Information **2015**, *6*, 275-286; doi:10.3390/info6020275

OPEN ACCESS

information

ISSN 2078-2489

www.mdpi.com/journal/information

Article

ODQ: A Fluid Office Document Query Language

Xuhong Liu [1,2,*], Ning Li [1,2], Yunmei Shi [1,2] and Xia Hou [1,2]

[1] School of Computer, Beijing Information Science & Technology University, Beijing 100101, China; E-Mails: ningli.ok@163.com (N.L.); sym@bistu.edu.cn (Y.S.); houxia@bistu.edu.cn (X.H.)

[2] Beijing Key Laboratory of Internet Culture and Digital Dissemination Research, Beijing 100101, China

* Author to whom correspondence should be addressed; E-Mail: liuxh0315@126.com; Tel.: +86-10-6487-9089.

Academic Editor: Willy Susilo

Received: 30 December 2014 / Accepted: 2 June 2015 / Published: 11 June 2015

Abstract: Fluid office documents, as semi-structured data often represented by Extensible Markup Language (XML) are important parts of Big Data. These office documents have different formats, and their matching Application Programming Interfaces (APIs) depend on developing platform and versions, which causes difficulty in custom development and information retrieval from them. To solve this problem, we have been developing an office document query (ODQ) language which provides a uniform method to retrieve content from documents with different formats and versions. ODQ builds common document model ontology to conceal the format details of documents and provides a uniform operation interface to handle office documents with different formats. The results show that ODQ has advantages in format independence, and can facilitate users in developing documents processing systems with good interoperability.

Keywords: fluid office document; query language; common document model; information retrieval

1. Introduction and Motivation

Big Data is recognized as the hot topic in the cloud-computing area, and has received increasing attention by researchers. Big data is high volume, high velocity, and high variety information that requires new forms of processing to enable enhanced decision making, insight discovery and process optimization. Big data usually includes structured data, semi structured data and unstructured data.

Office documents include fixed office documents and fluid office documents, they account for a considerable proportion of semi structured data. Lots of logic information is implicated in them and their value needs to be further developed. Due to the wide range of document formats, extracting valuable information from these documents by the same method is now an issue that needs to be addressed as a matter of urgency.

UOML is a universally representative operating language for the abstract description of fixed documents, with features including document organization, page description, index and search, content extraction, font management, *etc.* [1]. UOML was approved as an OASIS Standard in 2008 [2]. UOML is vendor-neutral, document-format-neutral, application-neutral, platform-neutral and programming language neutral. However, UOML can only extract content from a fixed office document.

As for fluid office document area, three major open document formats are often used at present, namely: Office Open XML (OOXML), Open Document Format (ODF) and Unified Office Document Format (UOF) [3–5]. Since they are all XML-based formats for office documents, you can extract information from them by using a query language for XML such as XQuery [6]. An integration technique of multiple document formats by using XQuery was put forward in the literature [7]. However, the query language for XML has currently only been designed for querying XML documents, they cannot provide the ability to query the following function points such as meta data, paragraph, section, table and so on. Moreover, the XPath used by XQuery to check the lexical properties of the source data is long and complex, so is hard for general users to understand and use.

For this reason, some organizations for standardization and office software manufacturers provide APIs for accessing the documents with corresponding format, such as OOXML API, UOF API and ODF API. The document manipulations are wrapped in these APIs, which makes it much simpler to access documents than XQuery does. However, API contains some intrinsic defects, mainly reflected in the following aspects [8]: (1) The APIs are designed to be dependent on document formats and development environment. For instance, ODF API depends on UNO component technology, while OOXML API needs VBA/.NET and windows environments; (2) differences exist between the different versions APIs of the same document format standard; (3) the APIs are complex and difficult to use.

Additionally, web information retrieval technology can also be used to access the fluid office documents, it focuses on text retrieval while the most important structured information is often lost during preprocessing [9]. For example, it is hard to extract content of a given paragraph from an article by web information retrieval technology.

