Timo Niemi and Turkka Näppilä

Conversion of XML-based Open Data into Relational Closed Data

UNIVERSITY OF TAMPERE
SCHOOL OF INFORMATION SCIENCES
REPORTS IN INFORMATION SCIENCES 33

TAMPERE 2014

UNIVERSITY OF TAMPERE
SCHOOL OF INFORMATION SCIENCES
REPORTS IN INFORMATION SCIENCES 33
DECEMBER 2014

Timo Niemi and Turkka Näppilä

Conversion of XML-based Open Data into Relational Closed Data

SCHOOL OF INFORMATION SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 978-951-44-9699-8

ISSN-L 1799-8158
ISSN 1799-8158

Conversion of XML-based Open Data into Relational Closed Data

Timo Niemi and Turkka Näppilä
University of Tampere, School of Information Sciences
Kanslerinrinne 1, 33014 University of Tampere, Finland
Email: firstname.lastname@uta.fi

Abstract

The lack of tools to support situational analytics based on data sources both in Open Data and in Closed Data is a serious problem especially in the enterprise environment. Typically, many Open Data sources are XML-based or easily convertible to XML, whereas majority of data sources in Closed Data and existing data analytics tools is based on the relational model. In order to facilitate this kind of analytics, we introduce an approach and tool which are able to convert an XML source to a part of a relational database. The developed conversion tool has a high degree of automation, which makes it possible to offer an easily useable interface for users. The conversion tool has also a great restructuring power, which enables in addition to data-to-data translations, data-to-metadata and metadata-to-data translations through its *pivot* and *unpivot* operations, respectively.

Keywords: Integration of Open Data and Closed Data; XML; relational model; conversion; formal specification.

1. Introduction

Any data produced by an organization that can only be used through a specific access control system are Closed Data for the persons who have no permission to use this system. Organizations belonging to the public or private sectors can also provide data on the Web freely without any restrictions for their reuse. This kind of publicly available information is called Open Data. It is typical of Open Data that its publishers provide data sets autonomously and independently of each other. This, in turn, means that Open Data is highly heterogeneous, supporting several different data formats, representations, and labels. On the contrary, Closed Data are represented and modeled in a consistent way; i.e., they are well integrated.

Due to the globalization, there is an increasing need to integrate different data sets in Open Data with each other, different Closed Data controlled by several organizations with each other and Open Data with Closed Data. The common challenge in these three integration needs is the capability to manage heterogeneity among available data. Already, applications based on pure Open Data would require that different data sets in Open Data would be better compatible with each other, in other words their degree of integration should be higher. For this purpose,

different tools have been developed. For example, the idea in the tool developed in [EDBT+13] is to avoid the use of several different labels to denote the same concept in a collection (corpus) of data sets in Open Data. This tool checks that the publisher of a new data set uses the acceptable vocabulary defined for the corpus before adding a new data set to the corpus of interest. Another approach to increase the integration degree among Open Data is to use the same data model for representing all data sets in the corpus at hand. In [VTBL-13], the RDF data model is proposed for representing structured, irregularly structured and unstructured data sources in the Linked Open Data cloud.

There are masses of information both in the public (e.g., certain health care information or national security data) and private (e.g., enterprise-sensitive data) sectors which must be kept as Closed Data. However, enterprises operating in the global environment have business partners with which co-operation requires mechanisms to share and integrate Closed Data between enterprises. In some extent, this has been realized in B2B (business-to-business) commerce where some standard XML vocabulary is used for exchanging and sharing data on the Web between enterprises belonging to a specific business domain. For example, in [LMGC+09] XML standards for several business domains are considered. In spite of this, heterogeneity of data structures and technologies has been recognized as one of the key reasons, which limit sharing of enterprise-sensitive information between business partners [RLBP-10]. This induces a need to develop novel tools for sharing and integrating Closed Data controlled by separate organizations.

During the last few years, increasing attention has been paid to how Open Data can be integrated with Closed Data. We address this issue in the paper. For example, in [RCHa-12] different possibilities to integrate XBRL (eXtensible Business Reporting Language)-based data sources with Open Data sources are considered. XBRL is a standardized XML-based mark-up language for representing financial data of enterprises. Of course, it is possible that in future enterprises will publish their XBRL-based data sources to a lager extent as Open Data. Until now, business analytics have mainly concentrated on utilizing only Closed Data internal to a specific enterprise. For example, popular OLAP (on-line analytical processing) tools (see e.g. [CDNa-11]) have been used to analyze multi-dimensionally data cubes whose contents have been collected from operational databases of enterprises, i.e., data cubes typically contain Closed Data.

However, several authors (see e.g. [VTBL-13, ETBL-13]) have recently recognized that there is lack of such business intelligence tools, which are able to support situational (or ad hoc) analytics based on both Closed and Open Data. By using data sources in Open Data together with data sources in Closed Data, business analytics can be enriched considerably. For example, it is possible to gain knowledge and insights from new product promotions, competing products/companies, market situations, consumer sentiment etc. based on data sources in Open Data.

The combination of Open Data and Closed Data for analytics is a complex process consisting of several different tasks as shown in [BETL-12]. One of the challenges is to find relevant data

sources from Open Data. DrillBeyond [ETBL-12] is a system, which allows the use of such attribute names in SQL queries, which do not appear in the underlying relational database. Through its IR facility, the system tries to find the relevant contents for these attributes. Although the user would know the data sets in Open Data, which are relevant to his/her information need, it is not clear that the user is able to use them. This is because the user is often unfamiliar with these data sources as opposed to data sources in Closed Data. The idea in the recent data management trend, called dataspace [FHMa-05, HFMa-06], is to increase interactively the knowledge of the user about the underlying data sources. In [NäNi-12, MNNK-14], we have introduced data profiling tools of our XML-based dataspace system in terms of which the user is able to find out the contents, structures and semantics related to data sources beforehand unfamiliar to him/her. In this paper, our starting point is that the user masters the contents, structures and semantics of relevant data sources in Open Data.

In spite of our starting point, there is still one major challenge to be considered. Namely, Open Data have to be integrated with Closed Data for situational analytics. Often the information needs based on situational analytics are casual and short-term in nature. Therefore, it is not meaningful to build a full-scale data integration system for this purpose because the building of this kind of systems is a time- and resource-consuming process [BPEF+10]. In this paper, we propose that the part of a data source in Open Data, which is relevant to the situational analytics case of interest, is extracted from this data source and materialized as a structure based on the data model used in Closed Data. We see that this approach has the following benefits. All data sources are compatible with each other and they are in the same repository controlled by one system used to manage Closed Data. In turn, this affords the possibility of utilizing the efficient storage and processing mechanisms of the data management system of Closed Data as such without any modification. Likewise, all analytics tools in the Closed Data environment are available. This kind of approach has been applied in [EWTB+13, ETBL-13] for extracting relational-style Open Data (e.g. HTML-tables) and converting them into the form processable in RDBMs. In this paper, we deal also with non-relational-style Open Data.

