Computer Science • 17 (3) 2016 http://dx.doi.org/10.7494/csci.2016.17.3.353

Adam Furmanek
Jakub Tokaj
Robert Marcjan
Leszek Siwik

REALIZATION OF A SYSTEM OF EFFICIENT QUERYING OF HIERARCHICAL DATA TRANSFORMED INTO A QUASI-RELATIONAL MODEL

Abstract *Extensible Markup Language was mainly designed to easily represent documents; however, it has evolved and is now widely used for the representation of arbitrary data structures. There are many Application Programming Interfaces (APIs) to aid software developers with processing XML data. There are also many languages for querying and transforming XML, such as XPath or XQuery, which are widely used in this field. However, because of the great flexibility of XML documents, there are no unified data storing and processing standards, tools, or systems.*

On the other hand, a relational model is still the most-commonly and widely used standard for storing and querying data. Many Database Management Systems consist of components for loading and transforming hierarchical data. DB2 pureXML or Oracle SQLX are some of the most-recognized examples. Unfortunately, all of them require knowledge of additional tools, standards, and languages dedicated to accessing hierarchical data (for example, XPath or XQuery). Transforming XML documents into a (quasi)relational model and then querying (transformed) documents with SQL or SQL–like queries would significantly simplify the development of data-oriented systems and applications.

In this paper, an implementation of the SQLxD query system is proposed. The XML documents are converted into a quasi-relational model (preserving their hierarchical structure), and the SQL–like language based on SQL-92 allows for efficient data querying.

Keywords XML, SQL, hierarchical data, relational model

Citation Computer Science 17 (3) 2016: 353–369

1. Motivation

The relational data model is widely considered as a standard for storing and manipulating data. It results from both the maturity of the tools and database systems as well as its ease of describing the semi-structured data. The relational data model is great for storing semi–structured data, since the properties of the entities can be easily mapped into columns. However, the relational data model has some major drawbacks when it comes to the manipulation of hierarchical data – representing deeply nested hierarchies as tables connected with foreign keys is not only cumbersome but also a great challenge for the database management system, which needs to execute complicated queries with many (self)joins in a short time.

There are situations when the XML data model is much more handy than the relational data model – representation of the hierarchical data is one such situation. XML can describe heterogeneous data (in contrast to the relational data model), and it also provides a great support for storing markup data or data with the evolving schema. In general, the XML data model is much more powerful and (due to its flexibility) able to store many more types of data structures.

Approaching the problem of storing and querying XML data, it would be desirable to take advantage of the flexibility of the hierarchical data model and, at the same time, use the strength of the relational data model and advanced database management systems.

There are three possible approaches. The first one is storing the XML documents inside the relational database just as textual data. Unfortunately, this approach is against both the idea of a relational data model and against the hierarchy of hierarchical data.

The second is equipping RDBMSs with some extensions (types, languages, tools, operators, etc.) for loading, storing, and querying XML data. DB2 pureXML [9] or Oracle SQLX [12] are some of the most-recognized examples. Unfortunately, all of them require knowledge of additional tools, standards, and languages dedicated for accessing hierarchical data (for example, XPath [13] or XQuery [14]).

The third possible approach is transforming XML documents into a (quasi)relational model and then querying the (transformed) documents with SQL or SQL–like queries. Such an approach has been proposed and discussed; for instance, in [1, 2, 4]. Such an approach (i.e., transforming XML documents into [quasi]relational) seems to be the most appropriate, since it is possible to utilize both the flexibility of the XML data format and the tools and languages for storing, manipulating, and querying relational data.

Utilizing the relational-oriented tools, models, and languages for manipulating hierarchical data is so important, since the relational data model is much more common and many tools are designed to work with it. It can be processed more efficiently, not only because of the great experience of the developers creating the database management systems and relational data models, but also because the creation of the

constraints and dependencies is simpler in a relational data model. It can handle redundancy in an elegant way thanks to normalization.

What is also important–or even crucial here–the Structured Query Language is one of the most-common and well-known languages for querying data. It is considered to be presumed knowledge of every software designer and developer. That is why the relational data model is still an attractive way for storing data, and also using this model for storing XML-documents would be greatly simplified when hierarchical data has to be stored and queried.

In this paper, the implementation of the SQLxD query language for manipulating hierarchical data is presented and discussed. The approach to transformation described in [5] and [6] was used as a basis for the implementation. The XML-2-Relational transformation described in [5] transforms the hierarchical XML document into its relational representation and allows us to create the necessary SQL tables storing the data. After transformation, the user is able to use almost-pure SQL-92 DQL queries to process the data.

