XPath Compiler for MongoDB

Ding Fan Guo Dan Lin Xin Li Xinran Zhang Anxing

April 2022

Abstract

Almost all computer applications we use today, from desktop to web and mobile devices, rely on one of the two major data interchange formats: JSON and XML. However, there are gaps between manipulating recent JSON data by XML query languages, in particular, XPath. To assist XPath and XML users, some libraries were designed to enable querying JSON files with XPath-like syntax. We examined and compared three popular libraries, none of them support pure XPath syntax. In this paper, we introduced a new lightweight library allows XPath query execution directly on MongoDB with JSON files in it to address the pain points.

Our library https://github.com/CS5421-2022-Spring/Xpath-for-json-and-MongoDB provides an API that specifies a database and collection, then takes a XPath query string argument and boolean value to indicate whether the input is in full syntax. It returns the query result in XML format to the users for further process. Experiments are done to compare the features between our project and related libraries. The results have proved that our project covers essential features among the existing libraries, and we provide additional support for direct database connection, complicated predicates and flexible output format.

1 Introduction

Motivation. XML has been around for over three decades and is an essential part of web applications. XML facilitates data exchange by providing a clear and unified data structure; and allowing variables to be configured dynamically and loaded. Many companies have built their database using XML to store information such as web content. XML databases have been well-established and maintained for a long time for them.

However, JSON is another format introduced in the early 2000s[4]. It stores data in a concise and easy-to-understand map format (key-value pairs). JSON is said to be slowly replacing XML because of its advantages of simplicity of data modeling, accurate mapping of domain objects, improved predictability, and simpler architecture. MongoDB is one of the most popular databases based on document structure and uses JSON-like representations for data records[3].

Therefore in the hybrid world, there are situations where developers only familiar with XML and XPath have to process JSON data or operate in MongoDB. It would be desired by them to be able to use XPath syntax to query and even manipulate those datasets without the learning efforts. Understanding the needs and with the hope to ease the transition from XML to JSON, we have designed and implemented such a library that can take an XPath query and JSON dataset as input, and evaluate the query against the JSON data in MongoDB and translate the XPath queries to MongoDB versions to achieve the goal. The output format is a prettified XML string.

Problem Statement. To build a compiler for XPath and query in MongoDB JSON files, we construct several problems and solve them step by step: (1) How to read and properly parse the XPath query to the methods that can be accepted by MongoDB? We should be aware of the following aspects: axes, predicates (even nested predicates) and popular built-in function calls such as `count()` and `text()`. (2) What are the XPath features that we should support? How to deal with the different commands between XPath and MongoDB queries? For example, MongoDB cannot directly find the descendant axes like in XPath. (3) What is the output format of our program? We need to consider the use scenarios and then find a solution that is user-friendly and is able to provide a format that the users can handle easily.

Outline of Solution. (1) Reading an XPath query, we process the XPath as a string by identifying and removing the XPath built-in function, then we parse the remaining query as a tree structure, recursively find the axes and predicates of each node. A shorthand XPath query will be transformed into a full syntax to be consistent with the others. (2) To achieve the searching target in XPath, we translate it into "filter" and "projection" of MongoDB `find()` function. We also translate the XPath "descendant" that not supported by MongoDB into additional queries. (3) Since our target users are familiar with XML and XPath, we format the MongoDB original output into an XML output.

Summary of Contribution. In this paper, a python library is built to compile XPath both in shorthand and full syntax queries into a MongoDB version and output a pretty XML result. A convenient and useful API is exposed for users to search their JSON files in MongoDB with XPath syntax.

Plan of Paper. In the following part of this paper, we will introduce the background and related works in sections 2 and 3 respectively. The design and implementation details will be discussed in section 4. In section 5, the performance of our work and the related work will be compared. Section 6 will conclude our work and discuss possible future work.

2 Background

2.1 Extensible Markup Language (XML)

XML is one of the most widely-used formats for sharing structured data with highly customizable tags for users to fit their needs. It can describe very complex data structures and is often used to transmit and store data over the network. XML has several characteristics. **Human-readability**: uses UTF-8 encoding to store all data in plaintext by default. **Nestable**: suitable for representing structured data. Users also benefit from the standardized structure of XML, all XML files can be parsed the same way regardless of the content[11].

2.2 XPath (XML Path Language)

The main purpose of Xpath is to address XML document elements. It also provides the basic methods for manipulating strings, numbers, and booleans. XPath uses a concise, non-XML syntax to facilitate the use of XPath within URIs and XML attribute values. XPath models an XML document as a tree of nodes, with different types of nodes, including element nodes, attribute nodes, and text nodes. XPath defines a `text()` method to calculate the string value of each type of node. Some node types also have names. XPath fully supports XML namespaces. In this way, the node's name is modeled as a pair consisting of a local part and a possibly empty namespace URI[12]. XPath syntax used in this project is explained in Appendix A.

2.3 MongoDB

MongoDB is an open-source document-oriented database based on horizontal scale-out architecture instead of traditional table structures like relational databases. Each data record is a document described in a JSON-like binary representation format called BSON, and can be retrieved by applications in JSON format. This format directly maps to native objects in most modern programming languages without extra normalization process, therefore it is favoured by many developers[7]. The usage of MongoDB query statements db. collection . find () and db. collection .findOne() are explained in Appendix A.

2.4 JSON (JavaScript Object Notation)

JSON is a lightweight data exchange format. The JSON format is relatively simple, easy to read and write, the format is compressed. JSON objects are easy to parse, client-side JavaScript can simply read JSON data through eval(). Because the JSON format can be used directly for the server-side code, it significantly simplifies the development on both the server-side and the client-side, and is easy to maintain. However, it is not as popular and widely used as the XML format, and it is not as versatile as XML. The promotion of JSON format in Web Services is still in its early stage[2].

2.5 Comparing JSON and XML

JSON and XML are both file types of storing data. They are comparable in readability and extensibility, and they both have rich parsing methods. However, there are still many differences between them listed in the table below.

Aspect	JSON	XML
Programming Difficulty	Programming in JSON is obviously much easier than that of XML, even without tools, one can write JSON code easily.	XML has abundant coding tools, such as Dom4j, JDom, etc. JSON also has tools provided by JSON.org, but it is not easy to write XML well.
Decoding difficulty	The parsing difficulty of JSON is almost none.	The parsing of XML has to consider the parent node of the child node.
Popularity	Just started to be used in the industry	Widely used in the industry.
Data size	Smaller	Larger
Interaction	The interaction between JSON and JavaScript is more convenient.	The interaction between XML and XPath is more convenient.
Descriptiveness	Less	More
Execution speed	Faster	Slower

3 Related Work

There are many related projects that can query JSON files using XPath-like syntax implemented in different languages. However, most of them need an extra modification of original XPath query string and are lacking some supported features of XPath. More importantly, their intended use scenarios do not include connection to MongoDB, which makes our work significant. In this section, we introduce some outstanding related works. The performance evaluation between them and our work will be conducted at section 5.

3.1 node-xml2js-XPath

This library allows users to query the JSON object with XPath syntax through node-xml2js(a library that parses xml type documents to a JSON type result). Then users can search the transformed xml document by `evalFirst()`, `JSONText()` and `find()` functions. It only supports shorthand version of XPath query with descendent selectors, attribute selectors and tag value selectors[10].