In our conversion tool, Open Data is based on XML and Closed Data on the relational model. Generally taken, Open Data contains data sets based on varying formats (relational tables, XML, RDF, XLS, PDF, text, etc.). However, XML is both a leading markup language for documents and the de facto standard format for exchanging and sharing data on the Web. In addition, an RDF document of the Semantic Web is an XML document consisting of triples with the fixed structure. Likewise, many other data formats can be converted into XML in a straightforward way. For example, HTML data sources are easily convertible to XML by eliminating the tags intended for displaying [NJJä-14]. One indication of the importance of XML among data formats is that the main relational database systems of IBM, Microsoft and Oracle all support XML publishing. From the foregoing, we can draw the conclusion that several data sets in Open Data are XML-based or easily convertible into it. It has been recognized in several contexts (see e.g. [NCJo-12, HRGa-10]) that today relational databases are used in most enterprise environments to store and manage data. Likewise, several analytics tools (e.g. OLAP tools) are based on that

the underlying data have been organized relationally. Therefore, our conversion tool produces relationally organized data from XML-based Open Data

Our conversion approach resembles the data exchange/translation problem where source data organized according to a specific schema (source schema) are reorganized to conform to the given target schema [FKMP-05]. This transformation is specified declaratively by a high-level formalism, called schema mappings [Kola-05, MHHe-00], which describe the relationship between the source schema and target schema. Data exchange problem has been widely studied in the context of relational data (see e.g. [WyRo-05b]), and recently it has been also studied in the context of XML (see e.g. [ArLi-08, TZWS-13]). Typically, these studies assume that both the source and target data have been represented based on the same data model whereas in our conversion tool source data (XML) and target data (relational) are based on different platforms. Further, the starting point in data exchange research has been that there are the exact schemas for both source and target data. However, it has been recognized (see e.g. [TaGr-10]) that most of the existing XML sources have no schema (DTD or XML schema). Therefore, our conversion tool, unlike existing data exchange tools, is not based on the existence of source and target schema and schema mappings between them. The only background assumption in our conversion tool is that the user knows the element and attribute names of the underlying XML source. By using these names, the user expresses in a straightforward way those relational attribute names of which the target relation consists.

Data mappings in traditional data exchange approaches describe the relationships between source and target data at the schema level (metadata) which induce the corresponding changes at the instance level. Data exchange settings involving instance-level data are called data-to-data translations. However, information of interest in an Open XML data source can be at the different abstraction level as those data in the Closed Data repository (relational database) which are needed for situational analytics. Therefore, our conversion tool has to have the capability both to transform data from the schema level to the instance level (metadata -to-data translations) and from the instance level to the schema level (data-to-metadata translations). Data-to-metadata and metadata-to-data translations are needed in advanced data restructuring. Especially, two data restructuring operations based on these translations, called *pivot* and *unpivot,* have been recognized essential in several contexts (e.g. in OLAP [WBBD+03], database virtualization [TDMS+10] and relational queries [WyRo-05a]). For example, in the relational model, data (attribute values) in rows of a relation are transferred to columns (attribute names) through *pivot* operation whereas *unpivot* is its inverse operation. Our conversion tool supports data restructuring based on *pivot* and *unpivot* operations.

Figure 1 gives an overview on how our conversion approach is used to extract the relevant parts from XML-based Open Data sources and to represent extracted data as conventional relations in the underlying relational database. First, the XML RELATION CONVERTER is used to convert the textual Open XML sources of interest (XML-1, XML-2, …, XML-K) to XML relations (denoted by XML RELATION-1, XML RELATION-2, .., XML RELATION-K), which we can modify freely in a closed environment. It is typical of the XML relation representation (see, [NNJä-09]) that

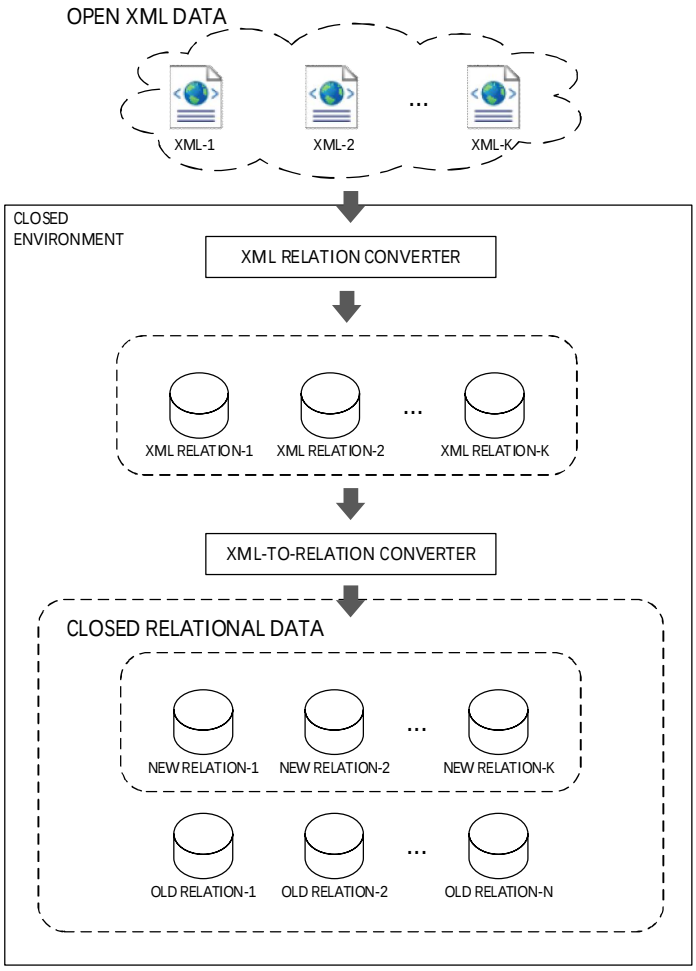


Figure 1. Conversion of Open XML data sources into closed relational data.

any textual XML source can be converted unambiguously into this form and vice versa. The role of XML-TO-RELATION CONVERTER is to extract and restructure information in a specific XML relation representation and produce the result in the form that is compatible with the conventional relations (OLD RELATION-1, OLD RELATION-2, ..., OLD RELATION-N) in the underlying relational database. In Figure 1, NEW RELATION-1, NEW RELATION-2, ..., NEW RELATION-N are conventional relations, constructed from XML RELATION-1, XML RELATION-2, .., XML RELATION-K, respectively. The actual ad hoc analytics based on this information is beyond this paper, i.e., in this paper we deal with the data integration, which is a necessity for this kind of analytics.

The contribution of this paper is to develop XML-TO-RELATION CONVERTER characterized above. For this conversion tool, we pursue the following specific goals.

