An Intermediate Representation-based Approach for Query Translation using a Syntax-Directed Method

Hassana NASSIRI[1], Mustapha MACHKOUR[2], Mohamed HACHIMI[3]
Laboratory of the Computing Systems and Vision, Laboratory of Engineering Sciences
University Ibn Zohr, Agadir, Morocco

***Abstract*—We aspire to make one query reasonably sufficient to extract data regardless of the data model used in our research. In such a way, users can freely use any query language they master to interrogate the heterogeneous database, not necessarily the query language associated with the model. Thus, overcoming the needing to deal with multiple query languages, which is, usually, an unwelcome matter for non-expert users and even for the expert ones. To do so, we proposed a new translation approach, relying on an intermediate query language to convert the user query into a suitable query language, according to the nature of data interrogated. Which is more beneficial rather than repeat the whole process for each new query submission. On the other hand, this empowers the system to be modular and divided into multiple, more flexible, and less complicated components. Therefore, it increases possibilities to make independent transformations and to switch between several query languages efficiently. By using our system, querying each data model with the corresponding query language is no longer bothersome. As a start, we are covering the eXtensible Markup Language (XML) and relational data models, whether native or hybrid. Users can retrieve data sources over these models using just one query, expressed with either the XML Path Language (XPath) or the Structured Query Language (SQL).**

Keywords—Data Model; Relational Database; eXtensible Markup Language (XML); translation; model integration; intermediate representation; ANTLR (ANother Tool for Language Recognition)

I. Introduction

The relational database has been the most data model used in most organizations to store and manage data. Likewise, the XML (eXtensible Markup Language) is progressively utilized as a universal solution to exchange data over the internet. At which point, many projects, and studies have been interested in integrating them and find means to interrogate both data. Some researchers focused on storing and querying XML data using a relational database system [1] [2]. Some others attempt to create general systems to manage XML, among other data formats [3]. However, approaches mentioned above have considerable advantages indeed but, along with limitations too, to some degree [4].

Nevertheless, by exploring some other orientations to query heterogeneous databases, especially those based on query translation, we perceived some related aspects to our intentions. Accordingly, adopting a translation tool can efficiently meet our aim, and using a syntax-directed approach would be a correct solution. To empower the process, we generate an intermediate query language that reflects the logical interpretation of the query. We called it the universal query language (UQL); a transitional phase that provides an intermediate representation to switch between steps accurately, instead of converting the source query language directly to the target query language. The system is capable of performing queries against XML and relational databases and against hybrid ones too.

Henceforth, there is no need to be familiar with the many query languages to access data from variant data models, nor to express queries with precisely the suitable query language that corresponds to the data model used to structure that data. One query represented with either the Structured Query Language (SQL) or The XML Path Language (XPath) is moderately enough [5].

We are relying on the syntax-directed translation method, in which the parser drives the source query language translation. Therefore, semantic analysis and interpretation are performed based on the syntax structure. For the hands-on part in building language processing tools, handwriting the parser may work, but it is obviously not the best approach in complex cases. Alternately, using a powerful parser generator can save us time, effort, and resources as it is capable of automating momentous phases along the process. For that, to implement the parser, we are using ANTLR (ANother Tool for Language Recognition). It takes as input a grammar that specifies a language and generates as output source code for a recognizer for that language. A language is specified using a Context-Free Grammar (CFG), expressed using Extended Backus–Naur Form (EBNF).

The paper is organized as follows: This introduction introduces the general context of the project. Section 2 brings in some preliminaries and terminologies. Section 3 presents our objects and summarizes the mechanisms of the overall system and the translation process. Section 4 explores the language recognition and processing phase. Section 5 presents the intermediate representation phase. Section 6 discusses the data extraction phase and the nature of the database understudy that can be handled by the system. Finally, Section 7 concludes.

II. Preliminaries

A. *Describing a Language using a Grammar*

A regular expression is quite useful but also leave little to no room for extension. Not all patterns can be described using regular expressions. The most obvious limitation is the lack of recursion. Statements can quickly turn out messy and hard to maintain [6] [7]. Thus, regular expressions are not quite enough. Instead, CFGs, the type-2 grammar in Formal Grammar Hierarchy classification as known as Chomsky Hierarchy [8], would be a great deal to define the syntax of a language.