2. XML-2-Relational transformation

The general idea of the transformation is to choose the specific nodes from XML documents using a simple addressing and then transform them into "rows": one node (together with nested nodes) is mapped to one row. The attributes of the node, children text values, and children elements are used as a base to construct "columns" of the "row." The final relation is the sum of all of the transformed "rows." This idea is shown in Figure 1. It depicts the execution of "SELECT * FROM document.people.person AS p" SQLxD query. All of the "person" nodes nested within the "document" and "people" nodes from the documents are chosen and then transformed into "rows," creating the relation. Because one node may have many subnodes of the same name, the transformation flattens them all into one set of values, selecting them based on their order of appearance.

Detailed information about the transformation, along with a formal specification, can be found in [5].

3. Design of the system

The presented SQLxD query system has been implemented in C# 5, with some modules written in IronPython [3]. This consists of four main components:

- Parser – this is responsible for parsing XML documents and transforming them to the structure, allowing further transformation described in [5]. It not only creates the casual node representation of XML tags but also recognizes them and classifies.
- Model – this module contains classes representing the XML documents and SQL tables.

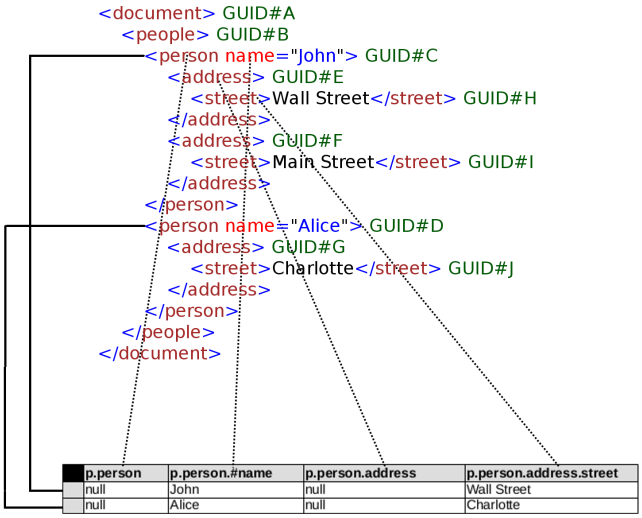

Figure 1. XML-2-Relational transformation.

- Query Parser – this is responsible for parsing the DQL query from its textual representation into the abstract syntax tree (AST). The AST is used later in the Query Logic module.
- Query Logic – this is the main module of the database system, its core component. It contains the logic executing the DQL queries. The operations supported are described later in this paper.

The general flow of processing XML documents and SQLxD queries in the system is shown in Figure 2.

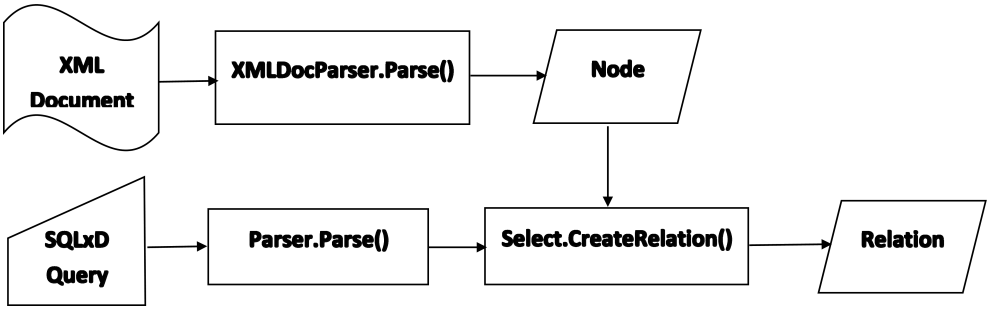

Figure 2. General flow of processing XML documents and SQLxD queries.

The Parser module implements the algorithm described in [5] and [6]. This time, we decided not to recognize the meaning of nodes with the same name – for instance, we do not try to recognize the difference between the Address tag representing the home address and the Address tag (so, a tag with the same name) representing the e-

mail address. We assume that the end user is responsible for the correct interpretation of the data. We treat the missing pieces of information (for instance, a missing city in the e-mail Address tag) as NULL values.