- Our conversion process described in Figure 1 is able to convert data in the XML (semi-structured) platform to the relational database (structured) platform. For example, in the data exchange settings the platform used in the source and target data typically remains as the same. Among others, this means that our conversion tool has to have the capability to generate null values for indicating missing attribute values in the target relation.
- All those features that can be made in the closed environment are excluded from the conversion tool. For example, information in the result relations NEW RELATION-1, NEW RELATION-2, ..., NEW RELATION-N must often be possible to connect to other relations in the

underlying relational database before ad hoc analytics. This can be done by applying SQL in the closed environment. The conversion tool, in itself, is a very complex process. Therefore, it is not appropriate to include such features in it, which can be done by other tools.

- It must have a great restructuring power, i.e., it has to support data-to-data translations, data-to-metadata translations (*pivot*) and metadata -to-data translations (*unpivot*).
- The conversion tool is specified precisely and comprehensively. In our formal specification, the XML relation representation and its attached constructor algebra have a central role. Because our specification defines exactly how the desired target relation is constructed from a given XML source, we can consider it as the abstract implementation of the conversion tool.
- The conversion tool must have a high degree of automation. This means that the tool have such an analyzing power which is able automatically to find out those structural relationships through which information pieces are associated semantically with each other in the underlying XML source. We show that based on the high degree of automation of our conversion tool it is possible to define a straightforward textual interface, which is at a high abstraction level from the viewpoint of the user.

The rest of the paper is organized as follow. In Section 2, we discuss works related to data exchange, which have similarities and differences with our conversion operation. In this section, we also discuss how and why our XML relation representation deviates from other representations proposed for XML. We specify our conversion tool formally and comprehensively. The essential formal notations and definitions are given in Section 3. In this section, we also give a simple and compact example, which has been used for demonstrating *pivot* operation in different contexts. We believe that this simple example helps the reader to be ensured about the used formalism. In Section 4, we define our conversion tool, which is based on two main phases. In this section, we also consider in detail how our conversion tool produces the desired target relation in the context of our simple example. In Section 5, we give our actual running example, which is larger and more complex than our simple and compact example. Among others, it contains several such features, which were not demonstrated in the context of our simple example. Sections 6 and 7 give the discussion and conclusions, respectively.

2. Related Work

It is typical of most data exchange approaches that both the source and target data are assumed to be based on the same data model. For example, in [FKMP-05, FKPo-05, Barc-09] source and target data are based on the relational model whereas in [ArLi-08, TZWS-13] they are based on XML. In [PVMH+02, HPTa-08] source and target data can be based on the nested relational model, i.e., data exchange can be performed between flat and regularly structured hierarchical relations. Although these works are able to handle hierarchical structures they have not been tailored to handle irregularly structured XML data whose platform differs considerably from the relational one. In our approach, it is important that Open Data are based genuinely on the XML platform. Further, the existing data exchange approaches typically deal with data- to-data translations whereas data-to-metadata and metadata-to data translations are largely ignored [HPTa-08]. The importance of supporting also data-to-metadata and metadata-to data

translations has been recognized in the data exchange settings based on the nested relational model [HPTa-08] and XML [TZWS-13].

Pivot and *unpivot* restructuring operations presuppose capability for data-to-metadata and metadata-to-data translations, respectively. Although the results of these operations can be represented as relations, they do not belong to the original relation algebra whose operations do not transform data between the schema and instance levels. Due to the importance of these restructuring operations, they have been proposed as an extension of the relation algebra [WyRo-05a]. Especially the *pivot* operation deviates considerably from other relational operations. Unlike, in the context of other relational operations, we do not know a priori what attribute names the result relation of this operation will contain. This is due to the fact that the different values of a given attribute in the relation to be pivoted are used as attribute names in the result relation, i.e., the attribute names in the result relation can be defined at runtime of the pivot operation. In [LSSu-96], schemas containing attribute names produced in this way are called dynamic output schemas. In this paper, we consider the restructuring operations *pivot* and *unpivot* in the context of XML data. Our starting point makes the specification of these operations still more challenging because XML data are organized hierarchically and, in addition, they are often irregular and incomplete.

The construction and use of our conversion tool is also different compared to typical data exchange tools. As discussed in [HPTa-08] a typical data exchange framework has three essential phases. First, the user describes the data exchange case of interest by some visual interface. Second, from this description and the underlying schemas, mappings between source and target data are generated. In the third phase, a query expressed by some language is generated from these mappings. The execution of this query constructs and materializes the target data. In other words, a typical data exchange can be characterized as a generation-oriented approach. Instead, our conversion tool does not generate any code, but it is rather an operation with two parameters at a high abstraction level. One of the parameters is an XML source without any separate schema and another parameter is a straightforward description of the target relation by using the data item names in the XML source at hand.

Unlike the existing data exchange approaches, our conversion tool does not utilize separate source and target schemas of data, because most XML-based Open Data sources have no attached schemas (DTD or XML schema) [TaGr-10]. In other words, if we would want to apply this kind of approach we should develop XML-based schema extraction techniques introduced in [MACh-03, HNWe-06] in terms of which the schema for a schemaless XML data source could be constructed. In this paper, we show that the conversion of an XML source can purely be based on its self-descriptiveness property; i.e., in a semi-structured XML source schema-level information (element and attribute names) and instance-level information (values of elements and attributes) co-exist.

If an XML source contains regularly structured data, then several approaches (see e.g. [FlKo-99, STZH+99, HJLP+04]) have been proposed to split an XML source to a set of conventional relations. Even in this case some (e.g. order) information is lost and the original XML source

cannot anymore be constructed from these relations. If the XML source contains information that is incomplete and irregular then the conversion case becomes more difficult from this further. The conversion between XML and relational data is highly non-trivial due to the fundamental differences in their modelling principles. XML data are hierarchical, semi-structured and ordered whereas relational data are flat, structured and unordered. Therefore, it is very hard, if not practically impossible, to map XML data unambiguously to the conventional relations [PCSS+04]. Due to this basic mismatch between XML and relational data, Tan et al. [TZWS-13] discuss why an XML data exchange setting cannot be translated to a relational data exchange setting.

Due to the above reasons, we propose the use of XML relations [NNJä-09] as the intermediate form between the textual XML source and the conventional relation. Although an XML relation is not a conventional relational database relation, it can be stored efficiently in a relational database system because it is structurally compatible with conventional relations. Next, we consider some essential differences between an XML relation and a conventional relation. For example, tuples in the XML relation representation can contain information belonging to the schema level whereas tuples in a conventional relational database system contain only information belonging to the instance level. Likewise, indexes used for indicating locations of information pieces in the underlying XML source have a central role in the XML relation representation whereas conventional relations do not contain this kind of indexing mechanism. Due to the considerable difference between XML relations and conventional relations, the constructor algebra is used for manipulating XML relations whereas conventional relations are manipulated by the relational algebra.

