ⟨country, capital⟩ pairs (say, ⟨France, Paris⟩). Recent WIE efforts (*e.g.*, [8, 3, 11]) have mostly relied on phrase patterns (*e.g.*, "X is the capital of Y") for large scale extraction. Such simple patterns, when coupled with the richness and redundancy of the Web, can be very useful in scraping millions or even billions of facts from the Web.

Typed-Entity Search (TES) As the Web hosts all sorts of data, several efforts (*e.g.*, [6, 5, 7, 4]) proposed to target search at 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.*, [2, 10, 14]) 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.

With so many ad-hoc efforts exploiting Web contents, such as WIE, TES, and WQA, there is a pressing need to distill their essential capabilities—We thus propose the concept of *Data-oriented Content Query System* (or DoCQS[1]), for generally supporting "content querying" for finding data over the Web. To motivate, we observe that, as their targets, those new applications all share one key objective—to search into the *content* on the Web to explore its rich data of all kinds—much beyond finding *pages* as in traditional Web search. As their functional requirements, these applications can be distilled into the following key capabilities:

Extensible Data Types: With their focus on fine grained information, all such applications target at data entities of various types. As different applications will need different types, a general system must support type extensibility, to build specialized data types upon existing ones, in a declarative manner. *E.g.*, we can "specialize" #number into #zipcode, #population, or #price in an online fashion. Note that, to distinguish from keywords, we prefix # for data types.

[1]DoCQS is pronounced as dokis.

WSDM'10, February 4–6, 2010, New York City, New York, USA.

Flexible Contextual Patterns: All these applications recognize desired data by its surrounding textual patterns. While phrase patterns are useful, it is limited to scenarios where sequential words can be completely specified. A general system should utilize all the available information that appears in the context of target information. Thus, the ability to support flexible, expressive contextual patterns, beyond simple phrases, is mandatory.

Customizable Scoring: To find target data, all these applications perform scoring on candidate answers, in their specific ways. A general system must support the customization of rank scoring to meet various domains, in the following aspects:

- *Weighting*: As evidences come from matching various patterns or multiple rules, we must be able to weigh those evidences to favor more confident ones.
- *Aggregation*: To account for evidences from everywhere—by the immense redundancy of the Web—it is crucial to aggregate information. We thus need customizable aggregation to apply different strategies for different applications.
- *Ranking*: Essential in any search system, we need ranking for presenting results in an ordered manner so that the most promising results appear at top places.

For these capabilities, we must generalize their support from current limited realization in various ad-hoc efforts, which rely on a restrictive set of fixed data types, simplistic phrase patterns, and hard-coded scoring functions. We thus define our DoCQS proposal:

DoCQS Definition: A *Data-oriented Content Query System* supports users, with respect to a corpus of documents such as the Web, to use keywords or data types to query for relevant values of their desired data types in the contents of the corpus, by specifying flexible patterns and customizing scoring functions.

To realize a general DoCQS, this paper proposes its corresponding content query language, CQL, and present the underlying indexing and processing framework, with the following contributions:

- We propose the concept of a general content query system, by distilling its essential requirements.
- We design a flexible query language CQL for content querying.
- We develop indexing design and query processing for the efficient support of DoCQS.
- We validate with extensive experiments over *realistic Web corpora* in concrete applications.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the data model for the DoCQS system. In Section 4, we present our design of the content query language. Section 5 introduces indexing framework. We present the experimental evaluation of our system in Section 6 before concluding in Section 7.

2. RELATED WORK

Towards searching fine granularity data, there are several recent systems (*e.g.*, [6, 7, 9, 5]). Chakarabarti et al. [6] propose to search for annotations using proximity in documents. *EntityRank* [7] proposes the problem of entity search, and studies a probabilistic ranking model. NAGA [9] builds a knowledge base over extracted entities and relationships and supports flexible query over it. All these works leverage off-the-shelf extraction tools for extracting entities from text, and thus support only a fixed set of data types. Our DoCQS provides a framework for type extensibility— specializing basic data types into specific data types in an ad-hoc manner, and therefore greatly extending the application scenarios.

Our work aims at designing a language and framework to support extracting and querying of fine grained data holistically over the entire corpus. Some recent works (*e.g.*, [13]) address "declarative" mechanisms for information extraction, proposing languages or algebra to specify information extraction. We note that these attempts take a *document-based* approach, in that their operations apply to one document at a time. Our framework performs extraction and search *holistically* over the whole corpus.

In implementation, in terms of our indexing techniques, our work is related with existing work on using inverted index for efficient entity extraction. Our proposal is essentially supporting SQL-like DB queries, over IR type of indexes. Ramakrishnan et.al [12] propose to use inverted index to support regular expression extraction. Agrawal et al. [1] study the problem of large-scale extraction of dictionary based entities using keyword inverted index. Our work goes beyond traditional inverted index, by building specialized indexes between keywords and data types. Therefore, as we show in Section 6, our solution achieves much better efficiency compared to such standard inverted index approaches. BE engine [3] proposes "neighborhood index" to efficiently support extraction based on phrase patterns. As the name indicates, it is limited to only extraction based on phrase patterns. Our DoCQS is flexible in supporting a wide variety of context operators, thus allowing extending new data types from basic data types. Further, our experiments also show that our indexing is more efficient than the neighborhood index.

3. DATA MODEL

With the search target of various data types inside documents, 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 need to come up with a new data model, to meet the various capabilities of a content query system (Section 1).

First, our search target is data types. Each type captures a domain of values (*e.g.*, prices). In the contents of a Web corpus, each value has multiple occurrences. To query data types, we need to record each occurrence with the corresponding instance value, document, and position—*i.e.*, a "tuple" of occurrence information. We can naturally represent such tuples in a relational table.

Second, the content query system calls for flexible pattern matching, as well as customizable scoring of weighting, aggregation, and ranking. These operations go much beyond the standard IR operations of containment check and relevance-based scoring of individual documents. In fact, pattern constraints, aggregation, and ordering are concepts more widely used in relational operations.

Indeed, the content-querying capabilities consistently call for a relational model, which supports the modeling of data types as relations, and the various relational operations for pattern matching, aggregation, and ranking (WHERE, GROUP BY, ORDER BY). Thus, taking relational modeling, we conceptualize the E-R diagram for the entities and relationships in our DoCQS in Figure 1.

In this framework, there are three types of entities in the E-R diagram: document D, built-in data type T_i (for the set of built-in data types, *e.g.*, #number, #organization) , and keyword K_j (for all keywords, *e.g.*, "Chicago"). Note that we can view both T_i and K_j as *Is-A* data type—since we can consider a keyword as a data type with a specific literal value (*e.g.*, "Chicago"). Each document can be identified by its unique document ID. Each *occurrence* of a data type (*e.g.*, #person) can be identified by a unique *ID*, its specific instance *value* (*e.g.*, "Bill Smith"), *confidence* (*e.g.*, 0.9 probability), *span* (*e.g.*, a span of 2 words).