Formally, A CFG [9] is a 4-tuple (N; ∑; S; P) where:

- N is a finite set of variables called nonterminals;
- ∑ is a finite set of terminals;
- S: An axiom is it the start nonterminal
- P is a finite set of productions (rewrite rules).

Each production has the form $N \rightarrow (N \cup \sum) *$

The head consists of a single nonterminal, and the body is a sequence of terminals and nonterminals.

We use the CFG to replace nonterminals by a string of nonterminals and terminals. The language of grammar is the set of strings it generates. A grammar could tell us the valid options to put together a piece of code for a given language and help us recognizing and identifying typical portions structures quickly.

B. *Grammar Notation*

There are many ways to describe a grammar, but we are using EBNF [10]. It is an extended version of the BNF (Backus-Naur form), an unambiguous, formal and mathematical way to specify CFGs. It is more concise and widely used as a formalism to describe a formal language grammar with a precise structure. It can be considered a metalanguage as it is a powerful way to define other languages. An EBNF grammar of a language consists of a set of terminal symbols and a set of productions for nonterminals, which shows the way terminal symbols are combined into a proper sequence.

C. *Another Tool for Language Recognition (ANTLR)*

It is possible to handwrite a parser from scratch, but this process can be complex, error-prone, and hard to change. Instead, there are many parser generators like Bison and Yacc [11] that take a grammar expressed in a domain-specific way, and generate code to parse that language. We are using ANTLR [12] [13] [14], It is a parser generator that uses LL(*) parsing [15] [16]. It takes a grammar as input and generates parsers that can build and walk parse trees and generate abstract syntax trees that can be further processed with tree parsers. From antlr.org, ANTLR is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. Also, ANTLRWorks [17] is a great ANTLR grammar development environment.

ANTLR is used by several popular frameworks, products, and projects, like:

- Apple, Oracle, NetBeans IDE, Eclipse projects (e.g., XText).
- Hive and Pig Languages use it to parse Hadoop queries,
- Twitter uses it to parse queries.
- Hibernate, Drools, JBoss, Groovy, Jython.

III. Aims and Mechanisms

A database is a set of information stored by a tool according to a data model, a defined structure. To extract and manipulate this information, we need a query language. Sources can be stored according to any model; this means that it can be heterogeneous. Now, let us assume that we have faced one of these scenarios: (1) A user who has some data sources in XML, and knows only SQL. (2) A user who masters XPath and wants to access relational data. The common point between these two use cases is that the user query language does not match the interrogated data model. It is not easy to retrieve data because the appropriate query language is needed, which is XPath, for example, for the first case, and SQL for the second one. Besides, most of the time, users cannot master all of these query languages all at once. Each query language has a particular specification and probably challenging to learn. It is where our proposed system comes in. To overcome the dependencies between the data model and the query language, we develop a system to extract data regardless of the nature of the model used (XML or relational). Using only one single query posed freely with any query languages (SQL or XPath), as explained in Fig. 1.

Fig. 2 depicts the principle of the translation proposed in our approach based on an intermediate representation in place of a direct translation, which is beneficial for other interpretations and independent transformations. It strengthens the system to be more independent and modular.

As shown in Fig. 3, it all begins from the user, who is free to choose between two different query languages, SQL, or XPath, to express the query and submit it to the reader. The latter provides a uniform interface between users and the system, and read their queries. At the outset, we are dealing with characters, but we aspire to get an abstract syntax tree that enables us to perform other actions for analysis. That is where the language recognizer phase (Section 4) takes action. It consists of two parsers, one to parse SQL queries and the other to parse XPath queries. For that, we developed a lexer grammar and a parser grammar for each query language. At the end of this stage, the output is a parse tree that will be fed to the Analyzer, where the abstract syntax tree is built and processed. Then, selecting only the relevant information to develop our universal query language in the UQL Builder phase using the mapping rules module to map each part of the query with a suitable part in the UQL. After that comes the role of the translator in translating it into the target language. Then, the converted query is executed in the data extraction phase.