The Model module is simply a container for the data. It consists of the classes representing the XML documents (which are Node and NodeType) and the classes used to represent the relational tables. The latter group of classes consist of the following elements:

- Cell, GuidCell – these two classes represent the single atomic value in the entity. The Cell is used to represent the value. It is able to distinguish the numerical values, date values, and boolean values from ordinary character strings. The GuidCell is used to perform the natural join operation, described later in this paper. Both of these classes also have the ColumnHeader class instance, described below.
- ColumnHeader, Row – these two classes represents every single tuple. The ColumnHeader contains information about the schema and column name and is used later to extract the data, and the Row class is a container for Cells and GuidCells.
- Relation – this class represents the set of Row instances, which are relational-like entities created from the hierarchical data. We can treat the Relation class as an SQL table.

The Query Parser module is responsible for parsing the Data Query Language queries and transforming them from textual representation to the Abstract Syntax Tree. It is written in IronPython [3] and uses the PLY [8] library for creating the lexer and parser of the grammar. The grammar is based on SQL-92 grammar [11] and allows us to execute almost all of the DQL operations assumed by the standard, but it has some differences in the method of addressing the particular columns and naming the tables. We do not provide the full grammar description in this paper because it is not necessary to understand the form of the query supported by SQLxD, which is presented in Listing 1. There are two main differences between the SQLxD and SQL-92 languages, which we will now describe.

Listing 1. Structure of the DQL query supported by SQLxD.

```
SELECT
  *
  | list_of_columns_functions_and_aggregates
FROM selector AS selector_name
{
  [ NATURAL JOIN selector AS name ]
  [ INNER JOIN selector AS name ON condition ]
  [ LEFT [ OUTER ] JOIN selector AS name ON condition ]
  [ RIGHT [ OUTER ] JOIN selector AS name ON condition ]
  [ FULL [ OUTER ] JOIN selector AS name ON condition ]
  [ CROSS JOIN selector AS name ]
}
[ WHERE condition ]
```

```
14  [ GROUP BY list_of_columns ]
15  [
16    ORDER BY list_of_columns [ DESC ]
17    [ SKIP number_of_rows ]
18    [ FETCH number_of_rows ]
19  ]
```

3.1. Addressing the columns

The first difference is the way of addressing the specific column. In SQL, we can use one-part addressing (object name) or two-part addressing (schema name.object name). In the SQLxD language, we need to be able to address a node that is potentially deep-nested. In order to achieve this, SQLxD provides the many-parts addressing scheme, in which nested tags are separated with dots. For instance, to point to the Address tag in the document shown in Listing 2, we need to write `Document.People.Person.Address`. We can also use two wildcards: a question mark `?` or a star `*`. The former means one nested tag (so we can point to the `Address` tag using `Document.?.Person.Address`), and the latter means any place deeper in the hierarchy (so we can point to the `Address` tag using `Document.*.Address`, or even `*.Address`). What is more, we always need to give the alias for the source of selection in the SQLxD language, while this is optional in SQL-92 standard [10] (as long as it does not create ambiguity).

Listing 2. Sample XML document.

```
1  <document>
2   <people>
3    <person name="John">
4     <address>
5      <street>Wall Street</street>
6     </address>
7     <address>
8      <street>Main Street</street>
9     </address>
10   </person>
11   <person name="Alice">
12    <address>
13     <street>Charlotte</street>
14    </address>
15   </person>
16  </people>
17 </document>
```

3.2. Natural join

Another difference between the SQLxD language and SQL-92 is the implementation of a natural join. In SQL-92, the natural join is a type of equi-join where the predicates

do not need to be explicitly specified, as they can be inferred from the structure of the tables. In the case when no respective columns are found, the natural join behaves like a cross join. In the SQLxD language, the natural join is used to present a parent-child relationship between a document's nodes. For instance, in Listing 2, the `address` node is always a child of the `Person` node, so they can be joined using the natural join. To select all of the values of the `person` nodes and join the `address` nodes to the corresponding `person` nodes, we need to do a natural join on the `person` and `address` relations.

4. Implementation of the set operations

In this section, some implementation details are presented. In particular, the implementation of joining, filtering with the Where clause, grouping and aggregates are discussed.

4.1. Joins

The base clause is the From clause. It specifies the data source for the relation, chooses the XML nodes according to the mask, executes the XML-2-Relational transformation, and produces the relation. After executing the From clause, we operate only on quasi-relational data.

First – the basic type of join is the cross join. Its implementation is rather straightforward: we simply take rows from two relations and create the result row that has the Cell and GuidCell instances from both rows. The Cell instances represent the atomic values, and the GuidCell instances represent the origin of the row.