As relationship, a data type occurs at the content of some document, at a certain position *pos*. Each *occurrence* is unique and can

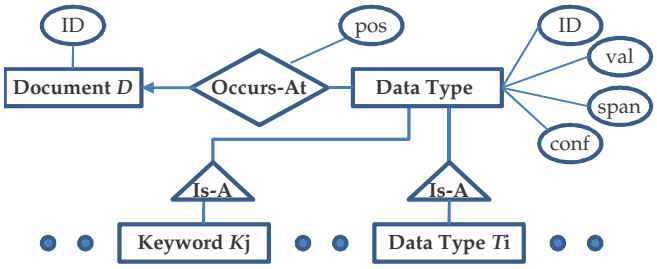
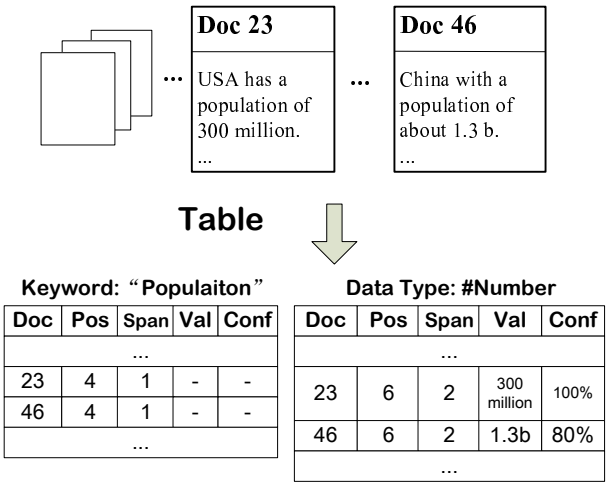


Figure 1: ER diagram for the proposed content query system.



Doc	Pos	Span	Val	Conf
...				
23	4	1	-	-
46	4	1	-	-
...				

Doc	Pos	Span	Val	Conf
...				
23	6	2	300 million	100%
46	6	2	1.3b	80%
...				

Figure 2: Data model: Relations for keywords and data types.

only appear at one position of a document. Therefore, the *Occurs-At* relationship between data type and document is *many-to-one*. As mentioned earlier, keyword is simply a special data type, whose value is always the keyword itself with a span of 1 .

With the ER diagram outlined, we are ready to come up with the schema of our relational model. We notice, with the many-to-one relationship between data type and document, we can combine the two entities into one relation. Moreover, we also realize that a specific document *ID* with a position pos can uniquely define an occurrence of a data type, therefore eliminating the need of the ID attribute of the data type. We can thus define the schema for two types of relations (keyword and data type) as follows:

- *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).
- *conf*: The confidence, which measures the probability that the data occurrence belongs to its data type. As data extraction is inherently imperfect, the confidence may not be 100%.

We note that the two types of relations will materialize to many tables: In the overall schema, we are dealing with M data type relations noted as $T_1, T_2, \ldots, T_M$, as well as N keyword relations noted as $K_1, K_2, \ldots, K_N$. Figure 2 shows an example in turning a text corpus into our relational model, with two concrete relations illustrated, one for keyword "population" and one for #number.

4. CONTENT QUERY LANGUAGE (CQL)

This section will discuss our design of the Content Query Language (CQL) to serve the need of DoCQS. We will first reason the general form of the query language, based on the capabilities needed and the relational model derived. Then we will present the overall general specification of the query language, followed by in-depth discussion of each component of the query language.

4.1 Design Principle

With the relational model chosen as our data model in Section 3, we now discuss the design of our query language based on the relational model.

While SQL is the widely accepted query language over relational model, with the special need of a content query system, our goal of the design is to customize the query language to meet the various requirements of DoCQS. Therefore, we propose our own query language CQL, which takes a special form of SQL with several new constructs for the need of querying data types inside document content, with examples shown in Figure 3.

Based on the ER diagram in Figure 1, a CQL statement targets at retrieving tuples from the relational tables, which are transformed from the text corpus as shown in Figure 2. Specifically, each retrieval statement involves several steps: specifying source tables, joining tables, filtering tuples, aggregating, and finally ranking results. Consider the example query **Q1** in Figure 3(a), which aims at deriving ⟨location, population⟩ pairs from documents. It first joins the #location and #number tables by document ID, utilizes conditions (*e.g.*, a sequential pattern {#location has population of ?⟨0,3⟩ #number}) to retrieve ⟨location, population⟩ pairs, aggregates those pairs to estimate the confidence based on the redundancy of potentially correct answers, and finally ranks the results by confidence. To customize for these operations, CQL takes a special form of **SELECT** . . . **FROM** . . . **WHERE** . . . **GROUP BY** . . . **ORDER BY** . . . . We now discuss in detail the functions of each clause.

First, in the SELECT clause, we are interested in retrieving relevant *values* of target data types. This is supported by having the SELECT clause to project onto the *val* attribute of data types.

Second, in the FROM clause we specify all the participating keyword and data type relations for the query. These are the basic relations upon which our query operates.

Third, our WHERE clause will support fuzzy text oriented patterns. We will introduce our own intuitive pattern syntax (*e.g.*, "*pattern*([professor #person]⟨10⟩)") to hide the detailed relational data operations in the background.

In addition to the standard SQL constructs, with the need to differentiate the importance of different patterns, as well as the need to judge how well each pattern is matched, our WHERE clause will accommodate a weighting scheme of the patterns, in addition to the specification of a series of text-oriented patterns.

Moreover, with the need to define specialized data types based on existing data types, our query language offers a special data type definition construct, which is inspired by the "view" concept in databases.

Finally, CQL supports a plugin function library, with a set of functions provided by default. The library contains various text retrieval measures such as term frequency, PageRank, *etc*. Such functions could be utilized for flexible customization in weighting and ordering.

4.2 CQL Specification

We now introduce the general form of our content query language. Similar to the data retrieval and manipulation aspects in

```
SELECT #location.val, #number.val
FROM #location, #number
WHERE                                        //w1 = 0.95, w2 = 0.8
    (P1=pattern("{#location has population of ?⟨0, 3⟩
                  #number}")∧
    P2=∼LikeLargeNum(#number))
    WITH P1.score ∗ P2.score ∗ w1
  ∨
    (P3=pattern("[#location population #number]⟨6⟩")∧
    P4=∼LikeLargeNum(#number))
    WITH P3.score∗P4.score∗w2
GROUP BY #location, #number
    WITH 1 − ∏(1−conf)
ORDER BY conf()
```

(a) Query **Q1**: Data retrieval with explicit-scoring.

```
SELECT #location.val, #number.val
FROM #location, #number
WHERE (                                      //w1 = 0.95, w2 = 0.8
    pattern("{#location has population of ?⟨0, 3⟩
              #number}")[w1]
    OR pattern("[#location population #number]⟨6⟩")[w2]
  )AND ∼LikeLargeNum(#number)
GROUP BY #location, #number
ORDER BY conf()
```

(b) Query **Q2**: Data retrieval with implicit-scoring.

```
DEFINE DATATYPE #GDP AS #number
WHERE pattern("[GDP #number]⟨10⟩")
   AND ∼LikeLargeNum(#number)
```

(c) Query **Q3**: Data type definition.