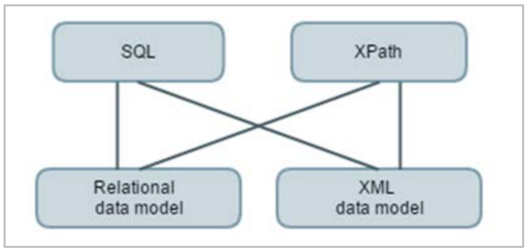

Fig. 1. One Query to Retrieve Data from XML and/or Relational Data Models [18].

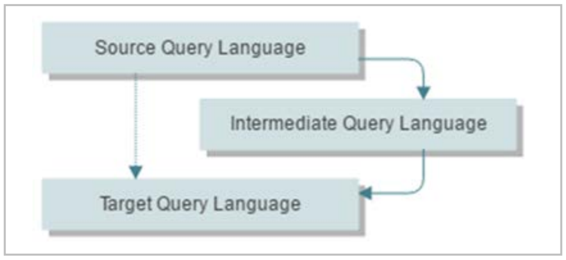

Fig. 2. The Principal of the Translation [19].

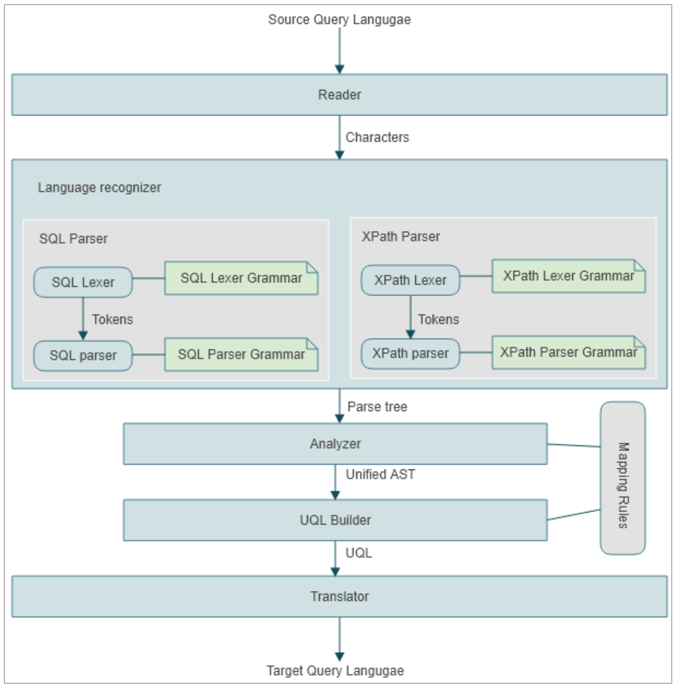

Fig. 3. The Query Translation Process.

IV. Language Recognition and Processing

The previous section presents the overview of the translation principle briefly, and the phases pursued to convert the source query into the target query. This section describes the several steps in the language recognition and processing phase.

ANTLR admit three variants of grammar specifications: lexers, parsers, and tree walkers or tree-parsers, as shown in Fig. 4. All of them are alike, and the generated files behave the same way because ANTLR uses LL(k) analysis for all of them.

The lexer reads the input character by character and translates it into a sequence of syntactical units called Tokens. Then, fed to the parser, which takes a stream of tokens and produces a parse tree according to the grammar rules. Afterward, the tree walker process the Parse Tree produced.

A. *Query Language Specifications*

The first step is the grammar. Because we are covering XML and relational database models, we need to define the grammar of their query languages, namely XPath and SQL.

SQL is a powerful query language for managing and manipulating data and can fit almost every interaction aspect. However, as the objective herein is interrogating data, we are focusing specifically on the select command, whose syntax is as follows in Fig. 5. Similarly, we are focusing on the most critical construct of XPath, the location path. Fig. 6 illustrates its EBNF notation.