Above all, the fluid document area lacks a uniform, simple and platform-independent technology to retrieve content from documents with different formats.

To solve this problem, we present in this paper a technique, that we call Office Document Query (ODQ) language, to build a uniform interface to query fluid office documents with different formats for user. We build an ontology-based common document model to hide differences between office document standards. ODQ encapsulates the document APIs and conceals their operation details. The simple, uniform and platform-independent interface enables interoperability between different document format standards.

2. Design of ODQ

2.1. Design Principles

To solve the problem mentioned in the previous section, six requirements of ODQ are elicited, as shown below.

(1) ODQ should be a non-procedural query language without branch and loop structure for fluid office documents, so can easily be embedded in any high-level language.
(2) ODQ can be used to access fluid office documents directly without any office software, so can easily be integrated in fluid office document application development.
(3) A common document model is required to conceal the format details between different document format standards and improve document interoperability.
(4) Common functions should be provided to meet the various needs of users. For instance, query, update and delete operations for metadata, paragraph, section, table, and so on.
(5) Independent from platforms, document formats and versions, programming languages and applications.
(6) Syntax should be as simple as possible to reduce the difficulty of learning for developers.

According to the principles above, the functions of ODQ are similar to those of Structured Query Language (SQL). As illustrated in Figure 1, database applications in the application-tier, such as finance application, ERP system and so on, access data in different database management system by the uniform SQL command. Similarly, various office applications in the application-tier, such as web office and information retrieval, can use the uniform ODQ command to extract, modify, or store information in fluid office documents regardless of their formats. ODQ provides a uniform interface to conceal the details of APIs and the difference between document formats; this makes the application-tier completely separated from the physical-tier. In this way, ODQ frees up application developers by letting them concentrate on building new applications without concern regarding the underlying document formats. Therefore, this process is similar to a database management system (DBMS), where ODQ acts in a role similar to SQL.

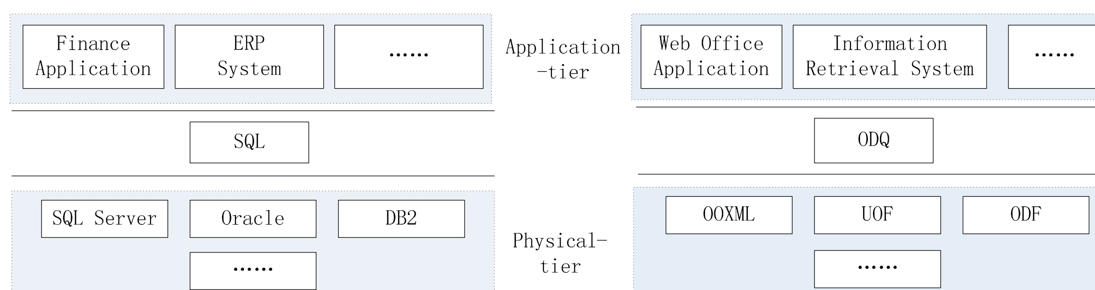

Figure 1. Structured query language (SQL) *vs*. office document query (ODQ).

2.2. Ontology-Based Common Document Model

Just like SQL is a query language designed specifically for relational data model and XQuery is for hierarchical data model, ODQ needs a common document model appropriate for querying fluid office document. The design of a common document model follows these two principles.

(1) As each fluid office document format has its own document model, the common document model should not only extract common function points from them, but also hide differences between them.
(2) The common document model should be as flattened as possible, which ensures the path expression in ODQ statement simple enough. A method proposed in the literature [7] can make the common document model flatten enough.

This paper adopts ontology to construct common document model. Ontology formally represents knowledge as a hierarchy of concepts within a domain, uses a shared vocabulary to denote the types, properties and interrelationships of those concepts [10].The ontology is a structural framework for organizing information and can be used to extract common function points from various fluid office document formats.