Figure 3: Some example CQL queries.

SQL, we define two types of operations in CQL, data retrieval operation and data type definition. We now describe their syntax.

Data Retrieval Operation To retrieve relevant data values of data types $T_1, \ldots, T_n$.

SELECT T_1[.*], ..., T_n[.*]
FROM $T_1, \ldots, T_n$
WHERE *condition*
[**GROUP BY** T_k[.*val*] [**WITH** *GS*(*conf*)]]
[**ORDER BY** *expr*]

where *condition* in the WHERE clause can take the following two forms:

1) Explicit-scoring condition
condition:=*pattern*$_1 \wedge \ldots \wedge$ *pattern*$_l$ **WITH**
LS(*pattern*$_1$.*score*, ..., *pattern*$_l$.*score*)
($\vee$ *condition*)*
2) Implicit-scoring condition
condition:=*pattern*[*w*] ({**AND**|**OR**} *condition*)*

In the above specification, T_i refers to a specific data type and $T_i.*$ indicates $\langle T_i.doc, T_i.pos, T_i.span, T_i.val, T_i.conf \rangle$. If no attribute is specified, "*.val*" is used for default because we are interested in the value of the data type in most cases. The FROM clause lists all the data-type tables involved; it omits the keyword tables (for simplicity) since their values are not of interest. The data type relations in the FROM clause are natural-joined by the *doc* attribute. For conciseness we use "," instead of $\bowtie_{doc}$.

To meet the query demands of content query tasks, the data retrieval operation allows users to customize their scoring function in the WHERE clause for measuring how well the given patterns are matched over individual documents, and in the GROUP BY clause for measuring how frequent results appear over the data corpus. To accommodate the different semantics of these two measures, the query language supports two scoring functions: *LS* in the WHERE clause for local scoring within a document, and *GS* in the GROUP BY clause for global scoring across documents.

By default, for users who do not want to specify detailed scoring functions, we support the implicit-scoring condition, which eliminates the need of explicit specification of the *LS* function in the WHERE clause and the *GS* function in the GROUP BY clause. In this case, the default *LS* and *GS* functions are implicitly applied. We will discuss this aspect in more details in Section 4.4.

Data Type Definition To derive specialized data type T_{new} from an existing base data type T_{base}.

DEFINE DATATYPE T_{new} **AS** T_{base}
WHERE *condition*

The data type definition is used for extending basic data types into special data types, such as defining #population from base type #number, as we motivated in Section 1. Derived data types will have the exact same schema as the basic data types, and can therefore be seamlessly used in the same way as the basic data types for retrieval. As the above syntax shows, we currently limit the definition to be based on one existing data type only, which captures most cases in practice.

With the general specification of CQL outlined, we now discuss the special constructs in the language that are tailored to the need of a data-oriented content query system. Specifically, we will zoom into the specification and weighting of patterns, the customization of scoring functions, and the definition of new data types.

4.3 Pattern & Weighting

The **WHERE** clause is used to select those tuples satisfying the indicated conditions. This section first introduces the specification of the pattern conditions, followed by the weighting and scoring scheme of conditions.

Pattern conditions play a key role in selecting desired results. For example, we can utilize pattern conditions to select numbers following phrase "population of", or person names near keywords "professor" or "doctor." By hiding relational operations (*e.g.*, joining a series of keyword and data type tables by some conditions) from users, these pattern conditions provide an intuitive interface for users to describe flexible patterns. In our framework, we support four basic pattern conditions, and they can be further extended for different application demands.

• Sequential Pattern
Syntax: $\{X_1\ X_2\ \ldots\ X_n\}$

Matched when keywords or data types X_i appear in sequence (*e.g.*, {#location has population of ?⟨0,3⟩ #number} in Figure 3(a)). It is defined as:

$$\sigma_{\bigwedge_i (X_i.pos + X_i.span = X_{i+1}.pos)}(X)$$

where a *term* X_i stands for any keyword or data type relation, and X is shorthand for their natual joins, *i.e.*,

$$X_1 \underset{X_1.doc = X_2.doc}{\bowtie} X_2 \ldots X_n.$$

Notice, X_i can also be a nested pattern relation, which allows the combination of different patterns. Wildcard is also allowed in the sequential pattern. In the example above, the wildcard ?⟨0,3⟩ allows 0 to 3 words (*e.g.*, "almost", "close to") between "of" and #number. In such cases, the above position constraint would be-

come $lower \leq X_{i+1}.pos - (X_i.pos + X_i.span) \leq upper$, when wildcard ?$\langle lower, upper \rangle$ is inserted between X_i and X_{i+1}.

• Window Pattern
Syntax: $[X_1, X_2, ..., X_n]\langle m \rangle$

Matched if all keywords/data types/patterns appear within an m-words window. It is defined as:

$$\sigma_{\bigwedge_{i,j}(X_i.pos+X_i.span-X_j.pos<=m)}(X)$$

Notice, a pattern could also be nested in other patterns, for example, [#number {United States}]⟨10⟩. In this case, X_2 is a pattern whose *pos* and *span* is not readily available. Online computation will generate X_2.*pos* as United.*pos* and X_2.*span* as (States.*pos* + States.*span* - United.*pos*). Finally, the conditions of the main pattern and nested patterns are connected by conjunction.

• Disjunction Pattern
Syntax: $(X_1|X_2|...|X_n)$

Matched if any keyword/data type/pattern in the list is matched, for example, {#number (people|inhabitant)}. It is defined as:

$$X_1 \cup X_2 \cup ... \cup X_n$$

• Inner Pattern
Syntax: $(X_o : X_i)$

Matched if X_o is matched and X_i lies in the scope of X_o, for example, pattern "(#organization:university)" returns #organization instances that contain the "university" keyword. It is defined as:

$$\sigma_{X_i.pos>=X_o.pos \bigwedge X_i.pos+X_i.span<=X_o.pos+X_o.span}(X)$$

These four basic pattern conditions are designed based on common content query demands. Sequential patterns focus on extracting data types surrounded by specific phrase context words (*e.g.*, numbers following phrase "population of" are highly possible to be of the population data type). Window patterns could be used to locate instances of data types of certain topics (*e.g.*, finding the inventor of TV near words "television", "invention"). Disjunction patterns allow data types to be generated by different patterns. Inner patterns can retrieve data types with specific content (*e.g.*, a #location containing "university" is likely to be the #university type).