3.2 XPath_JSON

XPath_JSON extracts data from JSON using XPath approach. It is a library implemented by Rajesh Kotari which tries to simplify the extraction of data from JSON using XPath like approach. To extract a value from JSON structure in python, we have to write `obj['key']` or `obj[index]` and it gets even more convoluted when we try to extract a value based on certain conditions. With XPath_JSON, users can search by `extract()` function with XPath format language, but there are still some special symbols different from XPath syntax[13].

3.3 JSONPath

JSONPath is a Java DSL(Domain Specific Language) for reading JSON documents like XPath for xml document. It expresses paths in the dot–notation or the bracket–notation and the root member object is always written as regardless if it is an object or array. Also, JSONPath has its own particular operators, functions and filter operators. As opposed to XPath, JSONPath's functions can only be invoked at the end of a path expression. There are three different ways to create filter predicates in JSONPath: Inner Predicates, Filter Predicates and Roll Your Own[5].

3.4 JSONQuery

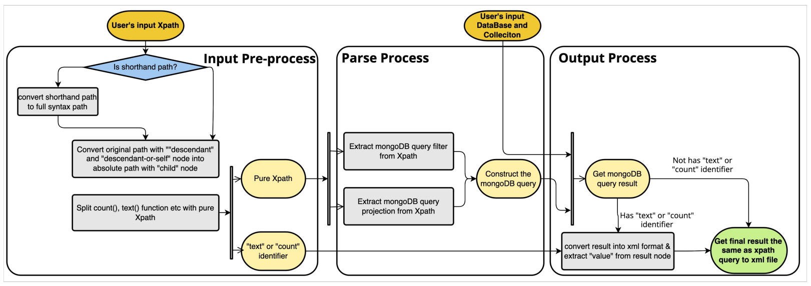
Zhengchun and Zhouzhuojie implemented a GO package for XPath query executions on JSON documents (JSONQuery for short). This program allows input to be an URL, a file, or a string and parses it into an XML tree of the input JSON object. A user can then use "Find" or "Findone" to fetch all or just one of the query results with XPath queries and the maintained XML tree. The final output is translated into an XML-like string[6].

4 Methodology

4.1 Overview

Almost all the previous work are based on querying JSON documents through XPath without querying MongoDB database. To achieve this, one approach is to retrieve the JSON document from MongoDB, and then perform a second query on the extracted result. However, this approach is cumbersome and error-prone in the middle steps. Moreover, this method is only feasible for lightweight JSON file, when the data volume gets large, the query speed is significantly impaired. Therefore, our work is contributed to achieve a library for searching MongoDB database directly, which is based on `find()` function to compile XPath into MongoDB query. According to our research, there is no previous work to implement XPath-type queries for MongoDB yet.

The architecture of our implementation is divided into three parts: input preprocess, parse process and output process. In the input preprocess, the input is sanitized to a full-syntax XPath query, with no functions and all axes are rewritten to child axes. In the parse process, the corresponding parts of "filter" and "projection" in MongoDB `find()` queries are retrieved from the XPath query, and then used to obtain the results from MongoDB. This stage mainly includes the analysis of conditional and relational statements. Finally, the output process includes the implementation for handling relevant functions parsed from the input preprocess and outputs the final results to users. The flow chart of the whole algorithm and implementation is as Figure 1



User's input Xpath
Input Pre-process
Is shorthand path?
convert shorthand path to full syntax path
Convert original path with ""descendant" and "descendant-or-self" node into absolute path with "child" node
Split count(), text() function etc with pure Xpath
Pure Xpath
"text" or "count" identifier
User's input DataBase and Colleciton
Parse Process
Extract mongoDB query filter from Xpath
Extract mongoDB query projection from Xpath
Construct the mongoDB query
Output Process
Get mongoDB query result
Not has "text" or "count" identifier
Has "text" or "count" identifier
convert result into xml format & extract "value" from result node
Get final result the same as xpath query to xml file

Figure 1: Overview of query process

To run our code and verify the result, the user needs to set up the MongoDB database and run "main.py". Main function ApplyMongoDBQuery(database, collection, XPathQuery, isFullSyntax) needs four input parameters. database specifies the name of the database, collection represents the collection name, XPathQuery represents the XPath statement and isFullSyntax represents whether the XPath is in full syntax or shorthand, since our algorithm supports both, the users need to inform the system the syntax they choose.

4.2 Input Preprocess

Before the parsing stage, the query is first preprocessed to examine if it contains built-in XPath functions and whether the input is in shorthand or full syntax.

4.2.1 Function Split

Our library only supports independent count() and text() functions (i.e., they cannot be part of the predicate). We utilized python's built-in regex library for pattern matching and process.

Matching Pattern	Input	Output
re.compile('ˆcount(.∗)')	count(child::library/child::album)	("child::library/child::album", "count")
re.compile('/(child::)?text\(\)$')	child::artist/child::name/child::text() child::artist/child::name/text()	("child::artist/child::name", "text") ("child::artist/child::name", "text)

A tuple is returned containing new sanitized query string and a string as a function indicator to acknowledge in the final stage that the output for this query should be further processed to handle the specified function. (See section 4.4.1 and 4.4.2)

4.2.2 Shorthand XPath Pre-process

If the input Xpath is a shorthand XPath query, the program will call the handle_concise.py script, to convert the shorthand XPath into full syntax XPath. This function is implemented by regular expression, and the matching mode is as follows:

Matching Pattern	Input	Output
re.compile('ˆ[a-z]+/')	library/	child::library
re.compile('ˆ/[a-z]+')	/artist	/child::artist
re.compile('ˆand\s[a-z]+')	[b <5 and c >6]	[child::b <5 and child::c >6]
re.compile('ˆor\s[a-z]+')	[b <5 or c >6]	[child::b <5 or child::c >6]
re.compile('ˆ[[a-z]+')	[b = 'A']	[child::b = 'A']
re.compile('ˆ//[a-z]+')	//b	/descendant::b

The above matching pattern covers all possible locations of nodes in shorthand XPath. Meanwhile, this function uses the pointer to traverse the shorthand XPath. Each node is matched using the above matching pattern, and the full syntax path is restored through the jump of pointer. A complete example is shown below:Input:album[age>=10]//artist[country='I']/name and Output: child :: album[child :: age>=10]/descendant::artist[child :: country='I']/child::name

4.2.3 Descendant Node Process

Since MongoDB do not support ambiguous queries, users must know the absolute path of the node in advance. However, XPath supports queries with the descendant and descendant-or-self nodes. To keep them consistent, we restore the XPath query containing the "descendant" or "descendant-or-self" nodes to an absolute path containing only the "child" node for processing. This part is implemented by convert_all_descendant_to_child (usage, XPath, database, collection, XPath_set). The parameter usage indicates whether this function handles the "descendant" node or the "descendant-or-self" node, XPath indicates the input

XPath query. And the parameter XPath_set is a set containing the result, it should be passed an empty set initially.