Since the direct conversion of XML data to the conventional relations, as indicated above, is very troublesome, we in our approach do it by using the XML relation representation (its detailed introduction is in Section 3.3.) as an intermediate form. It is typical of the XML relation representation that any XML source can be converted unambiguously to it and vice versa. On the other hand, an XML relation is conceptually and mathematically a relation, although it deviates considerably from relations in relational databases. Due to this conceptual compatibility, it is possible to utilize the efficient storage methods of relational databases. In the XML relation representation, indices are used to locate unambiguously any information piece of an XML source. In the paper, we show that because of indices it is possible to analyze complex structural relationships among XML data. This kind of analyzing capability is crucial in building the conversion tool with a high degree of automation (one of our goals above). The idea to use indices to locate XML data is not new. Indices used in the XML relations are often called Dewey numbers. For example, in ORDPATH [NNPC+04] the location of XML data is based on Dewey numbers. However, in ORDPATH there is one index to denote an attribute/element occurrence (i.e., an index refers to both attribute/element name and its attached value) whereas in the XML relation an attribute/element name and its attached value have different indices. This is necessary because in the restructuring of our conversion tool we have to transform data between the schema and instance levels.

Our XML relation representation approach also differs from ORDPATH in the sense that we have defined the constructor algebra for manipulating XML relation representations. The special feature of our constructor algebra is that its operations are able automatically to index the result XML relation based on the indices of operand relations. This is a very important feature when constructing complex XML fragments from simpler XML fragments represented as XML relations. In the paper we show that the XML relation-based result (or Phase I in 4.1) of our conversion tool can be implemented by this algebra. In other words, the purpose of use of our constructor algebra differs considerably from those XML algebras (see [FFMR+01, LPVe-00, CCSi-00, JLST-01] whose idea is to extract, select and navigate XML data.

3. Notational Conventions and Definitions

As our goal is to specify the conversion tool generally and exactly, we next introduce some notational conventions and definitions that are needed for this purpose.

In an XML relation, the order among data item (attribute or element) occurrences is represented unambiguously based on indices. Therefore, we first introduce how indices are represented and manipulated. After that, we give our constructor algebra in terms of which XML relations are manipulated in our conversion operation. The automatic re-indexing mechanism has a central role in the constructor operations. In addition to the XML source, we need conventions for representing the target relation schema related to the result. In the context of the target relation schema, we need the expressions for the advanced restructuring operations, *pivot* and *unpivot,* which produce such relational attribute names whose counterparts do not appear as such in the XML source at hand. Finally, we introduce how data are represented in the result or in the relational database environment.

3.1 Set-Theoretic Notations

In the specification of our conversion operation, we manipulate both sets and power sets (i.e., sets consisting of sets). Let S denote a set; then $\{S\}$ is a power set whose only element is the set S. We need this expression to form power sets. For example, let $S_1 = \{a, b\}$, $S_2 = \{a, c\}$, and $S_3 = \{b, c\}$ be sets; then $\bigcup_{i=1}^{3} S_i = \{a, b, c\}$, whereas $\bigcup_{i=1}^{3} \{S_i\} = \{\{a, b\}, \{a, c\}, \{b, c\}\}$. In our definitions, we also need to be able to analyze the type of the set at hand. We use the predicates is_set(S) and is_powerset(S) to determine whether S is a set, whose elements are not sets, or a power set, respectively.

3.2. Indices and Index Manipulation

In addition to set-theoretic notations, we also need formal representations for indices. Our indexing scheme is based on the Dewey-style structural indices that allow easy access to the ancestors and descendants of a specific data item. An index consists of a sequence of positive integers denoted between angle brackets. For example, $\langle 1,2,3,4 \rangle$ is an index with four components. The length of an index *index* is denoted by length(*index*). For example, length($\langle 1,2,3,4 \rangle$) = 4.

We refer to any two parts of an index *index* as *index* = ⟨$part_1 \perp part_2$⟩ where $part_1$ represents the components belonging to the first part of *index,* whereas $part_2$ is the index consisting of the remaining components in *index*. We use the symbol ξ or a string (e.g., *ind*) to refer to a sub-index, whereas single letters or integers (e.g., *i* or 1) refer to atomic index components. Thus, for example, the expression ⟨$i \perp \xi$⟩ applied to the index ⟨1,3,1,4⟩ means that *i* refers to the first component of the index (i.e., *i* = 1) and ξ is the sub-index ⟨3,1,4⟩. Similarly, the expression ⟨*ind* $\perp j$⟩ means that *ind* = ⟨1,3,1⟩ and *j* = 4. Correspondingly, the expression ⟨$i, j \perp \xi$⟩ results in *i* = 1, *j* = 3, and ξ = ⟨1,4⟩. These expressions are also utilized for generating new indices. For example, if *I-Set* is the index set {⟨1⟩, ⟨2⟩, ⟨2,1⟩} then the set expression {⟨1 ⊥ *index*⟩ | *index* ∈ *I-Set*} produces the index set {⟨1,1⟩, ⟨1,2⟩, ⟨1,2,1⟩}. Analogously, if *index* is the index ⟨1,3,2⟩ then the expression ⟨*index* ⊥ 1⟩ means the index ⟨1,3,2,1⟩.

The function lcp returns the longest common prefix of two indices. It is defined as follows.

$$\text{lcp}(index_1,\ index_2) = index, \tag{3.1}$$

if $index_1 = \langle index \perp \xi_1 \rangle \wedge\ index_2 = \langle index \perp \xi_2 \rangle \wedge \neg\exists\ index'\ (\neq index)\colon index_1 = \langle index' \perp \xi'_1 \rangle \wedge index_2 = \langle index' \perp \xi'_2 \rangle \wedge \text{length}(index') > \text{length}(index)$.

For example, the longest common prefix of the indices ⟨1,2,3,4,5⟩ and ⟨1,2,3,5,4⟩ is lcp(⟨1,2,3,4,5⟩, ⟨1,2,3,5,4⟩) = ⟨1,2,3⟩.

3.3. XML Relation Representation and Constructor Algebra

The constructor algebra builds the XML relation representation for any textual XML source by applying its operations, starting from the atomic XML data items (an element name, an attribute name, and a value). The XML relation representation for more complex XML structures are constructed by combining the XML relation representations of simpler structures. In this case, the operations of the constructor algebra automatically re-index the XML relation representations of the simpler structures. Next, we give the formal definition of our constructor algebra (based on [NNJä-09]).

Definition 1: An XML relation is constructed recursively by finite application of the following rules:

1. Let *v* denote the content or a single content component such as a word (if the content is a string) of an attribute or element. Now *v* is represented as an XML relation {(*v*, 'v', ⟨1⟩)}, where the constant 'v' is used to indicate that *v* belongs to the content of some attribute or element.
2. An attribute name *a* is represented as an XML relation {(*a*, 'a', ⟨1⟩)}. Here, the constant 'a' indicates the type of *a;* that is, it is an attribute name.
3. An element name *e* is represented as an XML relation {(*e*, 'e', ⟨1⟩)} with the constant 'e' expressing that *e* is an element name.