B. *Diagrammatic Form*

ANTLR uses a simple EBNF-like syntax to define the grammar. For example, a column's syntax in a select clause can be written, as shown in Fig. 7, and presented in Fig. 8 using the railroad diagram.

Lexer rules start with an uppercase letter, and parser rules start with a lowercase letter.

Each rule has one or more patterns that it matches.

The K_AS? Means matches zero or one occurrence of K_AS.

'|' mean alternative patterns for the rule.

C. *Parsing Queries*

Parsing has the following phases: lexical analysis, syntactic analysis, semantic analysis. In the lexical Analysis (Tokenization), the lexer split up the user query into tokens and defines precisely how these tokens can be recognized. It reads a character stream as an input and generates a token stream as an output. Some tokens can be discarded like whitespaces; they are ignored during parsing. In the syntactic analysis, the parser figures out the relationship between the tokens that the lexer has produced to generate a parse tree, a data structure that reflects the input query's syntactic structure. In the Semantic analysis, the parse tree is checked for invalid semantic.

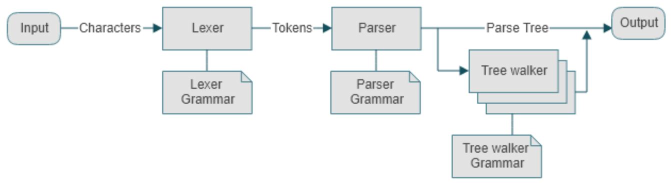

Fig. 4. Parsing Workflow.

select_clause ::=
SELECT (ALL | DISTINCT)? (<star> | (<select sublist> (<comma> <select sublist>)*))
from_clause ::=
FROM (<table reference> (<comma> <table reference>)*)
where_clause ::=
WHERE <condition>
group by clause ::=
GROUP BY (ROLLUP <lparen> <expression list> <rparen> | <expression list>)
having clause ::=
HAVING <condition>
order by clause ::=
ORDER BY <sort specification> (<comma> <sort specification>)*

Fig. 5. EBNF for SQL Select Grammar [20].

```
LocationPath ::=
            RelativeLocationPath
            | AbsoluteLocationPath
AbsoluteLocationPath ::=
            '/' RelativeLocationPath?
            | AbbreviatedAbsoluteLocationPath
RelativeLocationPath::=
            Step
            | RelativeLocationPath '/' Step
            | AbbreviatedRelativeLocationPath
Step ::=
            AxisSpecifier NodeTest Predicate*
            | AbbreviatedStep
AxisSpecifier ::= AxisName '::'
            | AbbreviatedAxisSpecifier
AxisName ::=
            ‘ancestor’
            | ‘ancestor-or-self'
            | ‘attribute’
            | ‘child’
            | ‘descendant’
            | ‘descendant-or-self'
            | ‘following’
            | ‘following-sibling’
            | ‘namespace’
            | ‘parent’
            | ‘preceding’
            | "preceding-sibling’
            | ‘self’
NodeTest ::=
            NameTest
            | NodeType ‘(' ‘)'
            | "processing-instruction' ‘(' Literal ')'
Predicate ::=
            ‘[' PredicateExpr ‘]'
PredicateExpr ::=
            Expr
AbbreviatedAbsoluteLocationPath ::=
            '//' RelativeLocationPath
AbbreviatedRelativeLocationPath ::=
            RelativeLocationPath '//' Step
AbbreviatedStep     ::=
            ‘.’
            | ‘..’
AbbreviatedAxisSpecifier        ::=
            '@’?
```

Fig. 6. EBNF for XPath Location Path [21].