The inner join is implemented as a cross join with some Where clause(s). We simply transform the clause `A INNER JOIN B ON predicate` to the form `A CROSS JOIN B WHERE predicate`. In fact, there is no difference between the cross join and the inner join, so the inner join is implemented as a nested-loop join.

The outer joins are implemented as an inner join with the addition of rows from the master side that are matched in a joining process.

The natural join is implemented differently. We need to join the rows representing tags with a parent-child relationship, so we need to know the origin of each row. By origin, we understand the place in the document of the node on which the row is based.

The Natural Join algorithm works as follows:

1. Parser module assigns a GUID to each node of the XML document that is read for further processing.
2. FROM clause is executed. This triggers the XML-2-Relational transformation of the chosen nodes. During the transformation, rows are created from nodes. GUIDs from nodes starting at the root of the document to the transformed node are added to the resultant row.

3. NATURAL JOIN clause is executed. Rows are joined based on their GUIDs. If the row with a lower count of GUIDs contains all of the GUIDs from the row with a higher count of GUIDs, this means that these two rows are in a parent-child relationship and are joined.

   Let us analyze the execution of the following SQLxD query:

```
SELECT p.person.#name, a.address.street FROM document.people.person AS p
NATURAL JOIN document.people.person.address AS a
```

While transforming the `document.people.person` nodes presented in Listing 2, we assign a GUID for every single XML tag. If we assign GUID A to the `document` tag, GUID B to the `people` tag, GUID C to the `person` John tag, and GUID D to the `person` Alice tag, the first row (representing John) gets GUIDs $[A, B, C]$ while the second row (representing Alice) gets GUIDS $[A, B, D]$. Analogously, rows constructed from the address tags nested within the John person tag gets GUIDs $[A, B, C, E, H]$ and $[A, B, C, F, I]$, and the row constructed from the address tag nested within the Alice person tag gets GUIDs: $[A, B, D, G, J]$. When Natural Join is executed on the tables constructed from these tags, the John row is joined with address rows $[A, B, C, E, H]$ and $[A, B, C, F, I]$, as they contain all of the GUIDs from the John row $[A, B, C]$, and the address with $[A, B, D, G, J]$ GUIDs is joined to the Alice row: $[A, B, D]$. This is shown in Figure 3.

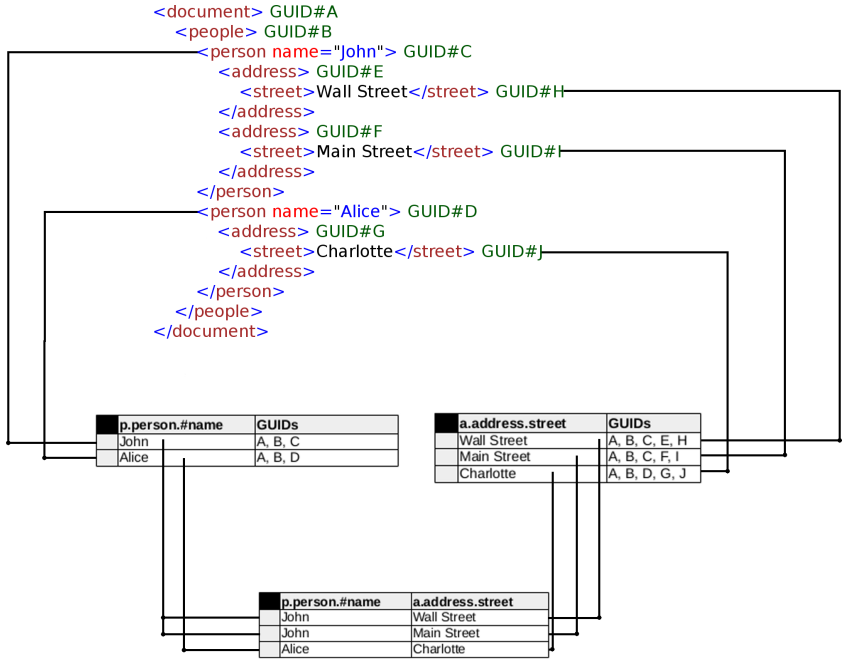

Figure 3. Natural Join example.

4.2. Filtering, grouping and ordering

The Where clause is implemented as a syntax tree of operators. SQLxD supports a full range of typical SQL operators – comparisons, boolean logic (AND, OR, NOT), IS NULL, and the LIKE operator (based on MS SQL implementation [7]). The grouping and ordering operations are based on GroupBy and OrderBy operators implemented in LINQ.