The common document model established in this research is 5-tuple $\mathrm{O} = (\mathrm{C}, \mathrm{A^C}, \mathrm{R}, \mathrm{O}, \mathrm{A^O})$, where:

(1) The set of function points C which includes most commonly used function points. For example, the *document* function point represents the whole document and the *paragraph* function point represents a certain paragraph in a document.
(2) The property set of function points $\mathrm{A^C}$ which includes all properties of function points defined in C. Each function point has multiple properties. Table 1 lists part properties of the *paragraph* function point.
(3) The set of relationships R which contains not only relationships between function points, but relationships between a function point and its properties.
    - *Part-of* relationship which describes a function point that is a part of another one. For instance, $part - of(paragraph, document)$ represents *paragraph* is a part of *document*.
    - *Property-of* relationship which describes the property that a function point has, for example, $property - of(fontName, paragraph)$ shows that *paragraph* function point has a *fontName* property.
(4) The resources O which includes all function points in various fluid document models.
(5) $\mathrm{A^O} \subset \mathrm{C} \times \mathrm{O}$ maps function points in common document model to those in a particular document model. For example, $map - to(paragraph, ooxml\!:\!p)$ represents the mapping of *paragraph* in common document model to the element *p* in OOXML document model.

The whole common document model extraction framework is divided into conceptual-tier and resource-tier. As illustrated by Figure 2, the common document model in the conceptual-tier is created by extracting the most commonly used function points from various fluid document models in the resource-tier. The *part-to* relationships between function points in the common document model organize them into a tree structure. For example, the *meta*, *section*, *paragraph*, *etc*., are children of *document*. The *property-of* relationship specifies the properties of a function point, for instance, the *fontName*, *fontColor*, *etc*., are properties of *paragraph*. The function points and properties in common document model are mapped to the function points in different fluid document models in the resource-tier by *map-to* relationship. For example, the *paragraph* in common document model can be mapped to the element *p* in OOXML document model and *text*:*p* in ODF document model.

Table 1. Partial properties of *paragraph*.

Properties	Description
text	Textual Content of a Paragraph
indentLeft	Left Indent Value of a Paragraph
indentRight	Right Indent Value of a Paragraph
lineSpaceType	Line Space Type of a Paragraph
lineSpaceValue	Line Space Value of a Paragraph
fontName	Font Name of a Paragraph
fontSize	Font Size of a Paragraph
fontColor	Font Color of a Paragraph