The next section explores the process of generating the intermediate query language after parsing the source query, which is an in-between phase that helps generates the target query quickly. Section 6 provides further details regarding how the extraction works.

```
column
    : '*'
    | table_name '.' '*'
    | expression ( K_AS? column_alias )?
    ;
```

Fig. 7. ANTLR EBNF Syntax for a column

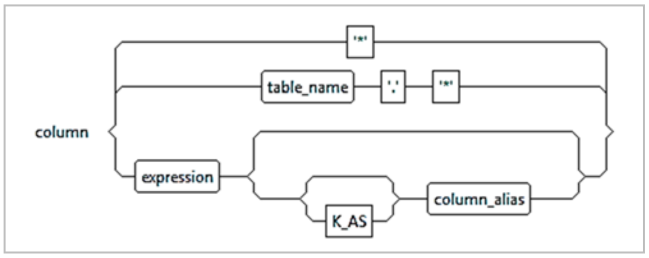

Fig. 8. Syntax Diagram for Column.

V. Intermediate Representation

The process of building the UQL starts from the output of the previous phase: the language recognizer. Then more steps have to be proceeded to get brief details that are needed, short, and to the point to efficiently generate the UQL. Besides, the parser generator builds a Concrete Syntax Tree (CST), not an Abstract Syntax Tree (AST). The CST reflects exactly the form of the grammar, every detail described in the syntax. It is like another representation of the grammar. That may seem easy to create but difficult to analyze and performed further interpretation with it. Whereas the AST contains only the mandatory elements needed and discard irrelevant details and extra information. It is more clear, compact, and easy to process than a parse tree. It is almost a direct translation of the grammar. We can get the abstract syntax from concrete syntax [22] [23]. After extracting the AST, all we need is to unify the ASTs and finally use the mapping rules to map every part of the unified AST and easily generates the UQL.

A. Parse Tree

A parse tree or derivation tree is a data structure that matches the syntactic structure of the input. For example, the SQL select query: “*select first_name, last_name from Employee where id = 1;*” has the following parse tree (Fig.9), presented in tree form in Fig. 10.

B. Abstract Syntax Tree

An AST is a variant of parse tree where we eliminate extra information and discard irrelevant details. Fig. 11 shows the AST for the query.

This example illustrates the output from a console, but we developed an interface ([5]) to present the analysis better.

```
select first_name, last_name from Employee where id = 1;
-- Reading data...
Your Query Language is: SQL
Columns: first_name last_name
Tables: Employee
Conditions : id=1
```

```
(select_statement (select_core (selectClause select (list_columns (column
(expression (column_name (any_name first_name)))) , (column (expression
(column_name (any_name last_name)))))) (fromClause from (list_tables
(table_or_subquery (table_name (any_name Employee))))) (whereClause
where (list_conditions (expression (expression (column_name (any_name
id))) (comp_operator =) (expression (literal_value 1))))) ;))
```

Fig. 9. Parse Tree for "Select First_Name, Last_Name from Employee where id = 1;".

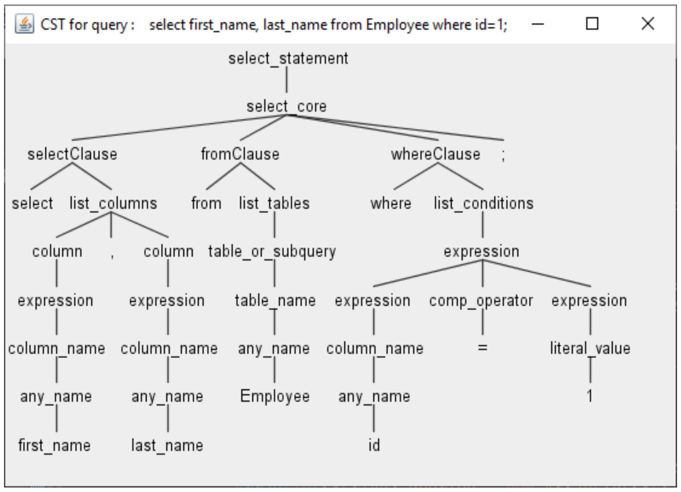
CST for query : select first_name, last_name from Employee where id=1;