In addition to the pattern conditions, which can be viewed as pre-defined Boolean functions, we also allow other user-defined Boolean or fuzzy functions in the WHERE clause. The difference between Boolean and fuzzy functions is that the former filters the result by its returned Boolean value, while the latter does not perform filtering but instead assigns scores to tuples by how well the condition is matched. For instance, we can use a Boolean function *IsYear*(#number) to select year-formatted numbers or use a fuzzy function ∼*LikeLargeNum*(#number) to favor large numbers.

Weighting of conditions is a new concept, which does not exist in SQL. Its function is to obtain the weights of conditions specified in the WHERE clause. First, we need to distinguish the importance of different patterns (noted by w_j), as we may have more confidence in one pattern condition over another. Second, we need to examine how well the pattern is matched (noted by $P_i.score$ for pattern P_i). For Boolean functions, the score is either 0 or 1, while for fuzzy functions, the score is a real value between 0 and 1 reflecting the matching degree. These two factors contribute to the final weight of a pattern.

Query **Q1** in Figure 3(a) shows how to customize the weighting of patterns in detail. First, the w_j parameter allows us to give different weights to different patterns. For instance, we can set w_1, w_2 to be 0.95, 0.8 for two conjunctive patterns $P_1 \wedge P_2$ and $P_3 \wedge P_4$ respectively (where a stronger conjunctive condition like $P_1 \wedge P_2$ gets higher weight). Then $P_i.score$ captures how well a pattern P_i is matched. For instance, $P_1.score$ outputs 1 if P_1 is matched, and 0 otherwise. For P_2, which is a fuzzy pattern, the score is a probabilistic value measuring the likelihood of a large number (*e.g.*, if the number value is greater than 1 billion, $P_2.score = 1$; if it is less than 1 billion more than 1 million, $p_2.score = 0.8; \ldots$).

4.4 Scoring Specification

As we have hinted briefly, there are two levels of scoring specification in our CQL.

First, local scoring LS is in charge of judging the local score of a specific tuple occurrence within a document. It takes as input the matching scores of patterns, and we can assigns weights to the patterns by multiplying their scores and the weights w_j in the formula. In addition, it can also take individual attribute values of the participating relations (*e.g.*, their *conf* values). It outputs a score *conf*, which indicates the confidence over the matched tuple occurrence. For instance, **Q1** uses $P_1.score{*}P_2.score{*}w_1$ as the local scoring function for the conjunction result of P_1 and P_2.

Second, global scoring GS is used for calculating the score of a specific group of tuple occurrences that share the same data values. This is where the aggregation comes into play, since the occurrences of the same group come from different pages and therefore their scores need to be put together. This global function normally takes into account the local confidence scores of the matched occurrences. It can also use other common information retrieval functions (like TF-IDF, Pagerank, *etc.*). For instance, Figure 3(a) uses $1 - \prod(1 - conf())$ as the aggregation expression, which means that given n tuples with confidence $conf_1, \ldots, conf_n$, the aggregated confidence will be $1 - \prod_{i=1}^{n}(1-conf_i)$.

For the ease of users who do not want to fine tune the detailed scoring function, we provide implicit scoring to eliminate the need of explicit specification of the LS scoring function in the WHERE clause and the GS scoring function in the GROUP BY clause, such as query **Q2** in Figure 3(b). In this situation, default *LS* and *GS* functions will be applied for scoring. Specifically, the default *LS* and *GS* functions take the following forms:

- *LS* for **AND** operator: Given $(P_1$ **AND** $P_2)[w]$, the confidence of the generated result is
  $conf_{LS}((P_1 \textbf{ AND } P_2)[w]) = P_1.score{*}P_2.score{*}w.$
- *LS* for **OR** operator: Given $(P_1[w_1]$ **OR** $P_2[w_2])$, the confidence is $P_1.score{*}w_1$, if the result tuple comes from matching P_1, and similar for P_2.
- Default *GS*: Given n tuples with confidence $conf_1, \ldots, conf_n$, $conf_{GS} = 1 - \prod_{i=1}^{n}(1-conf_i)$.

With the default *LS* and *GS*, query **Q2** is equivalent to query **Q1**. For conciseness, we will write CQL examples using the implicit scoring format for the rest of the paper.

4.5 Data Type Definition

View, in the relational database sense, refers to a stored query, which is accessible as a virtual table. In DoCQS, we borrow this concept to define new data types over existing ones. A defined data type has the same schema as the basic data type, and can be accessed later like the basic data types with no difference. The confidence values of a new data type comes from the matching scores of various conditions. Users can customize the scoring of confidence by specifying the *LS* function in the WHERE clause.

Query **Q3** in Figure 3(c) defines the #GDP data type, by identifying large numbers with keyword "GDP" around them within a 10-word window. During execution, the references of new data types will be translated into the stored data type definitions and get executed accordingly. As an alternative, our system also allows the

materialization of new data types and stores them on disk, which can be used directly in subsequent queries with faster response.

5. INDEXING & QUERY PROCESSING

This section discusses the indexing design, configuration and query processing of the DoCQS system. The data-oriented characteristics require a very different index architecture compared with relational tables used in traditional DBMS. In DoCQS, we mainly rely on IR style inverted indexes and present an effective index selection algorithm for query processing.

5.1 Indexing Design

We utilize the gist of inverted index as the essential indexing structure for the DoCQS system. Although keywords and data types are conceptually modelled as tables in our system for the convenience of defining the CQL language, we choose IR style inverted index as the underlying data structure since the most common operation of CQL is to traverse the keyword or data type tables. Sequential access of inverted index is well-known to be efficient for supporting traversal operations. For balancing processing efficiency and index space, we propose two layers of indexes in the DoCQS system: the basic inverted list layer and the advanced inverted index layer. We now zoom into these two layers.

5.1.1 Basic Inverted List Layer

The basic inverted list layer stores all information of keyword and data type tables defined in the data model (shown in Figure 2) in sequential lists. For keyword K_i, we build the traditional inverted list $I(K_i)$; while for data type T_j, its inverted list $I(T_j)$ also stores information about the other attributes (*e.g.*, *doc*, *val*) in addition to position information. Take #phone as example, its inverted list is of the form: $I(\text{\#phone}) \rightarrow \{\langle doc{:}1, pos{:}10, val{:}\text{“123-456-7890”}, ... \rangle, \langle doc{:}20, pos{:}13, val{:}\text{“789-456-1230”}, ... \rangle, ... \}$.