4.3. Aggregates and functions

SQLxD supports many typical aggregates, such as AVG, MIN, MAX, SUM, and COUNT. All aggregates are composed of three elements: outer expressions, inner expressions, and the aggregate logic.

The aggregate logic is responsible for calculating the actual result of the aggregate. This takes the group of rows of the relation and calculates the value. Because the aggregate does not need to operate on the raw values of the cells, we need to use an inner expression to transform the cell value to some different value; for instance, to calculate the length of a string. After calculating the aggregate, we may want to transform the value using a function, so we might need to use the outer expression. SQLxD also supports the composing of functions, so we are able to create a stack of functions.

5. Examples

In this section, two interesting and representative examples of SQLxD queries are presented. We use the sample document presented in Listing 3, presenting data from a hypothetical game. It includes heroes, countries, cities, things, artifacts, and potential targets. The actual semantic of this document is not important.

Listing 3. Sample document.

```
<game>
  <heroes>
    <hero name="Albert" id="1">
      <stats attack="25"/>
    </hero>
    <hero name="Bill" id="2">
      <stats attack="30"/>
    </hero>
    <hero name="Charlie" id="3">
      <stats attack="35"/>
    </hero>
  </heroes>
  <countries>
    <country name="Austria" id="1">
      <cities>
          <city name="Ansfelden" id="1"/>
      </cities>
    </country>
```

```
19      <country name="Belgium" id="2">
20        <cities>
21          <city name="Brussels" id="2"/>
22        </cities>
23      </country>
24      <country name="Chile" id="3"/>
25    </countries>
26    <things>
27      <thing name="sword" id="1"/>
28      <thing name="club" id="2"/>
29      <thing name="dagger" id="3"/>
30      <thing name="maul" id="4"/>
31    </things>
32    <artifacts>
33      <artifact name="king's␣sword" id="1"/>
34      <artifact name="gnoll's␣shield" id="2"/>
35    </artifacts>
36    <targets>
37      <target name="tower"/>
38      <target name="fortress"/>
39    </targets>
40  </game>
```

5.1. Query with joins

We start with a query presenting various joins. The query presented in Listing 4 joins heroes with their statistics (using a natural join), cities (using a left join), things (using a right join), artifacts (using a full join), and a next filters row.

The actual execution of this query is as follows:

1. The XML document is parsed. It is done using method Parse from a class XmlDocumentParser. As a result, we obtain an instance of a class Node, which represents the mail node of the XML document. Each child node is also parsed and represented as a nested Node instance.
2. Next, the query 4 is parsed. It is done by the Parse method of a class Parser. This method returns an instance of Select class (with optional list of parse errors). The Select class is the clue of the query – it has nested instances of all query elements (it is, in fact, AST of the whole query). It also contains a method CreateRelation returning a result of the query.
3. In order to execute the query, we first extract data using selector `FROM game.?.hero AS h`. There are five types of selectors: TopLevelSelector (selects nodes from the root of the document), NodeSelector (selects nodes from one level, matching them by name), LevelSelector (selects all nodes from one level), AnySelector (selects nodes from any level), and ChainedSelector (selects nodes using one selector and passes them to another selector). A sample selector stack is presented in Figure 5. Next, we transform the extracted nodes into a relational representation.

4. We perform an analogous process for data extracted using game.*.stats AS s selector.
5. We now perform NATURAL JOIN of the selected rows. In order to do this, we first calculate the CROSS JOIN of extracted rows (which is implemented as a two inner loops). Next, we filter rows, matching their GUIDs.
6. We have calculated NATURAL JOIN. Now, we extract rows using selector game.countries.country.cities.city AS c and transform them to a relational counterpart.
7. We now perform LEFT OUTER JOIN. In order to do this, we first calculate the CROSS JOIN of the results of NATURAL JOIN and rows selected in the previous step, and next filter them using the WHERE predicate. We also iterate over the rows to find ones with no match and add them with the NULL values.
8. Now, we want to calculate RIGHT OUTER JOIN. Firstly, we select rows using the selector. Then, we transform RIGHT OUTER JOIN to LEFT OUTER JOIN, and we perform it as in the previous step.
9. In the next step, we calculate FULL OUTER JOIN. Firstly, we calculate the LEFT OUTER JOIN of rows, and then we add rows from the right side of JOIN not matched by the left side.
10. Finally, we filter rows using the LIKE predicate, comparison, and IS NULL predicate. This is done by iterating over rows and verifying whether they match the predicate.
11. Next, we select specified columns and change their aliases.

The actual query flow is presented in Figure 4.