Fig. 10. Parse Tree in Tree Form.

```
'- select_statement
  |- selectClause
  | |- TOKEN[type: 63, text: select]
  | '- columns
  |    |- anColumn
  |    | '- TOKEN[type: 67, text: first_name]
  |    |- TOKEN[type: 2, text: ,]
  |    '- anColumn
  |       '- TOKEN[type: 67, text: last_name]
  |- fromClause
  | |- TOKEN[type: 41, text: from]
  | '- tables
  |    '- TOKEN[type: 67, text: Employee]
  |- whereClause
  | |- TOKEN[type: 66, text: where]
  | '- conditions
  |    |- condition
  |    | '- TOKEN[type: 67, text: id]
  |    |- comp_operator
  |    | '- TOKEN[type: 6, text: =]
  |    '- lit_value
  |       '- TOKEN[type: 68, text: 1]
  '- TOKEN[type: 1, text: ;]
```

Fig. 11. Abstract Syntax Tree.

We get the following query after applying the unification principle illustrated in Fig. 12, along with the mapping rules to generate the UQL. We will take the same example from [5].

```
<?xml version="1.0" encoding="UTF-8"?>
<UQLroot>
  <Object>
    <ObjectName>emps</ObjectName>
    <Properties>
      <Property>nom</Property>
    </Properties>
  </Object>
</UQLroot>
```