The function is demonstrated by an example XPathquery =`"child::library/child::album/descendant::title"`. in Figure 2

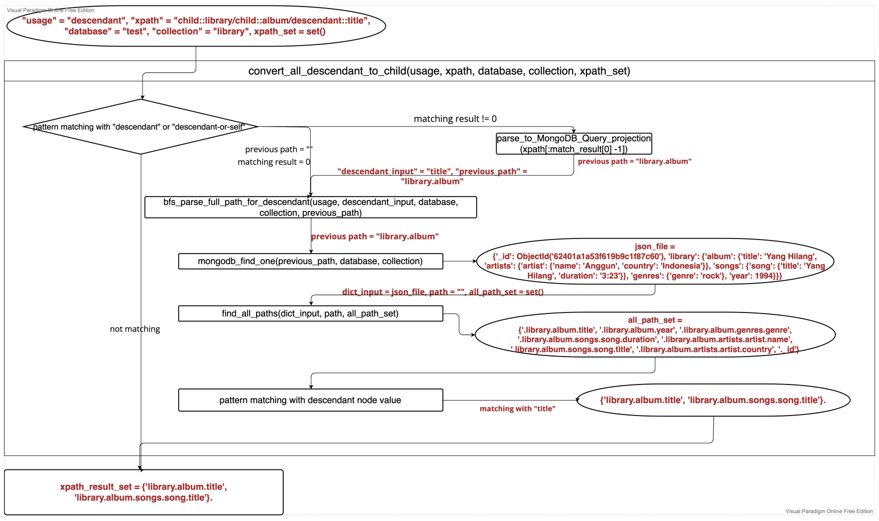

Visual Paradigm Online Free Edition
"usage" = "descendant", "xpath" = "child::library/child::album/descendant::title", "database" = "test", "collection" = "library", xpath_set = set()
convert_all_descendant_to_child(usage, xpath, database, collection, xpath_set)
pattern matching with "descendant" or "descendant-or-self"
matching result != 0
parse_to_MongoDB_Query_projection (xpath[:match_result[0] -1])
previous path = "library.album"
previous path = ""
matching result = 0
"descendant_input" = "title", "previous_path" = "library.album"
bfs_parse_full_path_for_descendant(usage, descendant_input, database, collection, previous_path)
previous path = "library.album"
mongodb_find_one(previous_path, database, collection)
json_file = {'_id': ObjectId('62401a1a53f619b9c1f87c60'), 'library': {'album': {'title': 'Yang Hilang', 'artists': {'artist': {'name': 'Anggun', 'country': 'Indonesia'}}, 'songs': {'song': {'title': 'Yang Hilang', 'duration': '3:23'}}, 'genres': {'genre': 'rock'}, 'year': 1994}}}
dict_input = json_file, path = "", all_path_set = set()
find_all_paths(dict_input, path, all_path_set)
not matching
all_path_set = {'.library.album.title', '.library.album.year', '.library.album.genres.genre', '.library.album.songs.song.duration', '.library.album.artists.artist.name', '.library.album.songs.song.title', '.library.album.artists.artist.country', '._id'}
pattern matching with descendant node value
matching with "title"
{'library.album.title', 'library.album.songs.song.title'}.
xpath_result_set = {'library.album.title', 'library.album.songs.song.title'}.
Visual Paradigm Online Free Edition

Figure 2: Conversion Process of Descendant Node

First, the pattern matching of “descendant” or “descendant-or-self” node is performed. If the match result is not empty, it will be further parsed. Noting that this function processes all descendant nodes one by one. In this example, the matched start and end locations of the descendant node are “match_result = (28, 45)”, which indicates that the path before the “descendant” node need to be saved, so, parse_to_MongoDB_Query_projection() in parse.py parses the path before the descendant node in XPath query to obtain the “previous_path”.

After parsing the previous path, We call bfs_parse_full_path_for_descendant (usage , descendant_input, database, collection , previous_path) function to restore the absolute path of the descendant node. “usage” indicates whether the function is used to retrieve the descendant node or the descendant−**or**−self node. “descendant_input” stores the matched descendant node value, in this example, “descendant_input” = “title”. “previous_path” is the result obtained in the previous step.

In the function of bfs_parse_full_path_for_descendant (), the first step is to get the JSON data structure from the database. Note that we only require the data structure under the previous path, that is, during the regular matching of the descendant keyword in the XPath query, if the matching result is 0, the projection in the findOne() function is the default value. If the matching result is nonzero, the part of XPath query before the “descendant” node will be seen as the projection of findOne() function and then get the final JSON structure. In the example above, the projection of findOne() function is ”‘library.album’ : 1”, and the obtained data structure ”json_file” is shown on the Fig.2.

Once we have got the tree structure, then call find_all_paths (dict_input , path, all_path_set) function to get the absolute paths of all nodes in the tree structure. ”dict_input” is tree structure for “json_file” obtained in the previous step. BFS algorithm is applied to traverse the tree, which shown on the appendix[1].

The BFS result will be stored into "all_path_set". Then the pattern matching `re.compile("(.+(?="+ descendant_input + "))")` is performed, therefore only paths with descendant_input are matched. In this example, the result is all_descendant_path = `"library.album.title", "library.album.song s.song.title"`. This set contains all unique paths with descendant nodes under the "library.album" path. In this way, our library supports "descendant" ambiguous matching.

This function is also responsible for restoring "descendant-or-self" nodes. In the restored result of path, only "descendant-or-self" axes include the current node that meets the matching conditions. For example, to query JSON structure { title { title ":"Tony}}, the XPath query child :: title /descendant:: title returns only one path `["title.title"]`, but child :: title /descendant−**or**−self::title will return a set of path `["title", "title.title"]`.

4.3 Parse Process

Our library aims to query MongoDB with XPath, but the query API supported by MongoDB needs to specify filter and projection. Therefore, to fill the gap between the two, how to generate corresponding filter and projection according to the XPath query statement entered by the user are very important.

It's difficult to generate a MongoDB query statement that is effectively equivalent to the XPath query, since XPath and MongoDB query are not 1:1 mapped. For example, XPath query support attribute query while in MongoDB there is no concept of attribute. For some other concepts in XPath query, it is still feasible to find the corresponding mapping and syntax in MongoDB. For example, the expression of filter and projection used in MongoDB query is similar to key value pair or JSON object. The expression of their keys are similar to the concept of path in XPath. The value to a key in the filter can be mapped to the predicate of a step in the XPath. It is worth noting that the predicates in XPath may be nested, which brings some difficulties to generating filters.

The main concern of the parse stage is how to find the filter and projection required by the corresponding MongoDB query based on the XPath query in the form of a complete path and full syntax. 4.3.1 focuses on generating MongoDB query projection, and 4.3.2 focuses on generating MongoDB query filter.

4.3.1 Extract MongoDB Query Projection

The XPath query obtained before parse process is the full syntax path, for example: /child :: library /child :: album[child:: artists [child :: artist /child :: name=`'Anang'`]]/child::artists/child :: artist [child :: country=`'Ind'`]/child::name. And the corresponding projection in MongoDB is library .album.artists. artist .name:1.

In XPath syntax, each query can be written as a path expression: /step1/ step2 /.../ stepn. Each step is in the form of axis :: node_Test[predicate], and steps are separated by slashes "/". In our current work, all axes(after input preprocessing) are child and node_Test can only be nodename, [predicate] is optional.

Based on the examples above, it is easy to understand our logic on obtaining the projection for MongoDB queries, we scan the XPath query from left to right and store the nodename in each step until the end of the XPath statement; i.e., the axis child:: and [predicate] are ignored for projection extraction.