For each WHERE clause, we generate a stack of operators representing the actual condition. A sample of such a stack is presented in Figure 6.

Listing 4. Sample query with joins.

```
 1  SELECT
 2       h.hero.#name AS name
 3     , s.stats.#attack AS attack
 4     , c.city.#name AS cityName
 5     , t.thing.#name AS thingName
 6     , a.artifact.#name AS artifactName
 7  FROM game.?.hero AS h
 8  NATURAL JOIN game.*.stats AS s
 9  LEFT OUTER JOIN game.countries.country.cities.city AS c
10      ON c.id = h.id
11  RIGHT OUTER JOIN game.things.thing AS t
12      ON t.id = h.id
13  FULL OUTER JOIN game.?.artifact AS a
14      ON a.id = h.id
15  WHERE h.hero.#name LIKE 'A%' O h.hero.#name IS NULL
```

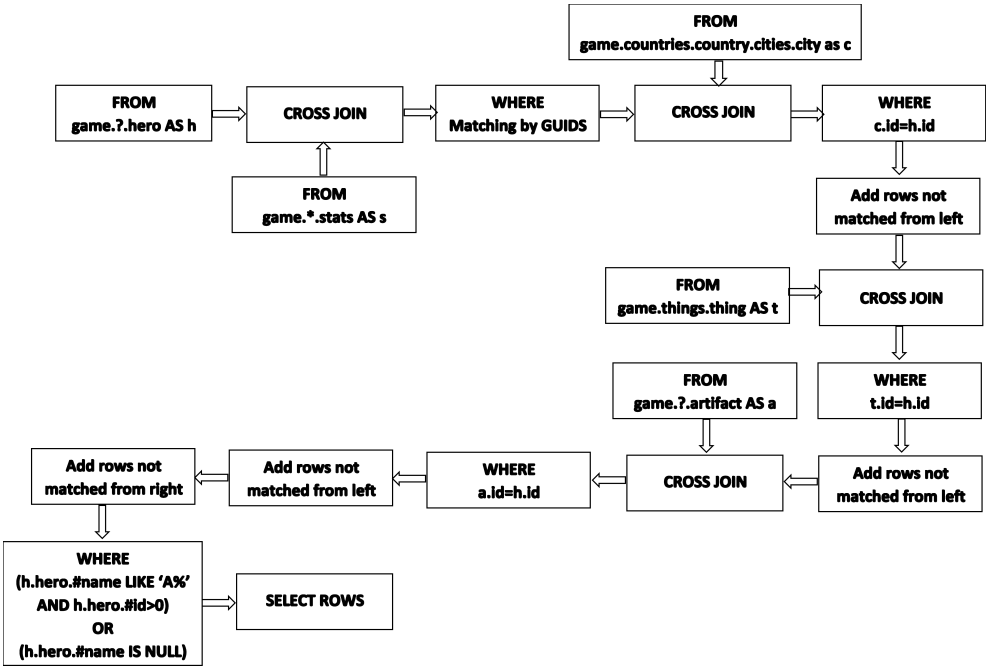

Figure 4. Flow of query with joins.

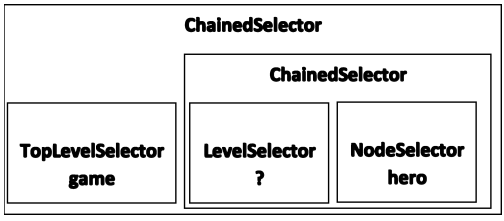

Figure 5. Sample selector stack used to extract nodes from game.?.hero.

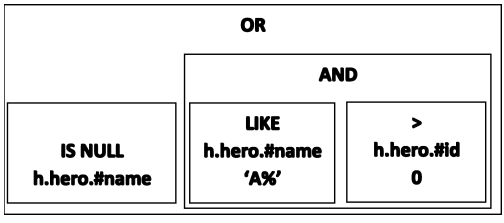

Figure 6. Sample condition stack used in WHERE clause.

5.2. Query with aggregates

Our library also supports common aggregates. The query presented in Listing 5 summarizes data using typical operators: COUNT, AVG, SUM, MAX, and MIN. We also use GROUP BY to group rows by name, ORDER BY to sort them by name in descending order, and select only one row (using OFFSET and FETCH syntax). We also use a sample method called SUBSTR, which calculates the substring of a string.