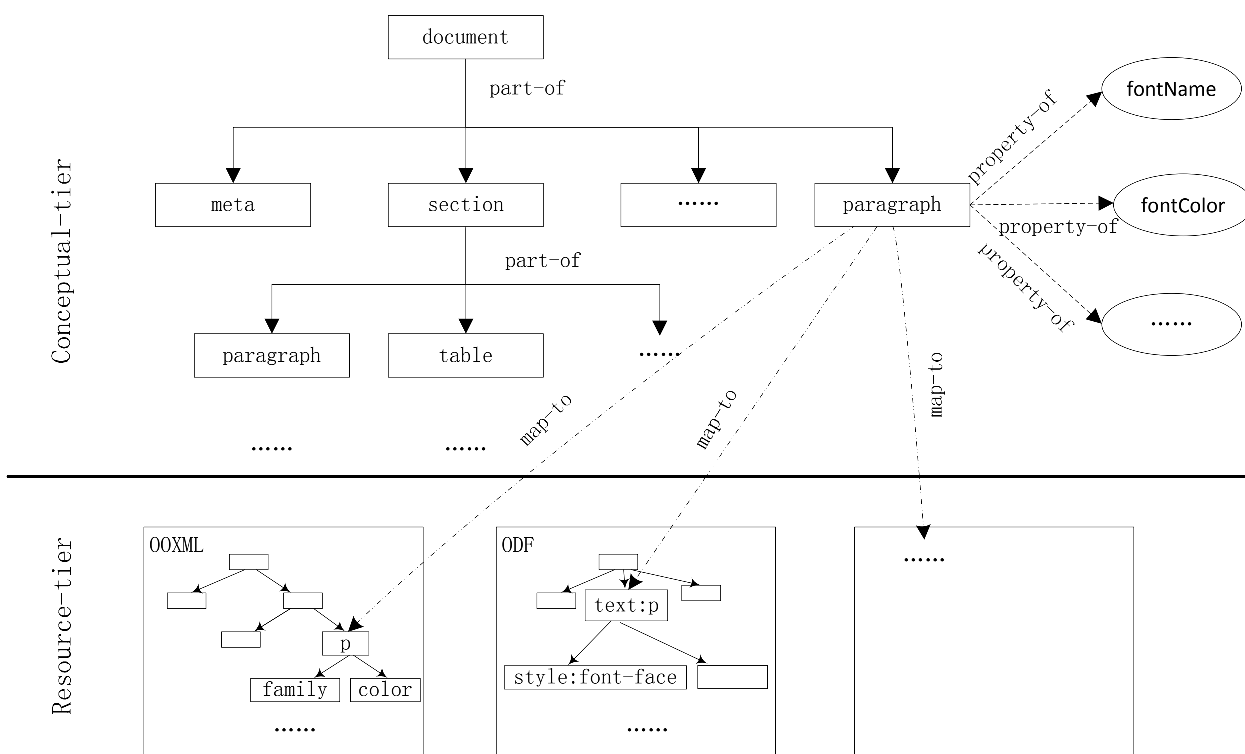

Figure 2. Common document model extraction framework.

Each function point in common document model has multiple properties. Table 1 lists part of properties of the *paragraph* function point. The *text* property represents the textual content of a *paragraph* without any style information. However, there is no *text* property in any document model; it is provided to facilitate query content from a certain paragraph.

The ODQ commands are designed to operate the common document model, but the results come from the underlying fluid office document.

2.3. ODQ Syntax Design

Considering that most developers are familiar with SQL syntax, ODQ refers to SQL. The simple syntax makes ODQ easy to learn. We use EBNF grammar to describe ODQ language. As an illustration, the *SELECT* command syntax for querying *document*, *paragraph*, *meta*, is list as follows.

(1) SELECT_STATEMENT :: = “SELECT”<AttributeList> | <NodeList>“FROM”<URL> [“WHERE”<ConditionList>]

(2) <AttributeList> :: = <Attribute> | <AttributeList>, <Attribute>

(3) <URL> :: = <NodeList> [“of”<NodeList>]*

(4) <ConditionList> :: = <Condition> [“AND” |“OR”<Condition>]*

(5) <Attribute> :: = <DocumentAttribute> | <MetaAttribute> | <SectionAttribute> | <PargraphAttribute>

(6) <DocumentAttribute> :: = “text” |“fontName” | “fontSize” | “fontColor”

(7) <MetaAttribute> :: = “Author” | “Title” | “Creator” | “CreationDate”

(8) <SectionAttribute> :: = “text” |“fontName”| “fontSize” | “fontColor”

(9) <PargraphAttribute> :: = “text” | “indentLeft” | “indentRight” | “fontName”| “fontSize” | “fontColor”

(10) <NodeList> :: =<Node> [<NumberRange>] | <NodeList>, <Node> [<Range>]

(11) <Node>:: = “document” | “section” | “paragraph” | “table” | “run” | “meta”

(12) <NumberRange> :: = “all” | <Range> | <NumberList>

(13) <Range> :: = <NumberList> “-”<NumberList>

(14) <NumberList> :: = <Number> | <NumberList><Number>

(15) <Number> :: = “1” | “2” | “3” | “4”| “5” | “6” | “7” | “8” | “9” | “0”

(16) <Condition> :: = <Attribute>“=”<AttributeValue>

(17) <AttributeValue>::=<TextValue>|<FontNameValue>|<FontSizeValue>| <FontColorValue> | <IndentLeftValue>

(18) <TextValue> :: = <String>

(19) <FontNameValue> :: = “宋体”| “黑体” | “Times New Roman”

(20) <FontSizeValue> :: = <NumberList>

SELECT command includes three main clauses.