With the basic inverted list layer defined above, CQL could be already executed (though not efficient, as we will improve later). Consider a simple query example **Q4**:

SELECT #number.val
FROM #number
WHERE *pattern*(“{population of $?\langle 0, 3\rangle$ #number}”)

To execute the query, the DoCQS system first loads three inverted lists: I(“population”), I(“of”), and I(#number). The pointers to the three lists are incremented for checking intersecting documents. Once an intersecting document is identified, the system retrieves the postings of this document from the three lists, and joins them by the sequential pattern specified to produce the matching tuples. This execution ends when one of the lists is exhausted.

5.1.2 Advanced Inverted Index Layer

Although the basic inverted list layer could already support CQL execution, query performance remains a significant issue. In query **Q4**, “of” and #number appear almost in every document, which means very long inverted lists. It is very expensive to load and scan such long lists. Moreover, we can not discard such high frequency keywords (*e.g.*, “of” “such”) as stop words, because they play concrete roles in describing interesting patterns. We need another index layer, the Advanced Inverted Index Layer, to further expedite query processing.

The design principle of the advanced inverted index layer in DoCQS is inspired by the index layer in databases. Traditional DBMS utilizes B+ Tree to speed up query processing using conditions in the selection clause. Consider a SQL query to retrieve the name of students who are over 18 years old. With index built for the age attribute, the system could directly get the tuples satisfying $age > 18$, without scanning through the whole table. Similarly, the basic idea of the advanced inverted index layer is to record redundant information and utilize pattern conditions to avoid traversal over the inverted list of highly frequent keywords or data types. Again consider query **Q4** in Section 5.1.1. The query execution at the basic inverted list layer is basically a join operation over I(“population”), I(“of”), and I(#number). If the pair $\langle$“population”,“of”$\rangle$ and $\langle$“population”,#number$\rangle$ have been pre-joined offline, the online execution only needs to traverse these two joined-lists, which are much shorter than I(“of”) and I(#number), thus greatly saving the query processing time. With this insight, we propose two kinds of special inverted index structures: Joint Index and Contextual Index, with examples shown in Figure 4.

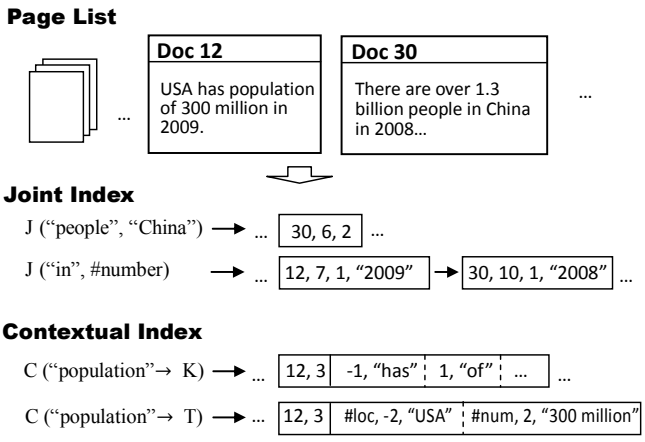


Figure 4: Advanced inverted index layer.

Joint Index The main idea of “Joint Index” is to pre-join two terms (which could be keywords or data types), and store their co-occurrence pairs within W-word window, where W could be configured by the system. As shown in Figure 4, a text segment like “There are over 1.3 billion people in China in 2008” is given. With W set to 5, since the distance between keywords “people” and “China” is 2 ($< W$), we record their co-occurrence information $\langle doc{:}30, pos{:}6, offset{:}\ 2\rangle$ in the joint index J(“people”,“China”), and we similarly construct J(“in”,#number). When DoCQS receives a query like “[China people]$\langle 5\rangle$”, it can use the joint index J(“people”,“China”) to greatly improve processing time because the size of the J(“people”,“China”) list is much shorter than either I(“people”) or I(“China”).

Formally, a joint index could be viewed as the pre-joining of two tables. Consider two terms X_i and X_j, whose corresponding tables (stored as inverted lists) are $I(X_i)$ and $I(X_j)$. Their joint index $J(X_i, X_j)$ is:

$$J(X_i, X_j) = I(X_i) \underset{|I(X_i).pos - I(X_j).pos| < W}{\bowtie} I(X_j)$$

For queries such as $\{X_i\ ?\langle 0, w-2\rangle\ X_j\}$ and $[X_i, X_j]\langle w\rangle$ ($w <= W$), the query time complexity is reduced from $O(I(X_i)+I(X_j))$ (using basic inverted lists) to $O(I(X_i) \cap I(X_j))$ (using joint index).

Although such joint indexes significantly reduces query time for a large number of patterns, due to their high space cost, we can not afford building the joint index for every pair of terms in the whole corpus. The extra space cost comes from the large number of term pairs, *i.e.*, N^2 (N is the number of terms in the corpus). To alleviate the high space cost of joint index, we next propose another type of more space efficient inverted index.

Joint Index	Description
$J(K_i, K_j)$	Keyword K_i with keyword K_j.
$J(K_i, T_j)$	Keyword K_i with data type T_j.
$J(T_i, T_j)$	Data type T_i with data type T_j.
Contextual Index	**Description**
$C(K_i \rightarrow K)$	From keyword K_i to keywords in K.
$C(T_i \rightarrow K)$	From data type T_i to keywords in K.
$C(K_i \rightarrow T)$	From keyword K_i to data types in T.
$C(T_i \rightarrow T)$	From data type T_i to data types in T.

Figure 5: All the index types in the advanced index layer.

Contextual Index The contextual index uses each term as the index key and stores its surrounding terms within its W-word context window. In Figure 4, given a text segment "USA has population of 300 million in 2009", the occurrence of "population" at ⟨doc: 12, pos: 3⟩ is recorded in its the contextual index C("population" $\rightarrow K$) (where K refers to keywords in its context), together with the information of its surrounding words (*e.g.*, ⟨val: "has", offset: -1⟩, ⟨val: "of", offset: 1⟩). For a query with sequential pattern {population of}, the system only needs to traverse C("population" $\rightarrow K$) instead of both I("population") and I("of").

Conceptually, a contextual index for term X_i is the union of a series of joint indexes sharing the term. Given term X_i, its contextual index $C(X_i \rightarrow X)$ is defined as:

$$C(X_i \rightarrow X) = \bigcup_{X_j \in X} I(X_i) \underset{|I(X_i).pos - I(X_j).pos| < W}{\bowtie} I(X_j)$$

where X represents keywords K or data types T in the W-word window context of X_i.

For queries such as $\{X_i$?⟨0,w-1⟩ $X_j\}$ where $w < W$, the time complexity is $min(O(C(X_i \rightarrow X)), O(C(X_j \rightarrow X)))$, which means the system will choose to use the shorter list between $C(X_i \rightarrow X)$ and $C(X_j \rightarrow X)$ for answering the query.