This design can handle most cases except when predicates are combined and nested. In this case, it may be problematic when reading the content after axes in each step of the XPath input since the predicates are ignored. For example, XPath query statement is /child :: library /child :: album[child :: artists [child :: artist / child :: name=`'Anang'`]]/child::artists/child :: artist [child :: country=`'Ind'`]/child::name. The step2 of this query statement is child :: album[child :: artists [child :: artist /child :: name =`'Anang'`]]. As we mentioned before, when manipulating the projection, our design only focuses on the nodename "album" in step 2, other parts should not be considered. However, the predicate of step 2 is a combination of path and predicates: [child :: artists [child :: artist /child :: name=`'Anan'`]].

If there is no special handling for nested predicates, the algorithm reads child :: artists [child :: artist /child :: name=`'Anang'`], and uses the "artists" obtained from step 2's predicate to compute the projection, then obtains an incorrect final result as library .album.artists. artists . artist .name:1. It is because the program misreads "artists" as the nodename of step2, instead, it is the predicate node name of step2. To resolve this issue, we designed a special process model. It correctly parses XPath query statements from left to right and generate the projection used in MongoDB query. The design is showed as Figure 3

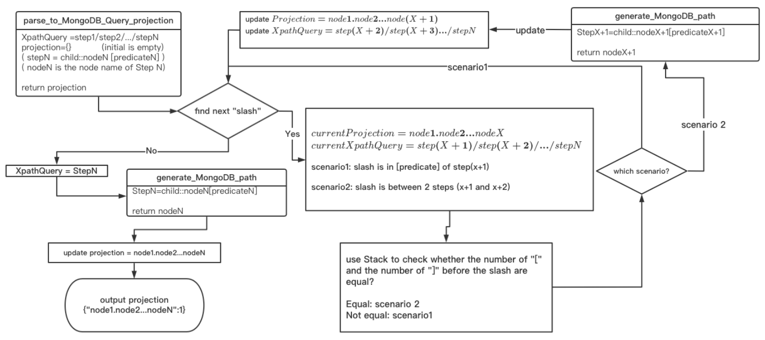

parse_to_MongoDB_Query_projection
XpathQuery =step1/step2/.../stepN
projection={} (initial is empty)
(stepN = child::nodeN [predicateN])
(nodeN is the node name of Step N)
return projection
update $Projection = node\mathbf{1}.node\mathbf{2}...node(X+1)$
update $XpathQuery = step(X+\mathbf{2})/step(X+\mathbf{3}).../stepN$
update
generate_MongoDB_path
StepX+1=child::nodeX+1[predicateX+1]
return nodeX+1
scenario1
find next "slash"
Yes
$currentProjection = node\mathbf{1}.node\mathbf{2}...nodeX$
$currentXpathQuery = step(X+\mathbf{1})/step(X+\mathbf{2})/.../stepN$
scenario1: slash is in [predicate] of step(x+1)
scenario2: slash is between 2 steps (x+1 and x+2)
scenario 2
which scenario?
No
XpathQuery = StepN
generate_MongoDB_path
StepN=child::nodeN[predicateN]
return nodeN
update projection = node1.node2...nodeN
output projection
{"node1.node2...nodeN":1}
use Stack to check whether the number of "[" and the number of "]" before the slash are equal?
Equal: scenario 2
Not equal: scenario1

Figure 3: Projection Generate Process

As shown in the figure above, parse_to_MongoDB_query_Projection() takes the sanitized XPath query as a parameter. Since the steps in the XPath statement are separated by slashes, to correctly match the beginning and end positions of each step, we identified the pattern using regular expressions and iterative statements. Whenever a slash is found, its location can be classified into two cases. In the first case, the position of the slash is in the predicate corresponding to a step. Then we should continue to find the next slash which is used to separate each step of the input XPath query. In the second case, the found slash is just in the middle of two steps to separate them for the input XPath query. At this time, we should split the XPath query with this slash to get a "step", match the corresponding node name, then update the projection path and continue to process the remaining XPath queries in the same way.

One of the important stages is how to judge whether the slash found time belongs to the first case or the second case. With the help of "stack", we read the string on the left side of slash from left to right. If it encounters "[" push the stack, if it encounters "]", it will pop the stack. Finally, by judging whether the stack is empty, we can judge whether the position of slash belongs to scenario

1 or scenario 2. When the stack is empty, it indicates that the number of "[" and the number of "]" on the left side of the slash are matched. At this time, the slash is just between two steps, which is scenario 2. Conversely, the slash is located in the predicate of the step, which is the scenario1. As mentioned in the figure above, according to our model the final output of the projection based on a XPath query statement is formed as "{node1.node2..."nodeN:1}.

4.3.2 Extract MongoDB Query Filter

The process of finding the filter is similar to the process of calculating the projection, but more complex. Whether the predicate in XPath query or the filter in MongoDB query, its main purpose is to add some filtering conditions to a node. If there is a predicate in a node in an XPath query, there must be a key value pair in the filter of MongoDB, which plays a role in filtering the same node with the same semantics. Therefore, in order to calculate the filter in MongoDB according to the XPath query. We need to parse each predicate in the XPath query to get each key value pair that ultimately forms the filter.

We found that when a step path has a predicate, we can analyze the predicate of the step path separately and consider the previous path before the step path. Then, we can get the corresponding expression of this predicate in MongoDB filter. Therefore, when we generate the corresponding filter statement according to XPath query, we will process the XPath query from left to right and process the predicate of each step path. At the same time, the previous path is continuously recorded and updated.

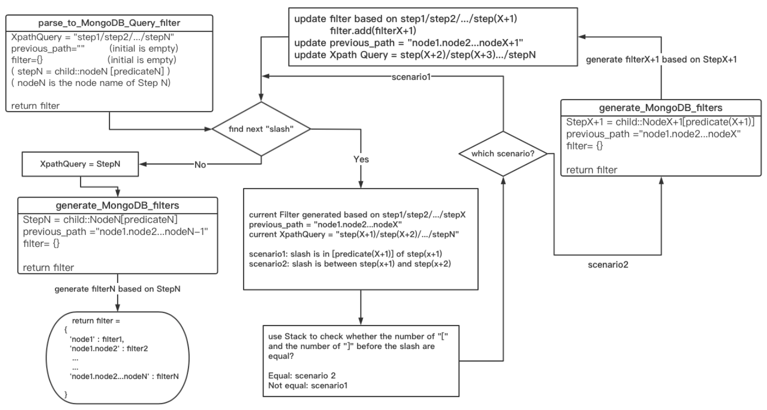

parse_to_MongoDB_Query_filter
XpathQuery = "step1/step2/.../stepN"
previous_path="" (initial is empty)
filter={} (initial is empty)
(stepN = child::nodeN [predicateN])
(nodeN is the node name of Step N)
return filter
update filter based on step1/step2/.../step(X+1)
filter.add(filterX+1)
update previous_path = "node1.node2...nodeX+1"
update Xpath Query = step(X+2)/step(X+3).../stepN
generate filterX+1 based on StepX+1
scenario1
find next "slash"
No
Yes
XpathQuery = StepN
which scenario?
generate_MongoDB_filters
StepX+1 = child::NodeX+1[predicate(X+1)]
previous_path ="node1.node2...nodeX"
filter= {}
return filter
generate_MongoDB_filters
StepN = child::NodeN[predicateN]
previous_path ="node1.node2...nodeN−1"
filter= {}
return filter
current Filter generated based on step1/step2/.../stepX
previous_path = "node1.node2...nodeX"
current XpathQuery = "step(X+1)/step(X+2)/.../stepN"
scenario1: slash is in [predicate(X+1)] of step(x+1)
scenario2: slash is between step(x+1) and step(x+2)
scenario2
generate filterN based on StepN
return filter =
{
'node1' : filter1,
'node1.node2' : filter2
...
...
'node1.node2...nodeN' : filterN
}
use Stack to check whether the number of "["
and the number of "]" before the slash are
equal?
Equal: scenario 2
Not equal: scenario1