(1) *SELECT* clause lists the contents that should be returned by the query. All function points here are from common document model, but the results are fetched from the underlying fluid office document by *map-to* relationship.

(2) *FROM* clause includes a path expression indicating the document from which content should be obtained. Path expression contains series function points with the *OF* keyword between them.

(3) *WHERE* clause is optional and indicates the conditions under which information will be included in the result.

Moreover, the query result to user is surrounded by XML tags which can easily be integrated in HTTP protocol. The names of the tags are the same as the properties that user queries. For example, a result surrounded by *<fontName>* tag will be returned if the user queries the font of some paragraph.

The following examples illustrate how to use *SELECT* command to query function points and their properties.

- Example 1: Query *text* property. Query *text* property of *document* function point to fetch the textual content of sample.uot.

SELECT text FROM sample.uot;

- Example 2: Query *style* property. SELECT *fontName* and *fontSize* property of the second paragraph in the second section of sample.docx.

SELECT fontName, fontSize FROM paragraph[2] of section[2] of sample.docx;

- Example 3: Query function point. Get the second paragraph in the second section of document sample.odf, and the results will include all properties of the second paragraph.

  SELECT paragraph[2] FROM section[2] of sample.odf;

- Example 4: Conditional query. Get all paragraphs whose font name is "Times New Roman".

  SELECT paragraph FROM sample.docx Where fontName = "Times New Roman";

3. Query Parsing and Result Generation

ODQ is a non-procedural query language without branch and loop structure, so can easily be embedded in any high-level language. The office applications can easily use ODQ to access documents of different formats. Figure 3 illustrates the application framework integrated with ODQ. ODQ query engine parses the ODQ command submitted by the application and generates execution plan according to the document format. The runtime system implements execution plan and access the underlying document by calling the corresponding document standard APIs. The working process is as follows.

The application submits an ODQ command to ODQ parser. The lexical analyzer converts ODQ command into a sequence of tokens and gives prompt if there is any syntactic error.

The parser reads the tokens and performs semantic checking, if there are no mistakes, a query tree will be generated.

The code generator explores query tree, and then output an execution plan. The adapter in the code generator converts the query according to the common document model and then further maps into the actual document model, e.g., UOF. The execution plan includes a series of APIs that associated with the particular fluid document format and versions. The series API functions are called by the runtime system to access and manipulate the data inside a fluid office document. Finally, the results will be returned to the application.