4. If R_1 and R_2 are two XML relations, then the concatenation constructor $R_1 \diamond R_2$ constructs an XML relation

$$R_1 \cup \text{index\_transformation}(\text{maxfirst}(R_1), R_2)$$

where $\text{maxfirst}(R_1) = |\{(c, t, i) \mid (c, t, i) \in R_1 : \text{length}(i) = 1\}|$.

In other words, the function maxfirst expresses the number of those indices in R_1 whose length is 1 (due to our indexing mechanism the cardinality above expresses the maximum first component among the indices of R_1), and $\text{index\_transformation}(j, R_2) = \{(c, t, \langle (i + j) \perp ind \rangle) \mid (c, t, \langle i \perp ind \rangle) \in R_2\}$. It is also worth noting that in the resulting XML relation the tuples in R_1 remain as such whereas the tuples in R_2 are re-indexed by summing the integer j (i.e., the result of maxfirst(R_1)) with the first components of the indices.

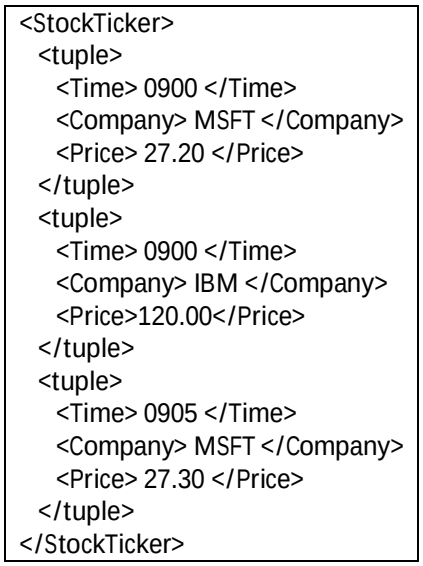
5. Let *A* be an attribute name as an XML relation (see, Rule 2) and *R* its content as an XML relation. The attribute constructor $A\ \theta\ R$ constructs the XML relation $A \cup \{(v, \text{'v'}, \langle 1 \perp ind \rangle) \mid (v, \text{'v'}, ind) \in R\}$. Here, the length of each index in *R* is incremented by inserting '1' as the first component.
6. If *E* represents an element name as an XML relation (see, Rule 3) and *R* its content with possible (nested) substructure as an XML relation, then the element constructor $E\ \omega\ R$ constructs an XML relation $E \cup \{(c, t, \langle 1 \perp ind \rangle) \mid (c, t, ind) \in R\}$ where $t \in \{\text{'e'}, \text{'a'}, \text{'v'}\}$.

To summarize, Rules 1–3 are used to represent the atomic XML data items as XML relations, whereas Rules 4–6 are used to construct more complex XML structure occurrences. Rule 4 is used to combine XML data items that are at the same hierarchy level. Rules 5 and 6 are used for constructing attribute and element occurrences, respectively. The above constructors have the closure property; that is, both their operands and their results are XML relations. Therefore, the constructors can be flexibly nested.

The root element comprises all the other elements, attributes, and content components of an XML source. In the XML relation representation, this element occurrence is associated with the index $\langle 1 \rangle$. We also assume that each XML source has a unique name, which is used to refer to the corresponding XML relation. The XML relation representation for an XML source *D* can thus be defined as follows.

Definition 2: The XML relation representation of an XML source *D* is a ternary relation *D*(*C, T, I*) whose tuples have been constructed through Rules 1–6 in Definition 1. In it, the tuple (*r*, 'e', $\langle 1 \rangle$) expresses the name of the root element of *D.*

Figure 2 shows a textual XML source and its XML relation representation.



C	T	I
StockTicker	'e'	⟨1⟩
tuple	'e'	⟨1,1⟩
Time	'e'	⟨1,1,1⟩
0900	'v'	⟨1,1,1,1⟩
Company	'e'	⟨1,1,2⟩
MSFT	'v'	⟨1,1,2,1⟩
Price	'e'	⟨1,1,3⟩
27.20	'v'	⟨1,1,1,3⟩
tuple	'e'	⟨1,2⟩
Time	'e'	⟨1,2,1⟩
0900	'v'	⟨1,2,1,1⟩
Company	'e'	⟨1,2,2⟩
IBM	'v'	⟨1,2,2,1⟩
Price	'e'	⟨1,2,3⟩
120.00	'v'	⟨1,2,1,3⟩
tuple	'e'	⟨1,3⟩
Time	'e'	⟨1,3,1⟩
0905	'v'	⟨1,3,1,1⟩
Company	'e'	⟨1,3,2⟩
MSFT	'v'	⟨1,3,2,1⟩
Price	'e'	⟨1,3,3⟩
27.30	'v'	⟨1,3,1,3⟩

Figure 2. A textual XML source and its XML relation representation.

For our conversion operation, we need to extend the concatenation constructor (Rule 4 of Definition 1) to also cover the situation, in which the operands may be empty sets or power sets whose elements are XML relations. We denote this extension of the concatenation operation by <> and define it as follows.

$$S_1 <> S_2 = \begin{cases} S_1, & \text{if } S_2 = \emptyset \\ S_2, & \text{if } S_1 = \emptyset \\ S_1 \diamond S_2, & \text{if is\_set}(S_1) \wedge \text{is\_set}(S_2) \\ \{S_1 \diamond t | t \in S_2\}, & \text{if is\_set}(S_1) \wedge \text{is\_powerset}(S_2) \\ \{t \diamond S_1 | t \in S_1\}, & \text{if is\_powerset}(S_1) \wedge \text{is\_set}(S_2) \\ \{t_1 \diamond t_2 | t_1 \in S_1 \wedge t_2 \in S_2\}, & \text{if is\_powerset}(S_1) \wedge \text{is\_powerset}(S_2) \end{cases} \quad (3.2)$$

It is worth noting that the above operation is able to concatenate each possible combination between XML relations in its operands.

3.4. Functions and Predicates for Analyzing XML Structures

XML sources are commonly visualized as ordered trees, consisting of nodes and the edges between them. This means that in traversing an XML source we need to analyze its nodes. The XML relation representation and its indexing mechanism greatly facilitate this kind of analyzing. Next, we define some functions and predicates for this purpose.

In the tree visualization of an XML source, a node is a leaf if it has only a value as its child (i.e., no other data item names). The predicate leaf(*index*) is true if the index *index* is associated with a data item name that is a leaf in the XML relation *X-Rel* at hand. It is defined as follows.

$$\text{leaf}(index) = \begin{cases} \text{true, if } (n, \_, index) \in X\text{-}Rel \wedge (v, \text{'v'}, \langle index \perp 1 \rangle) \in X\text{-}Rel \\ \text{false, otherwise.} \end{cases} \quad (3.3)$$

For example, in the context of the sample XML relation in Figure 2 leaf(⟨1, 1, 2⟩) = true (the index ⟨1, 1, 2⟩ is associated with the data item name Company), whereas leaf(⟨1, 2⟩) = false since data item name tuple with the index ⟨1, 2⟩ has other data item names as its children.