Figure 4: Filter Generate Process

The design of the model is shown in Figure 4. parse_to_MongoDB_query_filter() method takes the sanitized XPath query as a parameter. Similar to the process in 4.3.1, we still need to find the matching slashes otherwise we will get wrong results in recursion and iteration. The locations for matched slashes can also be classified into two scenarios shown in Figure 4. Every time a new step is matched, the generate_MongoDB_filters() function is invoked. The function maps the predicate of the step into a key-value pair in filter.

When processing filters, different from 4.3.1, we only need to extract the node name after "child::" by regular expression every time a step is matched. However, predicates in each step may be nested, or there can be complicated predicates connected by `"and"`. For example: XPathQuery=`"child::library/child::album [child::artists/child::artist[child::age>=20 and child::age<=30] and child::artists /child::artist[child::country='Ind']]/child::title"`. The predicate of step 2 is connected by two sub predicates. When processing, we intuitively split these sub-predicates by "and" and process them separately. However, we should note that "and" may be used to connect conditions in addition to nested predicates cases. For example, child :: artists /child :: artist [child :: age>=20 **and** child::age<=30] the "and" is used to connect 2 conditions age>=20, age<=30.

Therefore, when splitting the nested sub-predicates, we should note that child :: artists /child :: artist [child :: age>=20 **and** child::age<=30] **and** child::artists/child:: artist [child :: country=`'Ind'`] should not be splitted to child :: artists /child :: artist [child :: age>=20"; `"child::age<=30]`; child :: artists /child :: artist [child :: country=`'Ind'`]. The solution to this problem is similar to classifying matched "/" discussed in 4.3.1. The only difference is we check the position of "and" instead of "/".

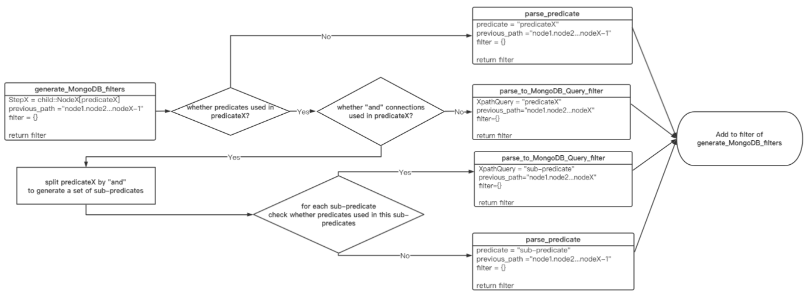

parse_predicate
predicate = "predicateX"
previous_path ="node1.node2...nodeX−1"
filter = {}
return filter
No
generate_MongoDB_filters
StepX = child::NodeX[predicateX]
previous_path ="node1.node2...nodeX−1"
filter = {}
return filter
whether predicates used in predicateX?
Yes
whether "and" connections used in predicateX?
No
parse_to_MongoDB_Query_filter
XpathQuery = "predicateX"
previous_path="node1.node2...nodeX"
filter={}
return filter
Add to filter of generate_MongoDB_filters
Yes
split predicateX by "and" to generate a set of sub-predicates
parse_to_MongoDB_Query_filter
XpathQuery = "sub-predicate"
previous_path="node1.node2...nodeX"
filter={}
return filter
Yes
for each sub-predicate check whether predicates used in this sub-predicates
No
parse_predicate
predicate = "sub-predicate"
previous_path ="node1.node2...nodeX−1"
filter = {}
return filter

Figure 5: Filter Generate Process

In Figure 5, if the input parameter "step" in generat_MongoDB_filters() does not contain nested predicates, it will call parse_predicate() directly to handle different conditions and parses conditions in the predicate into key-value pairs in the filter. We currently support 6 operators: $<$, $<=$, $>$, $>=$, $=$ and $!=$.

As shown in Figure 5 , if there is a nested predicate, we still need to consider whether "and" connects multiple sub-predicate as a connection operator.

First case: "and" is not present. For example step=child::album[child:: artists [child :: artist /child :: name=`'Anang'`]]. In this case, child :: artists [child :: artist /child :: name=`'Anang'`] can be seen as a new XPath query and recursively call parse_to_Mong −oDB_query_filter, then pass previousPath = 'library.album' as parameter in it. The result returned by the functions is the final filters generated.

Second case: "and" is present as a connection operator. For example, step= child::album[child:: artists /child :: artist [child :: age>=20 **and** child::age<=30] **and** child:: −artists/child::artist [child :: country=`'Ind'`]]. "and" connects child :: artists /child :: artist [child :: age>=20 **and** child::age<=30], child :: artists /child :: artist [child :: country=`'Ind'`]. In this case, a stack is used to match "[" and "]" and only split "and" that connects two sub-predicates.The example above is split into 2 sub-predicates. child :: album[child:: artists /child :: artist [child :: age>=20 **and** child::age<=30] and child :: artists /child :: artist [child :: country=`'Ind'`]]. Then we call parse_predicate() or parse_to_MongoD −BQuery_Filter() depending on if the sub-predicates are still nested. This model can correctly handle the multi-layer of "and" and nested predicates of a step.

The more nested layers, the more recursive calls.

4.4 Output Process

After the preprocess and parsing, the result we have obtained should be a MongoDB cursor of the find function. Parsing the cursor, a list of result objects can be located and extracted. However, such a result like |{'_id': {'oid': '623 f46229083253fb26c6c[0'}, 'library': {'album': { 'songs': {'song': [{'title': 'Timang'}, {'title': 'Miliki'}]}}}},{'_id': {'oid': '623f46229083253fb26c6ca2'}, 'library': {'album': {'songs': {'song': [{'title': 'Separuh'}]}}}}| may not match the expectation.

This result contains a lot of extra information that the user do not care about. More importantly, MongoDB does not provide the equivalent methods for XPath count and text functions. Therefore, further processing is needed to implement the two functions and prettify the output. The main function for this stage is finalOutput(cursor,projection,operator,pretty=True)

4.4.1 Count Function

For the count function, we have defined two use cases. The first one is to directly count the number of final results, and the second case is to include 'count()' in the predicate. Currently, only the former one is supported. For example, MongoDB_query = db.library.find({'library.album.artists.artist.name': 'Anang Ashanty '},{'library.album.songs.song.title': 1}).count(). XPathQuery = count(doc("library.xml")/child::library/child::album[child::artists/child:: artist /child :: name='Anang Ashanty']/child::songs/child::song/child::title). Projection = {'library.album.songs.song.title': 1}.
As mentioned above, parsing the cursor yields a list of dictionaries or nested dictionaries, with each layer's key implied by the MongoDB projection. The tricky part here is MongoDB's count function works differently from XPath, it only counts the number of result objects. If we simply call MongoDB's built-in count function, it will return 3 for the example above, however, the expected output is 6 from XPath.