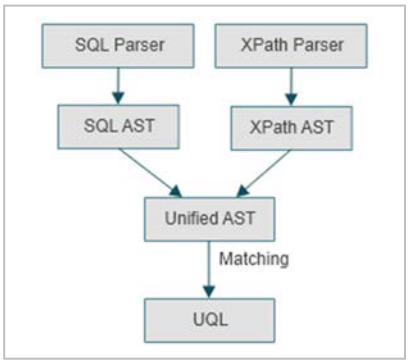

Fig. 12. The unified Abstract Syntax Tree.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="UQLroot">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Object"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Object">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ObjectName"/>
        <xs:element ref="Properties"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ObjectName" type=" xs:string "/>
  <xs:element name="Properties">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Property"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Property" type=" xs:string "/>
</xs:schema>
```

Fig. 13. XML Schema.

Now just one more step to complete this phase is the XML document validation. A well-formed XML document is one that conforms to the syntactic rules of the XML language. When an XML document has an associated DTD (Document Type Definition) or XSD (XML Schema Definition) and respects it, it is said to be valid. Validation is a way to verify that the document conforms to a grammar. We use the XML Schema depicted in Fig. 13 to describe the structure of our XML document.

VI. Data Extraction

The system can access data from heterogeneous data models, namely relational and XML, since the relational database has been the popular option to store and manage data since 1970 [24]. It is still the most data model used in most organizations and powerful database systems [25]. Likewise, XML is widely used as a standard to exchange data over the internet, and the Native XML Database tends to be a practical solution for variable data [26] and provides full support for XML query languages such as XPath or XQuery [27]. The system can also access data from a hybrid database, as major relational database management systems (DBMS) are appealing for hybrid engines so that they fit XML into a relational database environment [3], for instance, Oracle [28] [29] [30], IBM [31] and Microsoft. Furthermore, the extension to SQL for XML, SQL/XML, is making good advancements [32] [33].

AS shown in Fig. 14, after executing the query against the suitable database, we perform other transformations to determine what is to be done with the data, and how to go about doing it. Lastly, format the answer according to the user preference, if the choice is indicated. If it is not the case, we apply the obvious choice, a tree form for XML sources, and tabular form for relational and hybrid databases. The tabular layout is the selected format by default.

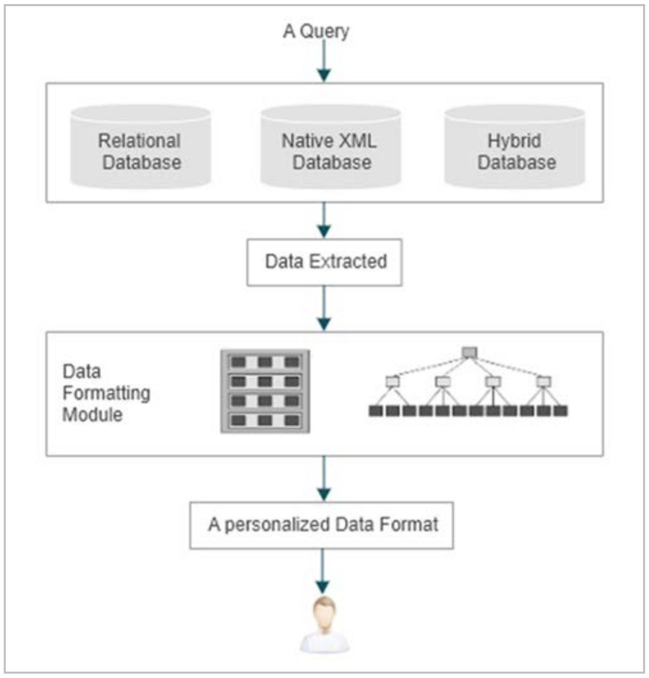

Fig. 14. Data Extraction Phase.

VII. Conclusion and Outlook

This paper presents our intermediate representation-based approach for translating queries, relying on the syntax-directed translation technique, to access data from heterogeneous sources and get the most out of each technology. The intermediate transition aids in empowering the system feature, especially in terms of independence. So that matching the data model with the query language corresponding remains no longer bothersome or a burden. Herein, we covered XML and relational data models, whether native or hybrid and hoping to incorporate others in future contributions.

References

[1] Florescu and D. Kossmann, “Storing and Querying XML Data using an RDMBS,” IEEE Data Eng. Bull., vol. 3, pp. 27–34, 1999, [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Storing+and+Querying+XML+Data+using+an+RDMBS#0.

[2] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, “Storing and querying ordered XML using a relational database system,” 2002 ACM SIGMOD Int. Conf. Manag. Data, SIGMOD’02, no. October, pp. 204–215, 2002, doi: 10.1145/564691.564715.

[3] M. Rys, D. Chamberlin, and D. Florescu, “XML and Relational Database Management Systems : the Inside Story,” SIGMOD ’05 Proc. 2005 ACM SIGMOD Int. Conf. Manag. data, pp. 945–947, 2005.

[4] J. Shanmugasundaram et al., “Relational databases for querying XML documents: Limitations and opportunities.,” Proc. 25th VLDB Conf., pp. 302-314. Morgan Kaufmann Publishers Inc., 1999, doi: 10.1016/j.acalib.2005.12.008.

[5] H. Nassiri, M. Machkour, and M. Hachimi, “One query to retrieve XML and Relational Data,” in Procedia Computer Science, 2018, vol. 134, pp. 340–345, doi: 10.1016/j.procs.2018.07.201.