Compared with joint index, the major advantage of contextual index is its relatively low space cost, because we only need one list for each term, thus at most N (the number of terms) lists. We can also leverage inverted-list compression techniques to efficiently store a term's context, since terms in the context appear in a sequential order. Empirically, we observed that, to fully build contextual indexes over a data corpus, it needs only 2-3 times the size of the standard inverted lists (with W set at 5). However, if both terms X_i and X_j have very long lists, their contextual indexes will also be long, and therefore the traversal of long lists can not be avoided.

To summarize, we show all the possible types of indexes in Figure 5 according to our index design. We next discuss how to configure index among these different types.

5.2 Index Configuration

Based on the index structure discussed above, this section introduces the detailed index configuration to achieve high query performance with reasonable space cost overhead.

To better understand the demand of query optimization in real query, we start with analyzing the actual joining time cost of inverted index lists of different length. Using the Wikipedia dataset with 3 million pages (Section 6), we first categorized words into three classes: high-frequency words (top 1,000 frequent words, including common words like "of", "one", "people"), low-frequency words (6,332,031 words that appear in less than 100 documents, including some rare terminology and noise words), and medium-frequency words (132,888 words, including less-common words like "CEO", "flatiron"). We then sampled 50 high-frequency words, 300 low-frequency words, and 300 medium-frequency words. For each pair of words from the sample, we performed the join of their inverted index lists to find their co-occurrences. The average join time is shown in Figure 6.

	High Freq	Medium Freq	Low Freq
High Freq	**1107 ms**	**382ms**	48 ms
Medium Freq	**382ms**	20 ms	14 ms
Low Freq	48 ms	14 ms	13 ms

Figure 6: Average joining time cost.

As we observe, from Figure 6, the most time consuming join operations (whose times are shown in bold-face) happen between high-frequency words with medium or high-frequency words. Nevertheless, we also empirically observed that such combinations are used most often in query patterns (*e.g.*, in {population of ...}, "population" is medium and "of" is high-frequency terms).

Therefore, our index configuration puts priority towards optimization for such expensive joins. We notice that it is not necessary to optimize for low-frequency terms because it takes little time to join with any other terms. Mature inverted index implementations (*e.g.*, Lucene) normally support efficient document skipping, which could directly jumps to a targeted document in the inverted index during traversal. Based on this feature, a join operation between a low-frequency word (*e.g.*, 10 documents) with a high-frequency word (*e.g.*, 1 million documents) takes at most 10 random accesses over the long inverted list, which is affordable. The time statistics in Figure 6 also indicate that it is not crucial to optimize joins between medium-frequency and medium-frequency words.

As the insight of Figure 6 inspires, we choose to build joint indexes between high-frequency terms, and contextual indexes from medium-frequency terms to high-frequency terms. This choice accounts for tradeoff between time efficiency and space overhead: For joins between high-frequency terms, efficiency is the major concern. For joins between high and medium-frequency terms, space cost should be considered, due to the large number of the medium-frequency words. These requirements match the provisions of joint indexes and contextual indexes, respectively, and thus our choice.

5.3 Query Processing

This section discusses how to select indexes for efficient query processing. Given a pattern condition, there are many related candidate inverted lists to choose from the basic inverted list layer and the advanced inverted index layer. Consider the following condition with two patterns:

pattern("[population #number]⟨4⟩")
OR *pattern*("{#number native people}")

Assume the window size W is 4. Keywords "native" and "people" are high-frequency words while "population" is a medium-frequency word in the data corpus. There are 8 indexes related to the query as displayed in Figure 7. Each index has different coverage. For instance, index C("population" $\rightarrow T$), where T denotes all the data types, can cover "population" itself and #number in its context, since #number lies within the window of "population" in the query. Meanwhile, index I(#number) can cover the #number data types for the two patterns at the same time, which means that I("#number") could be shared across different patterns in query processing. Consider the following two index selection strategies:

- **S1**: 1) I("population"); 2) I(#number); 3) J("native","people").
- **S2**: 1) C("population" $\rightarrow T$); 2) J("native",#number); 3) J("native","people").

Query

pattern("[population #number]<4>") **OR** *pattern*("{#number native people}")

Candidate Index

Index	Cost	Index	Cost
I("population")	87372	I("native")	167905
I("people")	239568	J("people", #num)	73243
J("native", "people")	32342	J("native", #num)	45719
I(#number)	2512913	C("population"→T)	60430·β

Query Coverage Graph

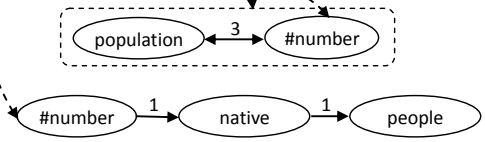


Figure 7: A query plan with index selection.

Either of the strategies is sufficient for the execution of the pattern condition. Take strategy **S2** as an example. The system first traverses C("population" $\rightarrow T$), selects the tuple list l_1 which contains #number occurrences with "population" around satisfying the first window pattern. It then traverses and joins J("native",#number) and J("native","people") to get list l_2 containing #number occurrences satisfying the second sequential contextual pattern. Finally, the system unions l_1 and l_2 to get the final result. For this simple query, the second strategy **S2** apparently works more efficiently than **S1**, because it avoids the transversal of the long list #number. To deal with complex patterns involving a large number of data types and keywords, DoCQS needs a smart algorithm to choose an efficient index selection strategy automatically.

We model the index selection problem as a graph coverage problem. The system first transforms a query into a *Query Coverage Graph* as Figure 7 shows. Each node in the graph indicates a keyword or data type appearing in the query, and the directed edge from node u to v indicates that v appears after u within the distance constraint specified by the weight of the edge. Each index covers part of a graph and has different traversal cost. We define the traversal cost of an index by the number of documents included in the list. For a contextual index, we penalize the cost by multiplying a coefficient β ($\beta > 1$), since a contextual index stores additional context word information. As a result, the index selection problem boils down to looking for a subset of indexes from the candidate index set to cover the whole query coverage graph with the minimal traversal cost.

Even in the simplest case, where all the indexes have the same traversal costs, this graph coverage problem is equivalent to the set cover problem, which is NP-Complete. This means an exhaustive algorithm with exponential complexity of O(2^K) or O(n^k) is needed for finding the optimal solution, where K is the number of candidate indexes, n the number of nodes, and k the average number of indexes that cover one node. Although many pruning conditions can be applied to shrink the search space, they can not guarantee stable performance for complex pattern conditions. For this reason, we design an O(K^2) polynomial greedy algorithm.