If a data item name with the index *index* is not a leaf, then the indices of its children in the XML relation *X-Rel* can be retrieved by the function children(*index*), which is defined as follows:

$$\text{children}(index) = \{\langle index \perp i \rangle | (\_, t, \langle index \perp i \rangle) \in X\text{-}Rel : t \in \{\text{'e'}, \text{'a'}\}\}. \quad (3.4)$$

In the context of our sample XML relation, children(⟨1, 2⟩) = {⟨1, 2, 1⟩, ⟨1, 2, 2⟩, ⟨1, 2, 3⟩}. This means that the tuple element with the index ⟨1, 2⟩ has three children: the data item names Time, Company, and Price associated with the above indices.

The children of a data item can be similar or dissimilar with each other with respect to their information contents. In XML, the similar data item occurrences are typically organized by using the same data item names and structures in them. Consequently, when all the children of a data item have the same name they can be assumed to represent similar information. In gathering information about the children of a data item, we have to know whether they represent similar information. For this, we define the predicate similar(*I-Set*) where *I-Set* is a set of indices, as follows:

$$\text{similar}(I\text{-}Set) = \begin{cases} \text{true, if } |\{n | index \in I\text{-}Set : (n, \_, index) \in X\text{-}Rel\}| = 1 \\ \text{false, otherwise.} \end{cases} \quad (3.5)$$

In our example, similar({⟨1, 1⟩, ⟨1, 2⟩, ⟨1, 3⟩}) = true, since all these indices are associated with the data item name tuple. Instead, similar({⟨1, 1, 1⟩, ⟨1, 1, 2⟩, ⟨1, 1, 3⟩}) = false, meaning that the children of the tuple data item with the index ⟨1, 1⟩ are not similar with each other.

The data item occurrences at the same hierarchical level in the XML structure are called siblings, which also means that they depend immediately on the same data item occurrence. This information can be utilized in retrieving siblings. Due to our indexing mechanism, the indices associated with siblings differ only in their last index component. We define the function next_sibling(*I-Set*) to retrieve the smallest index in the ascending document order from the index set *I-Set*, as follows:

$$\text{next\_sibling}(I\text{-}Set) = \langle index \perp i \rangle, \text{ if } \neg\exists j < i : \langle index \perp j \rangle \in I\text{-}Set. \quad (3.6)$$

Thus, in our sample XML relation next_sibling(⟨1, 1, 1⟩, ⟨1, 1, 2⟩, ⟨1, 1, 3⟩) returns the index ⟨1, 1, 1⟩.

Intuitively, based on our indexing mechanism, the closest occurrences of the data item n_1 with the data item n_2 are the ones whose indices share the longest common prefix. The function closest(*index, n*) returns the index of an occurrence of the data item name *n* that is closest to the data item occurrence with the index *index*. It is defined as follows.

$$\text{closest}(index, n) = index', \tag{3.7}$$

if $(n, \_, index') \in X\text{-}Rel \wedge \neg\exists(n, \_, index'') \in X\text{-}Rel$: $\text{length}(\text{lcp}(index, index'')) > \text{length}(\text{lcp}(index, index'))$.

In our sample XML relation, closest(⟨1, 1, 3⟩, Time) = ⟨1, 1, 1⟩. In other words, the closest occurrence of the Time data item with respect to the (Price) data item occurrence with the index ⟨1, 1, 3⟩ is the one with the index ⟨1, 1, 1⟩. It is worth noting that the function closest is able to obtain the closest data item regardless of the hierarchical level at hand.

3.5. Formalism for the Conventional Relational Databases

Although an XML relation is mathematically a relation which is able to represent an XML source relationally it is not such a relation notion on which relational database systems are based. For example, in the conventional relational database system a tuple consists only of information belonging to the instance level (i.e., attribute values), whereas a tuple in the XML relation can contain information which can belong either to the schema or instance level. Likewise, the index-based location information plays an essential role in the XML relation expression, whereas a relation of a conventional relational database lacks this kind of information. Therefore, our conversion operation contains a phase in which an XML relation organized according to the given target schema is converted into a conventional, relation processable in the relational database environment. For this, we need the formalism introduced in this section.

According to Ullman [Ullm-88], relational tuples can be viewed as mappings from attributes' names to values belonging to the domains of the attributes. An alternative to represent this kind of mappings is to give attribute name–attribute value pairs. Here, an attribute name (denoted by *a-name*) and its value (denoted by *value*) are represented as the term *a-name*(*value*). Based on this convention, a relational tuple is represented as (*a-name*$_1$(*value*$_1$), *a-name*$_2$(*value*$_2$), ... , *a-name*$_k$(*value*$_k$)) where *a-name*$_1$, *a-name*$_2$, ... , *a-name*$_k$ are attribute names and *value*$_1$, *value*$_2$, ... , *value*$_k$ are their values in a tuple, respectively. In our formalism, the tuple can also be empty, which is denoted by Λ.

In our conversion operation, we will construct a relational tuple by concatenating attribute name–attribute value pairs with each other. For this, we need the tuple concatenation operation concatenate(*tuple*$_1$,*tuple*$_2$), defined as follows:

concatenate((*name*$_1$(*value*$_1$), *name*$_2$(*value*$_2$), …, *name*$_k$(*value*$_k$)), (*name*$_1$'(*value*$_1$'), *name*$_2$'(*value*$_2$'), …, *name*$_k$'(*value*$_k$')) = (*name*$_1$(*value*$_1$), *name*$_2$(*value*$_2$), …, *name*$_k$(*value*$_k$), *name*$_1$'(*value*$_1$'), *name*$_2$'(*value*$_2$'), …, *name*$_k$'(*value*$_k$')).

(3.8)

The empty tuple Λ behaves in the concatenate operation as follows: concatenate(Λ, *tuple*) = concatenate(*tuple*, Λ) = *tuple*. For example, concatenate((Time(0900), Company(IBM)), (Price(120.00)), Λ) = (Time(0900), Company(IBM), Price(120.00)).

4. Formal Definition of the Conversion Operation

4.1. Formal Representation of the Relational Target Schema

Our goal is to provide the user with a powerful conversion operation that is straightforward to use. In the following, we will show that this operation needs only two kinds of information as its input: the existing XML source and the relational target schema. The operation produces the result relation organized according to the given relational target schema. The starting point of the use of our operation is that the user knows the data item names (i.e., attribute and element names) in the XML source at hand. This is important because we use these data item names as such in relational attribute names of the result relation. Of course, we could rename the data item names in the target relation in the context of the operation. There are two reasons for avoiding renaming. First, it would make our operation more complicate to use because, due to the renaming, we should add the schema mapping facility similar to data exchange (see, [Kola-05]). Second, the renaming of relational attributes can be done quite simply in the relational database system once the target relation has been constructed.