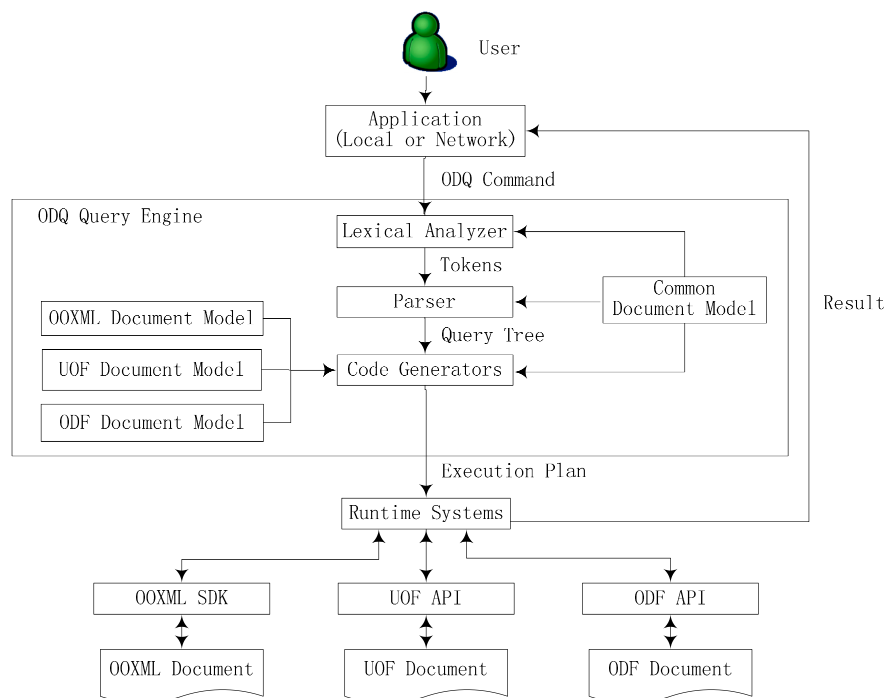

Figure3. Application framework integrated with ODQ.

For example, if a user wants to fetch text content of the first paragraph in the first section of a UOF document *test.uot*, the user needs to submit "*SELECT text FROM paragraph[1] OF section[1] OF test.uot*" to ODQ query engine. Part of the execution plan is as follows:

```
IDocument * doc = new IDocument();
ISectionSet* sections = doc->getSectionSet();
ISection* section = sections->getItemByID(1)
…….
```

The interface functions involved in above execution plan are defined in UOF API. The execution plan will call the interface functions in the OOXML API or ODF API if the user accesses an OOXML document or an ODF document.

Runtime system performs the execution plan and accesses the actual document, then returns the results to the user.

For those query accessing different parts and different document formats, code generator generates different execution plan. In this way, ODQ hides operation details of APIs and format differences, thereby providing a uniform query interface for user.

4. Experiments and Evaluation

The ODQ parser in our prototype is built on ANTLR. Figures 4 and 5 show how to fetch text content of the first two paragraphs respectively from an OOXML document and an UOF document by the similar ODQ command.

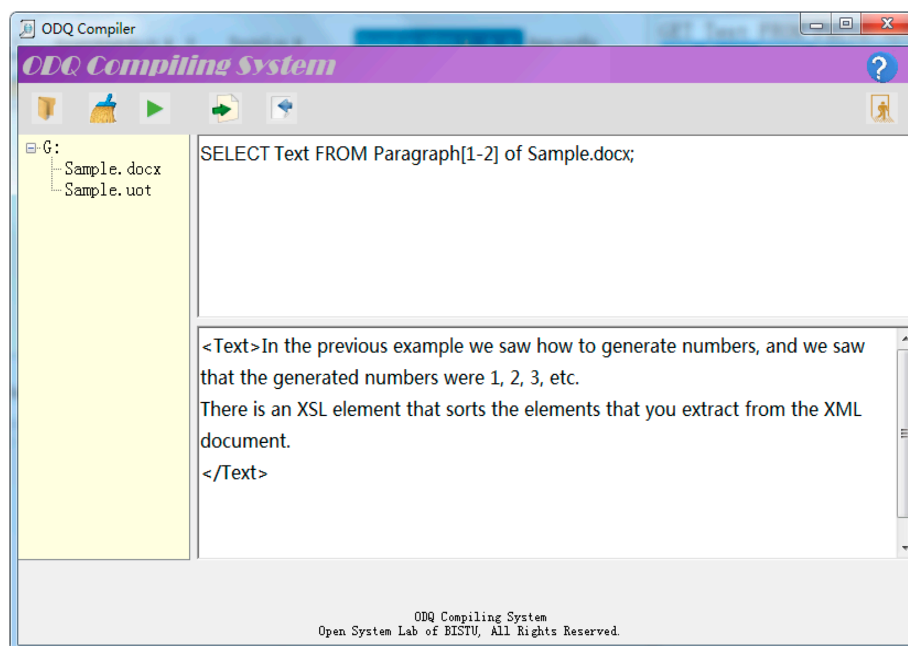

Figure 4. Fetch text content from the first two paragraphs from an OOXML document.