[6] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant, “Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions,” pp. 415–426, 2020, doi: 10.1109/ase.2019.00047.

[7] J. C. Davis, L. G. Michael, C. A. Coghlan, F. Servant, and D. Lee, “Why arent regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions,” ESEC/FSE 2019 - Proc. 2019 27th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., pp. 443–454, 2019, doi: 10.1145/3338906.3338909.

[8] N. Chomsky, “Three models for the description of language,” IRE Trans. Inf. Theory, vol. 2, no. 3, pp. 113–124, 1956, doi: 10.1109/TIT.1956.1056813.

[9] A. Cremers and S. Ginsburg, “Context-free grammar forms,” J. Comput. Syst. Sci., vol. 11, no. 1, pp. 86–117, 1975, doi: 10.1016/S0022-0000(75)80051-1.

[10] ISO/IEC 14977, “Information technology - Syntactic metalanguage - Extended BNF,” Int. Stand. 149771996(E), vol. 1996, pp. 1–24, 1996, doi: 10.1002/(SICI)1099-1670(199603)2:1<35::AID-SPIP29>3.0.CO;2-3.

[11] S. C. Johnson, “Yacc : Yet Another Compiler-Compiler,” Comput. Sci. Tech. Rep. No. 32, p. 33, 1975.

[12] T. Parr, “The Definitive ANTLR 4 Reference,” Anim. Behav., vol. 67, no. 4, pp. 627–636, Apr. 2004, doi: 10.1016/j.anbehav.2003.06.004.

[13] T. Parr, The Definitive ANTLR Reference - Building Domain Specific Languages. 2007.

[14] T. Parr and K. Fisher, “LL(*): The foundation of the ANTLR parser generator,” Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement., pp. 425–436, 2011, doi: 10.1145/1993498.1993548.

[15] T. J. Parr and R. W. Quong, “ANTLR : A Predicated- LL (k) Parser Generator,” pp. 1–21, 1994.

[16] T. Parr, S. Harwell, and K. Fisher, “Adaptive LL(*) parsing,” ACM SIGPLAN Not., vol. 49, no. 10, pp. 579–598, 2014, doi: 10.1145/2714064.2660202.

[17] Y. H. Wang and I. C. Wu, “ANTLRWorks: an ANTLR grammar development environment,” Softw. - Pract. Exp., vol. 39, no. 7, pp. 701–736, 2009, doi: 10.1002/spe.

[18] H. Nassiri, M. Machkour, and M. Hachimi, “Integrating XML and Relational Data,” in Procedia Computer Science, 2017, vol. 110, pp. 422–427, doi: 10.1016/j.procs.2017.06.107.

[19] H. Nassiri, M. Machkour, and M. Hachimi, “Querying XML and Relational Data,” Int. J. New Comput. Archit. their Appl., vol. 7, no. 2, pp. 50–55, 2017, doi: http://dx.doi.org/10.17781/P002328.

[20] “BNF for SQL Grammar.” https://docs.jboss.org (accessed Aug. 27, 2020).

[21] “xpath.” https://www.w3.org/TR/xpath/ (accessed Aug. 27, 2020).

[22] D. S. Wile, “Abstract syntax from concrete syntax,” pp. 472–480, 1997, doi: 10.1145/253228.253388.

[23] I. Ráth, A. Ökrös, and D. Varró, “Synchronization of abstract and concrete syntax in domain-specific modeling languages: By mapping models and live transformations,” Softw. Syst. Model., vol. 9, no. 4, pp. 453–471, 2010, doi: 10.1007/s10270-009-0122-7.

[24] E. F. Codd and S. Jose, “A Relational Model of Data for Large Shared Data Banks,” vol. 13, no. 6, 1970.

[25] Y. Bassil, “A Comparative Study on the Performance of the Top DBMS Systems,” arXiv Prepr. arXiv1205.2889, pp. 20–31, 2012, [Online]. Available: http://arxiv.org/abs/1205.2889.

[26] G. Pavlovic-Lazetic, “Native XML databases vs. relational databases in dealing with XML documents,” Kragujev. J. Math., vol. 30, pp. 181–199, 2007, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.7634&amp;rep=rep1&amp;type=pdf.

[27] W. C. W. D. March, XQuery 1 . 0 Formal Semantics, no. March. 2002.

[28] R. Murthy and S. Banerjee, “Xml schemas in Oracle XML DB,” Proc. 29th Int. Conf. Very large databases, vol. 29, pp. 1009–1018, 2003, doi: 10.1016/B978-012722442-8/50094-X.

[29] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, R. Murthy, and O. Corporation, “Oracle8 i - The XML Enabled Data Management System Oracle Corporation for XML,” vol. 2, no. 100, 2000.

[30] M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. W. Warner, V. Arora, and S. Kotsovolos, “Query Rewrite for XML in Oracle XML DB,” Data Base, 2004.

[31] F. Ozcan, D. Chamberlin, K. Kulkarni, and J. E. Michels, “Integration of SQL and XQuery in IBM DB2,” Ibm Syst. J., vol. 45, no. 2, pp. 245–270, 2006, doi: 10.1147/sj.452.0245.

[32] A. Eisenberg, J. Melton, and O. Corp, “Advancements in SQL/XML,” SIGMOD Rec., vol. 33, no. 3, pp. 79–86, 2004, doi: 10.1145/1031570.1031588.

[33] A. Eisenberg and J. Melton, “SQL/XML is making good progress,” ACM SIGMOD Rec., vol. 31, no. 2, p. 101, 2002, doi: 10.1145/565117.565141.