In our operation, the target schema contains both those attribute names which are extracted as such from the underlying XML source and expressions for constructing new attribute names through pivoting and unpivoting. Generally taken, a relational target schema is represented as a term *RN*(A_1, A_2, …, A_n) where *RN* is the name of the target relation and each A_i ($1 \le i \le n$) is either a relational attribute name, which has the corresponding data item name in the XML source, or an expression for generating new attribute names for the target relation. In the context of a target relation, we use the following predicates to test the type of a specific A_i.

- The predicate basic(A_i) is true, if A_i has the corresponding data item name in the XML source. In other cases, it is false.
- The predicate pivot(A_i) is true, if A_i is a pivot expression for generating new attribute names based on the values of a specific data item name. In other cases, it is false.
- The predicate unpivot(A_i) is true, if A_i is an unpivot expression for generating two new attribute names whose values are defined in the expression. In other cases, it is false.

It is obvious that the above predicates are exclusive; i.e., only one of the above predicates can be true in the context of any A_i.

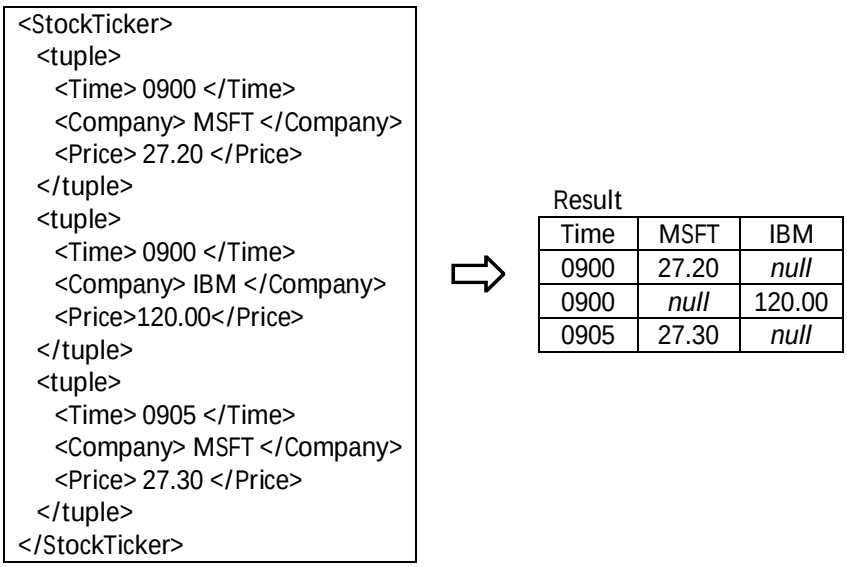


Figure 3. XML-to-relation conversion with pivot.

4.1.1 Pivot

As recognized in [HPTa-08], we cannot determine a priori the attribute names obtained by pivoting. This kind of schema is sometimes called a dynamic output schema [LSSu-96]. It is typical of pivoting that the related attribute names are generated at run-time. This means that we need such an expression that is able to generate these attribute names when processing our conversion operation. In addition to the names of the source (in our case an XML document) and target (in our case a relation) depositories, in a textual pivot specification (see. e.g. [TDMS+10]) we need to express both the source data item name (denoted by *N*) holding the values that will be pivoted to form new attribute names in the result and the source data item name (denoted by *V*) whose values are used as the values of pivoted (generated) attributes. Based on these notations the pivot operation is expressed by the notation *N*(*V*) in our target schema; i.e., if some A_i in the target schema has been expressed in this way, then the predicate pivot(A_i) is true. The source data item names *N* and *V* may have several occurrences in the underlying XML source. Therefore, in pivoting it is important to be able to connect the semantically related values of *V* to the values of *N* in the XML source at hand. For this purpose, we can use our function closest, defined in Formula 3.7.

In what follows, we will illustrate the specification of our conversion operation through the XML-to-relation conversion depicted in Figure 3. It is based on the example used in [HPTa-08]. It is obvious that this conversion involves pivoting. Hence, the related target schema is of the form Result(Time, Company(Price)) where Result is the name of the target relation, Time is a basic attribute name, and Company(Price) is a pivot expression indicating a set of attribute names that will be generated at run-time based on the distinct values of the Company data items (an XML element). The thus generated attributes will be populated with the values of the semantically related Price source data items. We shall later refer to the above target schema as STS1.

Let *TS* be a target schema $RN(A_1, A_2, \ldots, A_n)$; then the predicate r_name(*TS*) gives the relation name (*RN*) and the function e_set(*TS*) yields the set $\{A_1, A_2, \ldots, A_n\}$. In the context of the target schema STS1, r_name(STS1) = Result and e_set(STS1) = {Time, Company(Price)}. If A_i is a pivot expression in a target schema *TS* (i.e. $A_i \in$ e_set(*TS*): pivot(A_i) = true), then the predicate gen(A_i) gives the source data item name whose values will be pivoted to form the new attribute names in the target relation. Similarly, the predicate inst(A_i) gives the source data item name whose values are used as values of the generated attribute names. For example, in the context of STS1, the predicate gen(Company(Price)) gives Company and the predicate inst(Company(Price)) gives Price.

4.1.2 Unpivot

Unpivot is a kind of an inverse operation to pivot in the sense that it transfers information from the schema level (i.e., metadata) to the instance level. Let A_i be an unpivot expression (i.e., unpivot(A_i) is true); then A_i contains the following information:

(a) those source data item names that will be represented as the values of a specific new attribute in the target relation, and
(b) the new attribute name in the target relation that contains the values related to the source data item names expressed in *A*.

In our conversion operation, an unpivot expression is represented as a term $Attr([DN_1, DN_2, \ldots, DN_n], B)$. In this expression, *Attr* is an attribute in the target relation whose values will be the data item names $DN_1, DN_2, \ldots, DN_n$ in the XML source at hand. On the other hand, *B* is an attribute in the target relation whose values will be the values of the data items $DN_1, DN_2, \ldots, DN_n$. In other words, *Attr* and *B* are new metadata items not appearing in the original XML source. If there is an unpivot expression $A_i = Attr([DN_1, DN_2, \ldots, DN_n], B)$ in the target relation *TS* (i.e., $A_i \in$ e_set(*TS*): unpivot(A_i) = true) such that a source data item name *DN* appears in $[DN_1, DN_2, \ldots, DN_n]$, then the predicate to_be_unpivoted(*DN*) is true. The predicate a-name(*DN*) yields the target attribute name whose value *DN* will be (i.e., a-name(*DN*) = *Attr*) and the predicate v-name(*DN*) yields the target attribute name whose values the values of *DN* will be (i.e., v-name(*DN*) = *B*). Finally, the predicate new_attributes expresses the two new attribute names that are generated to the result relation through unpivoting (i.e., new_attributes(A_i) = {*Attr*, *B*}).