Each aggregate consists of three elements: outer ICellExpression, inner ICellExpression, and the actual aggregate logic. This construction is required because we might want to transform it using a chosen function (e.g., SUBSTR) before aggregating a value, which is done using inner ICellExpression. We might also want to transform the resulting aggregate value, which is performed by outer ICellExpression. So, the actual aggregate logic is as follows:

1. For each row, call inner ICellExpression.
2. For each obtained value, execute aggregate's logic (e.g., sum or calculate maximum).
3. For resulting value, call outer ICellExpression.
4. Return final result.

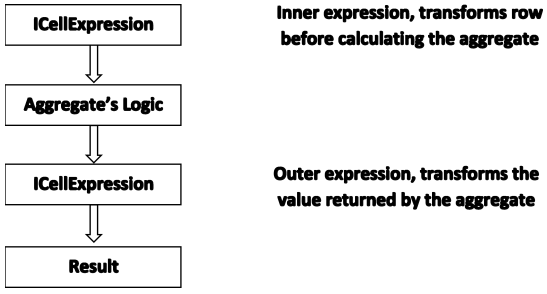

Figure 7. Aggregate's logic.

The aggregate's logic is presented in Figure 7. Because we might want to stack multiple functions, there is a special ICellExpression called ChainedCellExpression, which is used to transform a value using multiple functions (it is also possible to stack one ChainedCellExpression on another).

Listing 5. Sample query with aggregates.

```
1  SELECT
2        h.hero.#name as name,
3      , SUBSTR(h.hero.#name, 1, 1) as firstLetter
4      , COUNT(p.postac.#imie) AS namesCount
5      , AVG(s.stats.#attack) AS average
6      , SUM(s.stats.#attack) AS sum
7      , MAX(s.stats.#attack) AS max
8      , MIN(s.stats.#attakck) AS min
```

```
FROM game.heroes.hero AS h
NATURAL JOIN game.*.stats AS s
GROUP BY h.hero.#name
ORDER BY h.hero.#name DESC SKIP 1 FETCH 1
```

6. Results

As a result of our work, the .NET library containing the implementation of SQLxD was created using C# and IronPython. The library allows the end user to execute a query on an XML document using a simple interface and provides the necessary models for further processing of quasi-relational data. In fact, executing queries comes down to loading an XML document, providing query text as a string, and retrieving results. An example of the usage of the library in C# code is shown in Listing 6.

Listing 6. Sample query with aggregates.

```
string exampleQuery =
"SELECT␣*␣FROM␣document.customers.order␣as␣p";
string xmlDocument = File.ReadAllText("file.xml");
Node model = XmlDocumentParser.Parse(xmlDocument);
QueryParser.Parsing.Parser
queryParser = new QueryParser.Parsing.Parser();
Select select = queryParser.Parse(exampleQuery).Item1;
Relation result = select.CreateRelation(model);
```

The design of the library was described in section 3. The proposed method of transformation and query language can be verified on many levels (e.g., performance, ease of use, expressive power of the language). The easy-to-use and the expressive power of the language has been shown by the code examples in Listings 5, 4. The query performance issues and its optimization was not the aim of this paper. However, to better qualify the usefulness of the presented solution, some results showing the performance of the queries have been presented in Table 1. Tests have been made on a standard PC computer, Intel(R) Core(TM) i7-3630QM CPU, 16GB RAM, running a Windows10 operating system. Experiments have been made for three kinds of queries:

- simple query with condition;
- query with a simple JOIN;
- complex query with two JOINS, GROUP BY phrase, and aggregate functions

and three randomly generated exemplary data sets (XML files) representing customers/orders/order details hierarchy, with 100, 1000, and 10000 nodes. Table 1 shows the average execution times in seconds. The results are perhaps not perfect yet (especially for the largest data sets), but they are acceptable, at least in some practical-use cases. Obviously, the solution is not yet enterprise-ready, and the results cannot be compared with mature commercial implementations (e.g., those known from database management systems), but it should be mentioned that the proposed solution (at its current stage of implementation) does not use any optimization techniques like data caching or a query optimization module.

Table 1
Queries execution time (in seconds).

query	10 nodes	1000 nodes	10 000 nodes
SELECT c.id FROM clist.customer AS c WHERE c.id < 100	1.70 ± 0.01	1.81 ± 0.01	3.59 ± 0.01
SELECT c.id FROM clist.customer AS c NATURAL JOIN orders.order as o WHERE c.id < 100 AND o.#no > 300	1.72 ± 0.01	2.69 ± 0.01	24.50 ± 0.02
SELECT c.id, o.#no, SUM(p.id), AVG(p.id), COUNT(p.id) FROM clist.customer AS c NATURAL JOIN orders.order as o NATURAL JOIN details.product as p WHERE c.id < 100 AND o.#no > 300 and p.id > 20 GROUP BY c.id, o.#no	2.61 ± 0.01	2.90 ± 0.01	29.62 ± 0.02

7. Conclusions

The described implementation of the SQLxD query language supports most of the typical operations described in the SQL-92 standard. It can calculate the joins, filter and project the result, group it, and calculate aggregated values, so it is possible to perform most day-to-day data manipulations. It can be easily expanded to support more operations like CTE, windowing functions, or subqueries. Further work could also add the option to save the result of a DQL query as an XML document, which can be done using actual implementation (because of the GUIDs representing the source of each row).