If the operator parameter is "count", it will invoke a helper function handleCount to flatten the layers of cursor result and recursively read each JSON object until the desired key is reached and output the number of value objects to the key.

4.4.2 Text Function

The text function in XPath is used to output the text nodes and can be identified as /child :: text() or /text(). If the output nodes are not text type, XPath returns None or a list of numbers according to the number of output nodes. In our compiler, we consider this case to be an illegal input and the result is None.

When the operator parameter is "text", it invokes a recursive helper function handleText attempting to extract the text labels. For each object in the MongoDB result, we call text(**object**[first_key]), and append the text value or text value list to text_result then flatten the text_result list to return a clean result.

4.4.3 XML Output

The last step is to transform the output into a more user-friendly XML format. Specifically, the result can be divided into three types according to the

XPath operator. The following table describes all the output formats where the essential information is wrapped by square brackets.

Method	Output Format
text()	<result>[text1],[text2],...</result>
count()	<result>[number]</result>
normal searching	<result>[<node_tag1> <node_tag2>text</node_tag2> <node_tag2>text</node_tag2> </node_tag1>]</result>

The output process first examines the XPath function name passed by the preprocessing part, then invokes the corresponding helper functions. It transfroms the received result value into a object list, and append every child node to a result tree and print the final result with Python xml.etree.ElementTree module.

When the XPath operator passed by the preprocess program is null, which means the result should be a series of element nodes, the program only retains the descendants of the target result tag. They are then transformed from dictionary objects into XML nodes with dicttoxml Python module. Otherwise, if the target values are simple strings, which means there are no children within them, they will be used as the target tag's text value.

Having the XML tree parsed by the above functions, the last work to do is to prettify the XML tree with toprettyxml method.

5 Performance Evaluation

5.1 Experimental Set-up

In this section, a set of experimental XPath queries will be used to test the performance of our work, XPath itself, and related works. The related works are notated as JSONPath[5], XPath_JSON[13], and JSONQuery[6] respectively. A test library of music albums in both JSON and XML format will be used, which are `'test-library.JSON'` and `'test-library.xml'`. While XPath and our work can executes XPath syntax queries directly from the test library files, the other methods need some modification for the library and the queries, which are discuessed in the next subsection. Our work uses Python 3.9 and XPath is run in XQuery version 3.1. JSONPath requires JDK 1.7 or above. XPath_JSON requires Python 3.7 or above. JSONQuery experimental environment is Golang 1.14.

5.2 Results

Indeed, it is hard to apply all the XPath features to JSON files since the JSON files have different features from XML files in nature. To clarify these differences, extensive test cases have been conducted. The execution results of five works are compared in Appendix B, including our work, XPath, JSONPath, XPath_JSON, and JSONQuery. The features of each work concluded from the test cases are compared in section 5.3.

5.2.1 JSONPath Result Analysis

JSONPath can't process a JSONArray object directly in the first step but our test-library.JSON document is of JSONArray type, so we added a new key data to transform the data into a JSON object. It has to use [*] to deal with the JSONArray object. However, this leads to another problem. JSONPath can not express Example 5 of Appendix B by [*] for the simple reason that the filtered artist object is irregular, a JSONArray object of a combination of JSON Arrays and JSON objects. JSONPath library may omit some qualified results that should not be removed if the same query is process in XPath. The integrity of the query results may not always be preserved, as is indicated in the Example 8 and 9 of Appendix B. Moreover, JSONPath returns text contents if the result nodes are text type. So it loses tree structure automatically as shown in Example 10 of Appendix B. Finally, JSONPath uses length() at the tail of a query path to calculate the length of a result array. In Example 11 of Appendix B, the result is a list of length of each JSONArray object, rather than the final count result like XPath and our compiler.

5.2.2 XPath_JSON Result Analysis

Like JSONPath, XPath_JSON can not directly handle a JSONArray object in the first step, so we transformed the data into a JSON object in the same way mentioned above. XPath_JSON should support equality, inequality and comparison operators like Example 4 of Appendix B, but it has problem when parsing long query more than 3 node steps as is shown in Example 2, 3 and 5. This library do not support complicated queries with and, or, not operators, not to mention the multiple predicates and nested predicates. XPath_JSON always returns text contents if the result nodes are text type, just like JSONPath.

5.2.3 JSONQuery Result Analysis

JSONQuery can not analyze a list of objects at the first step either. A special modification is applied, which removes the repeated library and album tags and combines all the objects into the album section. According to the implementation, the query strings are also modified by inserting /*/ to indicate the object list in the album section. The executed queries in Appendix B show this modification. As shown in example 6 and 7 in Appendix B, JSONQuery only support ordinary path parsing and operators =, !=, **and**, **or**. It does not support most of the algorithmic operators, count, multiple predicates, and nested predicates. Note that this work does not pass out test cases for **and** and supports these operators in specific cases.

5.3 Discussion

As the Appendix B test cases indicate, a table of the supported features for each work is concluded. All the listed features are realized in our work. We support both full syntax and shorthand XPath query. Users don't have to learn a new query language when they use our compiler, while other three works all need their own query syntax. Our compiler can identify basic axes: child, descendant and descendant-or-self, whereas other works generally do not support all of them.We also support multiple predicates and nested predicates the same as XPath query. Here only JSONPath can deliver such complicated queries.General operators like equality, inequality, comparison, and/or/not operators are available in our compiler. JSONQuery and XPath_JSON can not support all these operators. Two functions count and text are supported in our work. We can calculate the number of result nodes and output the text nodes the same as XPath. While JSONPath, JSONQuery and XPath_JSON always return text contents if the result nodes are text type. Only our work is able to print the output in both JSON and XML format.

Supported Feature	Our Work	XPath	JSONPath	JSONQuery	XPath_JSON
Full Syntax and shorthand	Yes	Yes	NA	NA	NA
Tree Result (remain tree structure in the result)	Yes	Yes	Yes	No	Yes
Axes: child, descendant, descendant-or-self	Yes	Yes	NA	Yes (but do not support descendant-or-self)	NA
multiple predicates	Yes	Yes	Yes	No	No
nested predicates	Yes	Yes	Yes	No	No
operators in predicates: =, !=, <, <=, >, >=, and, or, not	Yes	Yes	Yes (== for equality, && for and, || for or, ! for not)	Yes (only supports =, !=, and, or)	Yes (but do not support and, or, not)
count()	Yes	Yes	Yes (but the count output is for each JSON object)	No	Yes
text()	Yes	Yes	NA (When the result nodes are text type, JSONPath returns text contents by default.)	Yes	NA (When the result nodes are text type, JSONPath returns text contents by default.)
result in XML type	Yes	Yes	No	No	No
result in JSON type	Yes	No	No	No	No

6 Conclusion

To facilitate a smooth interaction between XML databases and MongoDB, we have implemented the "XPath compiler for MongoDB" library by Python that supports direct XPath query executions in MongoDB. The highlight of our work is the capability of executing both shorthand and full-syntax XPath queries against MongoDB datasets. Comparing with related work discussed in this paper, our library can be distinguished from them with the extra support for multiple predicates and nested predicates handling as well as flexible output format.