Algorithm 1, as shown above, first builds up the query coverage graph for the pattern conditions of a given query. Second, for each node in the graph, the algorithm collects all related indexes covering the node. Third, by iteratively checking the transversal cost of each index, it drops the index list with the highest traversal cost, such that each node can still be covered by at least one index. The iteration stops when no more indexes can be removed from the candidate set. In Section 6, we will show that the plan selected by our algorithm is very close to that of the optimal query plan in terms of actual query processing time.

Algorithm 1 Index Selection Algorithm

1: Convert the given query Q into the a directed graph $\mathcal{G} = \{\mathcal{U}, \mathcal{E}\}$. Each node u indicates a keyword/data type appearing in Q. Each edge $e = \langle u, v \rangle$ in $\mathcal{E}$ indicates that for u, v appears after u satisfying certain position constraints which is recorded as the weight.
2: Candidate Index Set R. R is initialized to include all indexes
3: **for** each $u_k \in \mathcal{U}$ **do**
4: Start from u_k, DFS the graph, get an index set C_{u_k} which stores all indexes related with u_k.
5: **end for**
6: **loop**
7: $T \leftarrow R$. T is a temporary set to store indexes that could be removed.
8: **for** each C_{u_k} **do**
9: **if** C_{u_k} contains only one index i' **then**
10: Remove i' from T
11: **end if**
12: **end for**
13: **if** T is empty **then**
14: Return R as result.
15: **end if**
16: Select $i_{max} \in T$ with maximal transversal cost.
17: Remove i_{max} from R.
18: Remove i_{max} from C_{u_k}, if $i_{max} \in C_{u_k}$.
19: **end loop**

6. EXPERIMENTAL RESULTS

This section testifies the expressiveness of CQL, and evaluates both the time and space cost of the DoCQS system. We build two datasets from two different domains, Wikipedia Text Domain and Academic Personal Homepage Domain. On each domain of data, we show how to use CQL to describe different query tasks, and demonstrate that the system could retrieve high-precision results with reasonable time and space cost.

WikiPedia Text Domain This dataset comes from the Wikipedia corpus downloaded in March 2009. We choose Wikipedia because it is an entity-rich dataset containing a lot of data type information. As the current DoCQS system focuses on unstructured text, we remove all infoboxes and tables on the pages. After data cleaning, the corpus size is 7Gb including 3 million pages. On the corpus, we target at three basic data types: number, person and location. We extracted 83 million number occurrences by a JFLEX parser, 23 million person occurrences and 28 million Location occurrences by the Stanford Named Entity Recognizer[2].

Academic Personal Homepage Domain This dataset is composed of pages about academic people in computer science. We retrieve the name list containing 724,817 authors from DBLP Bibliography website. Using each name as query, we use Google to retrieve the top 3 related pages, and collect a 9GB data corpus containing 2 million webpages. Three basic data types are extracted: 61 million person occurrences, 20 million organization occurrences and 1 million email occurrences.

[2]http://nlp.stanford.edu/software/CRF-NER.shtml

	Population	Capital	CEO
Precision	65.5%	90.0%	67.0%

	Professor-Email	Professor-University
Precision	86.0%	96.0%

Figure 8: Precision measurement.

All experiments are carried on a PC with 2.4GHz Intel Core 2Duo CPU, 1T disk and 3Gb of RAM. We leverage and extend Lucene index to support our designed index structures.

6.1 Case Study

This section studies the expressiveness of the CQL language. We study two example tasks (TES and WIE mentioned in Section 1) over the aforementioned two domains, and show that these content query tasks could be well supported by CQL in our framework.

6.1.1 Wikipedia Text Domain

On the Wikipedia text domain dataset, we conduct experiments for the Typed-Entity Search (TES) application, to support searching specific types of information based on user inputs. More specifically, we design three tasks, described as follows:

1. Based on #number, retrieve population of a given country.
2. Based on #location, retrieve capital of a given country.
3. Based on #person, retrieve CEO of a given company.

Due to space limitation, we only show the CEO search example in query **Q5**:

SELECT #person.val
FROM #person
WHERE (*pattern*("[{CEO of IBM} #person]⟨6⟩")[0.9]
OR *pattern*("[{CEO ?⟨0, 3⟩ #person} IBM]⟨20⟩")[0.5])
AND ∼*prefer*(*TF*("IBM")>10, 0.6, 0.2)
GROUP BY #person
ORDER BY *conf*()

In query **Q5**, the "IBM" keyword is the user input. It can be substituted by any other company names (*e.g.*, Microsoft, Bank of America, *etc.*). Functions such as ∼*prefer*,*TF* are supported in our system. *TF*("IBM") will return the number of "IBM" occurrences in a document . ∼*prefer*(*TF*("IBM")>10, 0.6, 0.2) returns score 0.6 when the value of *TF*("IBM") is greater than 10, or 0.2 otherwise. The query contains two patterns, and the former is more restrictive than the latter. We favor the first pattern by assigning it with a higher weight of 0.9. **Q5** uses the implicit scoring function.

To measure the precision of the three tasks, we use 200 country names as input for population and capital search, and 100 IT companies [3] for CEO search. For each query, we manually check whether top 3 results include the correct answer. The precision results are listed in Figure 8. We find that Capital search retrieves the highest precision, while CEO search and Population search are comparatively lower in precision. The low precision is mainly due to the lack of redundancy in Wikipedia corpus where a lot of data type instances usually only appear once. Users can further define more restrictive rules to retrieve more precise results. The result shows that the system can support various TES tasks by returning satisfactory results.

6.1.2 Academic Personal Homepage Domain

On the second dataset of the academic personal homepage domain, we conduct experiments for the Web-Information Extraction (WIE) application, which aims at collecting facts from the Web. This dataset contains a lot of author homepages, upon which we study two tasks:

- Professor-Email extraction. Extract professor-email pairs from the dataset. #professor is defined based on #person.
- Professor-University extraction. #university is defined based on #organization.

Take #professor as example, we define #professor by query **Q6**:

DEFINE DATATYPE #professor.val **AS** #person
WHERE *pattern*("[(prof|professor) #person]⟨4⟩")[0.8]

Query **Q6** defines #professor to be #person with keywords "professor" or "prof" around. With #professor defined, the professor-email pairs could be extracted by query **Q7**:

SELECT #professor.val, #email.val
FROM #professor, #email
WHERE *pattern*("{#professor ?⟨0,20⟩ #email}")
GROUP BY #professor, #email
ORDER BY *conf*()

Query **Q7** collects professor-email pairs in which #professor and #email appear within window of 20 words. The collected results are ordered by the overall pair frequency (implicitly involved in the scoring calculation of the GROUP BY clause) in the data corpus. This statement extracts 12,174 professor-email pairs and 34,982 professor-university pairs from the data corpus, and we randomly sample 100 pairs from each of them for precision measurement, shown in Figure 8. The high precision of results validates the applicability of utilizing CQL language for content query tasks.