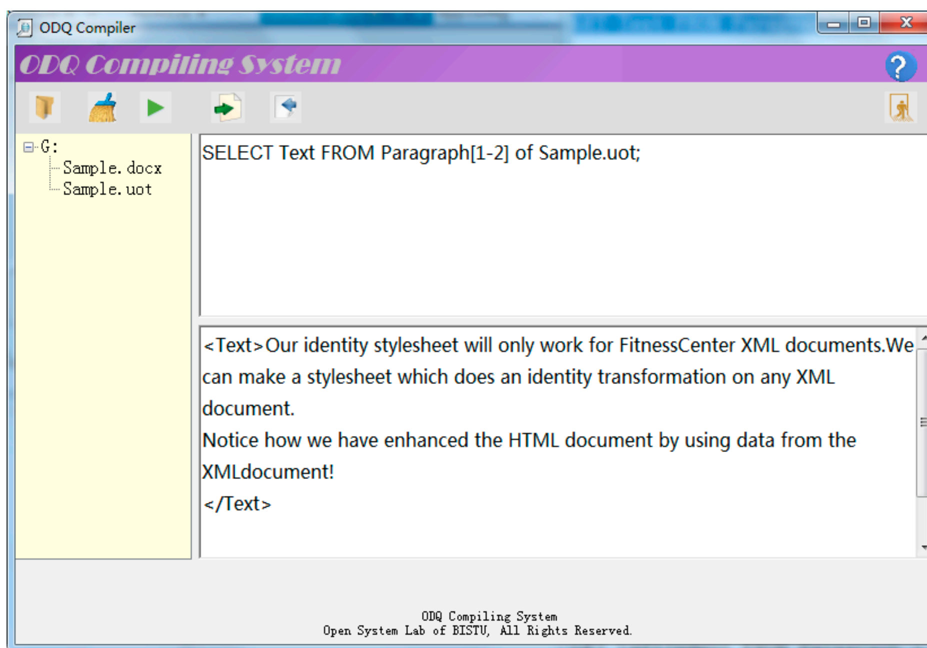

Figure 5. Fetch text content from the first two paragraphs from an UOF document.

The above experiments show that ODQ provides a uniform interface for user to access documents of different formats.

Furthermore, ODQ has a simple syntax similar to SQL. For the same function, ODQ provides more compact code than those using APIs. We illustrate three kinds of code to fetch text from the first two paragraphs of a document, *i.e.*, by ODQ command, by OOXML APIs, and by UOF APIs.

Table 2. Comparison of code to fetch text of the first two paragraphs by three ways.

ODQ Command	SELECT text FROM paragraph[1–2] of filename (note: can access document with any format)

OOXML API
(note: can only access OOXML document)
	for (inti = 1; i<= 2; i++)
	{
	Paragraph p = doc.Range().Paragraphs[i];
	text += p.Range.Text;
	}

UOF API
(note: can only access UOF document)
	for(inti = 0; i< 2; i++)
	{
	IParagraph p = (IParagraph) textDoc.getParagraphs().getItemByIndex(i);
	ITextRunSet runs = p.getTextRuns();
	for(int j = 0; j <runs.getCount(); j++)
	{
	ITextRun r = (ITextRun) runs.getItemByIndex(j);
	text += r.getTextContent();
	}
	}

Table 2 shows that it is much easier to access document by ODQ command than by calling document APIs. We select 13 frequently used queries against *meta*, *paragraph*, *section* and *table*, then implement

them respectively by ODQ commands, by OOXML APIs, and by UOF APIs. The amounts of code of three ways are list in Table 3.

Table 3. Amounts of code of three methods.