Due to the limited time given on this project, we do realize there are limitations of our project. The architecture of this project allows future extensions to be added easily. We would like to extend the project to support union operator | between two nodes, this can be easily implemented by combining the results of two queries. Additional support for aggregate functions such as **sum**(), avg(), **min**() and **max**() would also be beneficial for our users. Our project is able to preprocess the query with any element operator $*$, and we plan to finish this feature in the parsing stage later.

References

[1] Breadth-first Search. https://en.wikipedia.org/wiki/Breadth-first_search.

[2] JSON. https://www.json.org/json-en.html.

[3] JSON vs XML. https://hackr.io/blog/json-vs-xml.

[4] JSON wiki. https://en.wikipedia.org/wiki/JSON.

[5] JsonPath. https://github.com/json-path/JsonPath.

[6] JsonQuery. https://github.com/antchfx/jsonquery.

[7] MongoDB. https://www.mongodb.com/why-use-mongodb.

[8] MongoDB db.collection.find. https://www.mongodb.com/docs/manual/reference/method/db.collection.find/.

[9] MongoDB db.collection.find. https://www.mongodb.com/docs/manual/reference/method/db.collection.findOne/.

[10] Node Xml2js XPath. https://github.com/dsummersl/node-xml2js-xpath.

[11] XML. https://developer.mozilla.org/en-US/docs/Web/XML.

[12] XPath. https://developer.mozilla.org/en-US/docs/Web/XPath.

[13] Xpath_JSON. https://github.com/kotari/xpath_json.

Appendix A

XPath Syntax

Path Syntax. "/" represents selecting from the root node. "//" represents selecting from the current node that matches the selection criteria no matter where the nodes are. "*" matches any element node, it is used as a wildcard here like in many other languages.

Axis. An axis represents the relation to the current node, it is used to find relative nodes to the current node in the XML file. The axis part can be omitted when writing in XPath shorthand. In the code, we will implement the following three most useful axis. "child::" selects the children of the current node "descendant::" selects all descendants, including the children, grandchildren, etc, of the current node. "descendant-or-self::" selects all descendants of the current node and the current node itself. Note that shorthand form it is "//".

Predicates. Predicates are used as filters to find specific nodes based on the conditions defined, they are written inside square brackets "[]"

Operators. "|" computes the union of the operands, it can only be operated on two node-sets. XPath also supports common logical operators in the predicates: "and", "or", "not". It also provides support for comparison operators: "<", ">", "≥", "≤", "=" and "!=".

Functions. XPath offers some useful built-in functions. "count()" returns the number of nodes returned from the query. "text()" returns only the text element of the node

MongoDB Query Statement

db.collection.find(filter, projection) The function returns a cursor to the documents that match the "filter" criteria. When the find()=method "returns documents", the method is actually returning a cursor to the documents[8].

Parameter	Type	Description
filter	document	Optional. Specifies selection filter using query operators. To return all documents in a collection, omit this parameter or pass an empty document ({}).
projection	document	Optional. This parameter determines which fields are returned in the matching documents. To return all fields in the matching documents, omit this parameter. This parameter determines which fields are returned in the matching documents. It takes a document in the form {<field1>:<value>, <field2>:<value>...}

db.collection.findOne(filter, projection) The two parameters(query and projection) are the same as described in section 2.3.1. It is different from the find() function in the sense that findOne() only returns one document that satisfies the condition. If there are multiple documents satisfy the condition, it returns the first one according to the natural order which reflects the order of documents on the disk. The function returns null if no satisfying document is found[9].

Appendix B

Example 1 Leaf Nodes

Work	Executed Query	Result
XPath	doc("test-library.xml")/ child::library/child::album /child::title	<result> <title>Bua Hati</title> <title>Separuh Jiwaku Pergi </title> <title>Yang Hilang</title> <title>Fantastic</title> <title>Sanggar Mustika </title> <title>No War</title> </result>
Our Work	child::library/child::album /child::title	<?xml version="1.0" ?> <result> <title>Yang Hilang</title> <title>Bua Hati</title> <title>Separuh Jiwaku Pergi </title> <title>Sanggar Mustika</title> <title>No War</title> <title>Fantastic</title> </result>
JsonPath	$.data[*].library.album.title	["Bua Hati", "Separuh Jiwaku Pergi", "Yang Hilang", "Fantastic ", "Sanggar Mustika", "No War"]
Xpath_json	data/library/album/title	['Bua Hati', 'Separuh Jiwaku Pergi', 'Yang Hilang', 'Fantastic', 'Sanggar Mustika', 'No War']
JsonQuery	library/album/title	Bua Hati,Separuh Jiwaku Pergi, Yang Hilang,Fantastic,Sanggar Mustika,No War

Example 2 Predicates of equality to string

Work	Executed Query	Result
XPath	doc(”test-library.xml”)/child:: library/child::album/child:: artists/child::artist[child:: country=’Indonesia’]/ child::name	<result> <name>Anang Ashanty</name> <name>Kris Dayanti</name> <name>Anang Ashanty</name> <name>Anggun</name> </result>
Our work	child::library/child::album/ child::artists/child::artist [child::country=’Indonesia’] /child::name	<?xml version=”1.0” ?> <result> <name>Anang Ashanty</name> <name>Kris Dayanti</name> <name>Anang Ashanty</name> <name>Anggun</name> </result>
JsonPath	$.data[*].library.album.artists. artist[?(@.country== ’Indonesia’)].name	[”Anang Ashanty”, ”Kris Dayanti”, ”Anang Ashanty”, ”Anggun”]
Xpath_json	data/library/album/artists/ artist/#[country==Indonesia]	None
JsonQuery	library/album/*/artists/artist [country=’Indonesia’]/name	Anang Ashanty,Anggun

Example 3 Predicates of not equality to string

Work	Executed Query	Result
XPath	doc(”test-library.xml”)/child:: library/child::album/child:: artists/child::artist[child:: country!=’Indonesia’]/ child::name	<result> <name>Wham!</name> <name>Siti Nurhaliza</name> <name>Job Bunjob Pholin</name> </result>
Our work	child::library/child::album/ child::artists/child::artist [child::country!=’Indonesia’] /child::name	<?xml version=”1.0” ?> <result> <name>Wham!</name> <name>Siti Nurhaliza</name> <name>Job Bunjob Pholin</name> </result>
JsonPath	$.data[*].library.album.artists. artist[?(@.country!= ’Indonesia’)].name	[”Wham!”, ”Siti Nurhaliza”, ”Job Bunjob Pholin”]
Xpath_json	data/library/album/artists/artist /#[country!=Indonesia]	None
JsonQuery	library/album/*/artists/artist [country!=’Indonesia’]/name	Wham!,Siti Nurhaliza,Job Bunjob Pholin

Example 4 Predicates of equality to number

Work	Executed Query	Result
XPath	doc(”test-library.xml”)/child::library/child::album[child::year=1994]/child::artists/child::artist/child::name	<result><name>Anggun</name></result>
Our work	child::library/child::album[child::year=1994]/child::artists/child::artist/child::name	<?xml version=”1.0” ?><result><name>Anggun</name></result>
JsonPath	$.data[*].library.album[?(@.year==1994)].artists.artist.name	[”Anggun”]
Xpath_json	data/library/album/#[year==1994]/artists/artist/name	[’Anggun’]
JsonQuery	library/album[/*/year=1994]/*/artists/artist/name	/