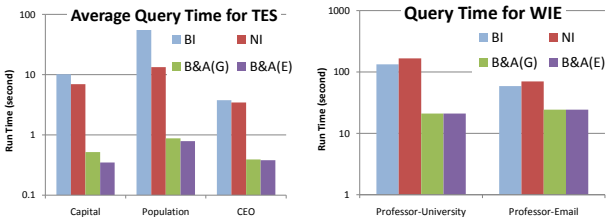
In summary, all the queries studied in the above two domains involve all the essential CQL characteristics. We show that various content query tasks could be well supported by CQL. We also find that a content query task usually involves multiple patterns and complex scoring functions which need careful tuning for good performance. DoCQS simplifies this process in allowing administrators to avoid modifying underlying system programs by quickly trying out different queries using CQL.

6.2 Time Cost Analysis

For a data-oriented content query system to support arbitrary user-defined queries, performance is a major concern. This section compares the query performance of the tasks defined above using 4 different index selection strategies: 1) BI: using the basic inverted list only; 2) NI: using the neighborhood index (the core index structure of the BE engine described in [3], whose main idea is to store the immediate neighbors of each word in the metadata to speed up phrase queries); 3) B&A(G): using indexes selected by our greedy query plan generation algorithm over basic inverted list and advanced inverted index; 4) B&A(E): using the optimal index plan based on exhaustive plan generation over basic inverted list and advanced inverted index. Using the former two as baseline, we quantitatively demonstrate that the advanced index layer helps greatly improve the query performance. By comparing the latter two cases, we show that our index selection algorithm achieves close to optimal query processing time.

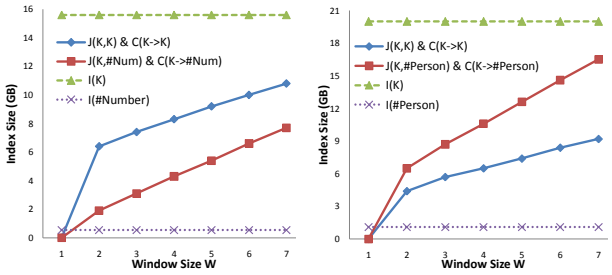
On Wikipedia Text Domain, the average query time on three index structures are compared in Figure 9(a). It shows that the advanced index layer achieves six to ten times improvement compared with merely using the basic inverted index, and two to seven times faster than the neighborhood index. That's because the neighborhood index only store immediate neighbors, providing limited optimization room for complex CQL queries. In some cases where

[3] http://www.netvalley.com/top100am_vendors.html



(a) Wikipedia domain. (b) Academic domain.

Figure 9: Query time comparison.



(a) Wikipedia domain. (b) Academic domain.

Figure 10: Space cost with different window size W.

few words appear consecutively, the neighborhood index degenerates to the standard inverted index. With the help of the advanced index layer, the average query time of our system is less than one second per query. Results also validate that our greedy algorithm can derive approximately optimal query plans.

On Academic Personal Homepage Domain dataset, we directly measure the execution time for two tasks, shown in Figure 9(b). It still shows that our indexing framework works much more efficiently than the other two baseline indices. As there are no consecutive patterns in the query, the performance on the neighborhood index is even worse than on the basic inverted list layer. For these two queries our greedy algorithm can derive the exactly same optimal query plan with the exhaustive algorithm, thus the query time for the latter two cases are the same. Compared with query on the first domain for TES application, it takes longer query time for WIE application because there is no specific words(like country name or company name which appears much less in the data corpus) involved in WIE query. However, since WIE applications are usually carried on offline, the time cost is still acceptable.

6.3 Space Cost Analysis

The size of the advanced inverted index is variable by the window size W for the joint index and the contextual index. In Figure 10, we show the space cost of the indexes with different window sizes, and also compare them with the space cost of the basic inverted list for keywords and data types. It can be found that, as the window size increases, the space cost increases smoothly, allowing larger window size to support more flexible patterns. In actual implementation, we choose 5 as the default window size. The detailed space cost is summarized in Figure 11. As we can see, the space cost is acceptable compared with the original corpus size.

Domain	Basic Inverted List		Advanced Inverted Index	
	$I(K)$	$I(T)$	$J(K,K)\&$ $C(K \to K)$	$J(K,T)\&$ $C(K \to T)$
Wikipedia	15.6GB	1.25GB	9.2GB	12.1GB
Academic	20GB	2GB	7.4GB	19.7GB

Figure 11: Space cost in actual implementation.

7. CONCLUSIONS

This paper proposes a general data-oriented content query system (DoCQS), with a flexible content query language (CQL) to support search data in text. For efficient support of the query language, we study index design and query processing, by introducing novel indexes and an effective index selection algorithm. Experiments show the effectiveness and expressiveness of the query language, as well as efficient query processing with reasonable space overhead. We plan to provide online demo of the DoCQS system at `http://wisdm.cs.uiuc.edu/demos/docqs`.

8. REFERENCES

[1] S. Agrawal, K. Chakrabarti, S. Chaudhuri, and V. Ganti. Scalable ad-hoc entity extraction from text collections. *PVLDB*, 1(1), 2008.

[2] E. Brill, S. Dumais, and M. Banko. An analysis of the askmsr question-answering system. In *EMNLP*, 2002.

[3] M. Cafarella and O. Etzioni. A search engine for natural language applications. In *Proceedings of the 14th international conference on World Wide Web*, pages 442–452. ACM New York, NY, USA, 2005.

[4] M. J. Cafarella. Extracting and querying a comprehensive web database. In *CIDR*, 2009.

[5] M. J. Cafarella, C. Re, D. Suciu, and O. Etzioni. Structured querying of web text data: A technical challenge. In *CIDR*, 2007.

[6] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW*, pages 717–726, 2006.

[7] T. Cheng, X. Yan, and K. Chang. EntityRank: searching entities directly and holistically. In *Proceedings of the 33rd international conference on Very large data bases*, pages 387–398. VLDB Endowment, 2007.

[8] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in knowitall. In *WWW*, 2004.

[9] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, 2008.

[10] J. J. Lin and B. Katz. Question answering from the web using knowledge annotation and knowledge mining techniques. In *CIKM*, 2003.

[11] P. Marius, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Organizing and searching theworldwideweb of facts - step one: the one-million fact extraction challenge. In *AAAI*, 2006.

[12] G. Ramakrishnan, S. Balakrishnan, and S. Joshi. Entity annotation using inverse index operations. In *EMNLP*, 2006.

[13] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, 2008.

[14] M. Wu and A. Marian. Corroborating answers from multiple web sources. In *WebDB*, 2007.