The main goal of the transformation is the ability to represent hierarchical documents as relational tables so we can manipulate the data with the well-known SQL language. This goal has been achieved, and SQLxD can be used in many environments. Of course, this product is not yet enterprise-ready, but it shows that the concept of an XML-2-Relational transformation can join the two well-known worlds of data representation into one, allowing the user to store data in a readable format and with a flexible schema, and manipulate it using the common SQL language.

Acknowledgements

This research was partially supported by the Polish National Centre for Research and Development grant No. DOB-BIO6/08/129/2014 and by The Polish Ministry of Science and Higher Education under AGH University of Science and Technology Grant 11.11.230.124 (statutory project).

References

[1] Amer-Yahia S., Du F., Freire J.: A Comprehensive Solution to the XML-to-relational Mapping Problem. *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*, WIDM '04, pp. 31–38, ACM, New York, NY, USA, 2004, http://doi.acm.org/10.1145/1031453.1031461.

[2] Barbosa D., Freire J., Mendelzon A.O.: Designing Information-preserving Mapping Schemes for XML. *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 109–120, VLDB Endowment, 2005, http://dl.acm.org/citation.cfm?id=1083592.1083608.

[3] IronPython. http://ironpython.net, accessed: 2014-05-23.

[4] Khan L., Rao Y.: A Performance Evaluation of Storing XML Data in Relational Database Management Systems. *Proceedings of the 3rd International Workshop on Web Information and Data Management*, WIDM '01, pp. 31–38, ACM, New York, NY, USA, 2001, http://doi.acm.org/10.1145/502932.502939.

[5] Marcjan R., Siwik L.: The Concept of Transformation of XML Documents into Quasi-Relational Model. S. Kozielski, D. Mrozek, P. Kasprowski, B. Malysiak-Mrozek, D. Kostrzewa, eds., *Beyond Databases, Architectures, and Structures*, *Communications in Computer and Information Science*, vol. 424, pp. 569–580, Springer International Publishing, 2014, http://dx.doi.org/10.1007/978-3-319-06932-6_55.

[6] Marcjan R., Wyrostek J.: Processing XML documents on the basis of quasi-relational model and SQLxD language. *Studia Informatica*, vol. 32(2A), pp. 111–120, 2011.

[7] LIKE (Transact-SQL). http://msdn.microsoft.com/en-US/en-EN/library/ms179859(v=sql.105).aspx, accessed: 2014-05-23.

[8] PLY (Python Lex-Yacc). http://www.dabeaz.com/ply/, accessed: 2014-05-23.

[9] DB2 pureXML. http://www-01.ibm.com/software/data/db2/linux-unix-windows/xml/index.html/, accessed: 2014-05-23.

[10] SQL-92. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, accessed: 2014-05-23.

[11] BNF Grammar for ISO/IEC 9075:1992 - Database Language SQL (SQL-92). http://savage.net.au/SQL/sql-92.bnf.html, accessed: 2014-05-23.

[12] SQLX. http://docs.oracle.com/cd/B10501_01/appdev.920/a96616/arxml34.htm, accessed: 2014-05-23.

[13] XPath Reference. http://www.w3schools.com/XPath, accessed: 2014-05-23.

[14] XQuery Reference. http://www.w3schools.com/xQuery/default.asp/, accessed: 2014-05-23.

Affiliations

Adam Furmanek
AGH University of Science and Technology, Krakow, Poland, afurmanek@student.agh.edu.pl

Jakub Tokaj
AGH University of Science and Technology, Krakow, Poland, jtokaj@student.agh.edu.pl

Robert Marcjan
AGH University of Science and Technology, Krakow, Poland, marcjan@agh.edu.pl

Leszek Siwik
AGH University of Science and Technology, Krakow, Poland, siwik@agh.edu.pl

Received: 6.08.2015
Revised: 20.04.2016
Accepted: 26.04.2016