Testing Functions	OOXML API	UOF API	ODQ
Get author of a given document	1	1	1
Get title of a given document	1	1	1
Get creator of a given title of a document	4	4	1
Get creation time of a given title of a document	4	4	1
Get text content of a given document	4	10	1
Get text content of a given section	4	10	1
Get text content of a given paragraph	1	7	1
Get all paragraphs whose font name are “Times New Roman”	4	14	1
Get text content whose font name is “Times New Roman” from the first section.	4	15	1
Get font name of a given paragraph	4	4	1
Get a given paragraph	1	1	1
Get a given table	1	1	1
Get a cell from a given table	2	2	1

Moreover, web information retrieval is a commonly used technology to extract information from fluid office documents. The format information is lost during the preprocessing, so queries related to format cannot be implemented. For example, it cannot perform the last six functions in Table 3.

Using query language for XML such as XQuery can also retrieve content from fluid office documents. However, the XQuery grammar is more complex than ODQ as illustrated in Table 4.

Table 4. Comparison of code to fetch text of the first two paragraphs by ODQ and XQuery.

ODQ Command	SELECT text FROM paragraph[1–2] of filename
XQuery	declare namespace w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"; for $x in doc("document.xml")/w:document/w:body/w:p[2]/w:r for $y in $x/w:t return xs:string($y)

5. Conclusions

This paper presented a query language ODQ for accessing fluid office document. Based on the framework proposed, a prototype system is designed and implemented. Experiment results show that ODQ has following four major advantages over others.

- It hides differences between fluid office document formats and facilitates interoperability among all kind of office documents.
- It offers a united interface for user to handle different format documents.

- Thanks for the features of ODQ, e.g., non-procedural, platform- and language-independent, it is easy to embed into document-based applications to access the fluid office documents either remotely or locally.
- It has simple syntax, thus is very easy to use.

As for further studies, it is necessary to improve query efficiency and accomplish more functions. These studies are in progress and are expected to present results in the future.

Acknowledgments

The authors are grateful to Scientific Research Fund Project of Beijing Education Commission (KM201511232012) for funding this research. This research is also supported by the Opening Project of Beijing Key Laboratory of Internet Culture and Digital Dissemination Research (ICDD201409).

Author Contributions

Ning Li designed research; Xuhong Liu, Yunmei Shi and Xia Hou performed research; Xuhong Liu wrote the paper. All authors have read and approved the final manuscript.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Wang, D.L.; Jiang, H.F.; Zhang, C.Y. UOML: An unstructured operation markup language. *Inf. Technol. Inf.* **2007**, *3*, 121–125.
2. OASIS. Information Technology—UOML (Unstructured Operation Markup Language) Part 1 Version 1.0. Available online: http://docs.oasis-open.org/uoml-x/v1.0/os/uoml-part1-v1.0-os.html (accessed on 3 October2013).
3. ISO/IEC. Information Technology—Open Document Format for Office Applications (OpenDocument) v1.0. Available online: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43485 (accessed on 12 August2014).
4. ISO/IEC. Information Technology—Office Open XML file formats. Available online: http://www.iso.org/iso/news.htm?refid=Ref1181 (accessed on 12 August2014).
5. Specification for the Chinese office file Format (GB/T). Available online: http://doc.csres.com/showdoc-2541-44390.html (accessed on 2October2014).
6. Tang, Y.; Tian, Y.A.; Li, N. Analysis of methods to access XML-based fluid office documents. *Comput. Eng. Design* **2014**, *4*, 1458–1464.
7. Sun, Q.G.; Zhu, W.; Liu, H.J.; Zhang, P. Data integration of open document format on XQuery. *Comput. Syst. Appl.* **2008**, *7*, 32–34.
8. Ling, F.; Liu, X.H.; Tian, Y.A. Flatten design of open document query. In Proceedings of the International Conference on Cyberspace Technology (CCT2013), Beijing, China, 23 November 2013.

9. Li, N.; Liang, Q.; Shi, Y.M. The function of format information in document understanding. *J. Beijing Inf. Sci. Technol. Univ.* **2012**, *6*, 1–7.
10. Wang, H.; Gu, J.; Su, X.N. Research on the Model and Its Application of Ontology-driven Knowledge Management System. *J. Libr. Sci. China* **2013**, *3*, 98–110.

© 2015 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).