Example 5 Predicates of comparison operators

Work	Executed Query	Result
XPath	doc(”test-library.xml”)/child::library/child::album[child::year>=1990]/child::artists/child::artist/child::name	<result><name>Anang Ashanty</name><name>Kris Dayanti</name><name>Anang Ashanty</name><name>Anggun</name><name>Job Bunjob Pholin</name></result>
Our work	child::library/child::album[child::year>=1990]/child::artists/child::artist/child::name	<?xml version=”1.0” ?><result><name>Anang Ashanty</name><name>Kris Dayanti</name><name>Anang Ashanty</name><name>Anggun</name><name>Job Bunjob Pholin</name></result>
JsonPath1	$.data[].library.album[?(@.year >= 1990)].artists.artist.name	[”Anang Ashanty”, ”Anggun”, ”Job Bunjob Pholin”]
JsonPath2	$.data[].library.album[?(@.year >= 1990)].artists.artist[*].name	[”Anang Ashanty”, ”Kris Dayanti”]
Xpath_json	data/library/album/#[year>=1990]/artists/artist/name	None
JsonQuery	library/album[year>=1990]//artists/artist/name	/

Example 6 Predicates of operator and

Work	Executed Query	Result
XPath	doc("test-library.xml")/child::library/child::album[child::year>=1990 and child::year <=2000]/child::artists/child::artist[child::country='Indonesia']/child::name	<result> <name>Anang Ashanty</name> <name>Kris Dayanti</name> <name>Anang Ashanty</name> <name>Anggun</name> </result>
Our work	child::library/child::album [child::year>=1990 and child::year<=2000]/child::artists/child::artist[child::country='Indonesia']/child::name	<?xml version="1.0" ?> <result> <name>Anang Ashanty</name> <name>Kris Dayanti</name> <name>Anang Ashanty</name> <name>Anggun</name> </result>
JsonPath	$.data[*].library.album [?(@.year>=1990 && @.year<=2000)].artists.artist [?(@.country=='Indonesia')]. name	["Anang Ashanty", "Kris Dayanti", "Anang Ashanty", "Anggun"]
Xpath_json	/	/
JsonQuery	library/album[year>=1990 and year<=2000]/*/artists/artist[country='Indonesia']/name	/

Example 7 Predicates of operator or

Work	Executed Query	Result
XPath	doc("test-library.xml")/child::library/child::album [child::year>=1990 or child::year Anang Ashanty Kris Dayanti Anang Ashanty Anggun	<result> <name>Anang Ashanty</name> <name>Kris Dayanti</name> <name>Anang Ashanty</name> <name>Anggun</name> </result>
Our Work	child::library/child::album [child::year>=1990 or child::year	<?xml version="1.0" ?> <result> <name>Anggun</name> <name>Anang Ashanty</name> <name>Kris Dayanti</name> <name>Anang Ashanty</name> </result>
JsonPath	$.data[*].library.album[?(@.year>=1990 \|\|@.$year <= 2000$)].artists.artist[?(@.country =='Indonesia')].name	["Anang Ashanty", "Kris Dayanti", "Anang Ashanty", "Anggun"]
Xpath_json	/	/
JsonQuery	library/album[year>=1990 or year<=2000]/*/artists/artist [country='Indonesia']/name	/

Example 8 Nested predicates

Work	Executed Query	Result
XPath	doc(”test-library.xml”)/ child::library/child::album [child::artists/child:: artist[child::name=’Anang Ashanty’] and child::artists /child::artist[child:: country=’Indonesia’]]/child ::title	<result> <title>Bua Hati</title> <title>Separuh Jiwaku Pergi</title></result>
Our Work	child::library/child::album [child::artists/child::artist [child::name=’Anang Ashanty’] and child::artists/child:: artist[child::country= ’Indonesia’]]/child::title	<?xml version=”1.0” ?> <result> <title>Bua Hati</title> <title>Separuh Jiwaku Pergi</title> </result>
JsonPath	$.data[*].library.album[?(@.artists.artist.country == ’Indonesia’ && @.artists. artist.name==’Anang Ashanty’)].title	[”Separuh Jiwaku Pergi”]
Xpath_json	/	/
JsonQuery	library/album[artists[/*/ artist/name=’Anang Ashanty’] and /artists/*/artist[country =’Indonesia’]]/title	/

Example 9 Nested predicates with operator and

Work	Executed Query	Result
XPath	doc(”test-library.xml”)/ child::library/child::album [child::artists/child::artist [child::name='Anang Ashanty'] and child::artists/child:: artist[child::country= 'Indonesia']]/child::title	<result> <title>Bua Hati</title> <title>Separuh Jiwaku Pergi</title> </result>
Our Work	child::library/child::album [child::artists/child::artist [child::name='Anang Ashanty'] and child::artists/child:: artist[child::country= 'Indonesia']]/child::title	<?xml version=”1.0” ?> <result> <title>Bua Hati</title> <title>Separuh Jiwaku Pergi</title> </result>
JsonPath	$.data[*].library.album[? (@.artists.artist.country == 'Indonesia' && @.artists. artist.name=='Anang Ashanty')].title	[”Separuh Jiwaku Pergi”]
Xpath_json	/	/
JsonQuery	library/album[artists[/*/ artist/name='Anang Ashanty'] and /artists/*/artist [country='Indonesia']]/title	/

Example 10 Text method

Work	Executed Query	Result
XPath	doc(”test-library.xml”)/ child::library/child::album /child::title/child::text()	<result>Bua HatiSeparuh Jiwaku PergiYang HilangFantasticSanggar MustikaNo War</result>
Our Work	child::library/child::album /child::title/child::text()	<?xml version=”1.0” ?> <result>Yang Hilang,Bua Hati,Separuh Jiwaku Pergi, Sanggar Mustika,No War,Fantastic </result>
JsonPath	$.data[*].library.album.title	[”Bua Hati”, ”Separuh Jiwaku Pergi”, ”Yang Hilang”, ”Fantastic ”, ”Sanggar Mustika”, ”No War”]
Xpath_json	data/library/album/title	['Bua Hati', 'Separuh Jiwaku Pergi', 'Yang Hilang', 'Fantastic ', 'Sanggar Mustika', 'No War']
JsonQuery	library/album/*/title/text()	Bua Hati,Separuh Jiwaku Pergi, Yang Hilang,Fantastic,Sanggar Mustika,No War

Example 11 Count method

Work	Executed Query	Result
XPath	count(doc(”library.xml”)/ child::library/child::album [child::artists/child::artist /child::name=’Anang Ashanty’] /child::songs/child::song)	<result>6</result>
Our Work	count(child::library/child:: album[child::artists/child:: artist/child::name=’Anang Ashanty’]/child::songs/child ::song)	<?xml version=”1.0” ?> <result>6</result>
JsonPath	$.data[*].library.album[? (@..name anyof [’Anang Ashanty’])].songs.song.length()	[3, 3]
Xpath_json	data/library/album/#[artists/ artist/name==Anang Ashanty]/ songs/song/#	None
JsonQuery	count(library/album/*/[artists /artist/name=’Anang Ashanty’] /songs/song)	/