Figure 4 describes a conversion case involving unpivoting. (The case is conceptually the inverse of the conversion case depicted in Figure 3.) In this conversion case, we have the target schema Result2(Time, Company([MSFT,IBM]), Price)) where Company([MSFT,IBM]), Price) is an unpivot expression. Based on this schema, both to_be_unpivoted(MSFT) and to_be_unpivoted(IBM) are true. Further, the predicates a-name(MSFT) and a-name(IBM) yield Company and the predicates v-name(MSFT) and v-name(IBM) yield Price. The predicate new_attributes(Company([MSFT,IBM]), Price)) yields the set {Company, Price}.

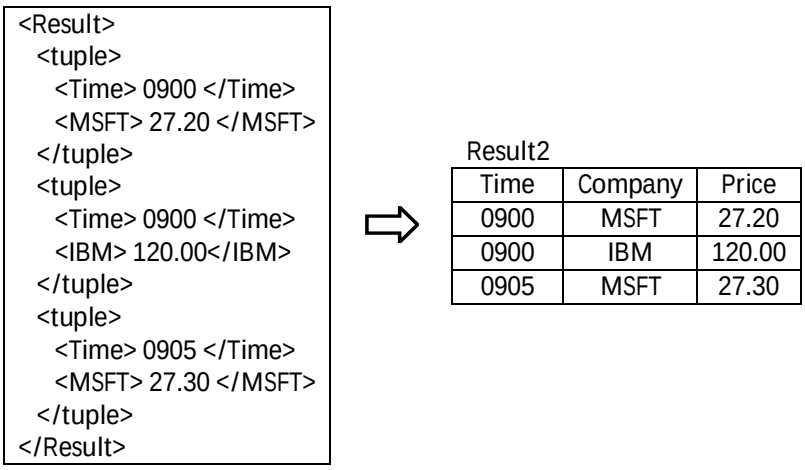


Figure 4. XML-to-relation conversion with unpivot.

4.2. Phase I: The Construction of the Intermediate Form

Recall from Section 1 that we first convert the textual XML sources retrieved from the Open Data environment into the corresponding XML relations. In the first phase of our conversion operation, we extract and restructure the relevant information from these XML relations and produce the intermediate form consisting of information expressed in the relational target schema at hand. The intermediate form is a set consisting of elements which themselves are XML relations. Each element contains all semantically related information that is needed for constructing a tuple of the result relation; i.e. the number of XML relations in the intermediate form is the same than the number of tuples (rows) in the result relation. In the second phase of our conversion operation, a conventional relation that is processable in relational databases is constructed. Next, we define these two phases in detail and we assume that the reader knows the formalisms and functions introduced above.

In the tree visualization of an XML source, the leaf nodes provide the information content from which the result relation is constructed. All the other nodes are element name occurrences that group hierarchically the information in the leaves. In this paper, we call these nodes non-leaf nodes. Typically, in an XML source there are several nodes (attribute or element name occurrences) labeled identically. Therefore, it is important to find out what leaf nodes are associated semantically with each other in the closest way. Let *l* be a leaf node, then the leaf node occurrence with a specific name (say *a*) is related semantically to *l* if this leaf node occurrence has the lowest common ancestor with *l* among all nodes with name *a*. We can use the function closest defined in Formula 3.7 to obtain the leaf node occurrence with a specific name that is semantically related to a given leaf node. It is worth noting that in XML the leaf nodes may lie at the different hierarchical levels. In an XML relation, the length of the index attached to a leaf node expresses its hierarchical level. Further, if *ind* is the index of a leaf node (an attribute or element name) in an XML relation, then the index $\langle ind \perp 1 \rangle$ expresses its value. Since leaf nodes do not contain any substructures, there is only one index whose prefix is *ind*.

The role of the first phase in our conversion operation is to select, flatten and restructure the information in the XML source into the intermediate form. Let *A* and *B* be two data item names whose occurrences are leaf nodes such that *A* is at a higher hierarchical level than *B*. In this case, in constructing the intermediate form we have to duplicate an occurrence of *A* for each occurrence of *B* that is semantically associated with it. We can implement this duplication through the generalized concatenation operation of the constructor algebra introduced in Section 3.3. Through duplication, we flatten information in the XML source. This is necessary because the relational model is based on non-hierarchical (flat) structures. In this context, we have also to restructure (through pivoting or unpivoting) existing XML data for the result relation.

In our conversion operation, the first phase is carried out by the function extract&restructure(*X-Rel, TS, I-Set*) where *X-Rel* is the XML relation representation of an XML source, *TS* is a target schema organized according to Section 4.1, and *I-Set* is a set consisting of the indices of the data item name occurrences are to be traversed next in the underlying XML source. Initially, *I-Set* has the value {⟨1⟩}; i.e. the traversal of an XML source starts from its root.

The function extract&restructure is defined recursively so that every node in the tree visualization of the XML source at hand is traversed. As explained above, a special interest is in the cases where a leaf node is met in the traversal of an XML tree. Typically, we make from each leaf node a separate XML relation, which is connected to other separate XML relations related to other leaf nodes by the generalized concatenation operation. It is worth noting that the concatenation operation produces a larger XML relation in which information of separate XML relations is automatically re-indexed. Before the formal definition of the function, we informally consider those different cases of which the traversal of an XML source consists.

CASE 1: If a non-leaf node is met, then we have to traverse all its children recursively. In this case, the set I-Set consists only of one index. For example, the traversal of the root of an XML source represents this case.

CASE 2: If a leaf node is met, we have four options for its manipulation. First, if the leaf node at hand does not appear in the target schema, then it is not taken into the intermediate form. This is indicated so that the function yields the empty set as the result. It is worth noting that the generalized concatenation operation defined in Formula 3.2 is able to handle empty sets as its arguments; i.e. this kind of leaf nodes are ignored when concatenating leaf nodes.

CASE 3: Here, the situation is similar to CASE 2 except that the leaf node at hand appears in the target schema; i.e. it is taken into the intermediate form. By the constructor algebra, we make from this leaf node and its value a separate XML relation, which is connected to other extracted and/or restructured information in constructing the intermediate form.

CASE 4: This case concerns the situation where the encountered leaf node is participated in a pivot expression in the target schema *TS,* so that its value is used as an attribute name in the result relation. In this case, such a separate XML relation is constructed, in which the value of the encountered leaf node appears as an element name and whose value is the value of

the data item expressed in the target schema, which is associated semantically with the encountered leaf node.

CASE 5: This case is related to situation where the encountered leaf node has to be unpivoted. This means that two pieces of new information must be constructed for the intermediate form. We have to construct both the XML relation, where the encountered leaf node appears as the value of the relational attribute expressed in the target schema, and the XML relation, where the value of the encountered leaf node appears as the value of the relational attribute expressed in the target schema. The concatenation of these two XML relations contains the unpivoted information related to encountered leaf node.

CASE 6: In the traversal of the XML source, we have the situation where we are dealing with substructures depending on a specific node (these substructures are drawn from CASE 1). If these substructures (*I-Set* contains the indices from which the substructures start) have been labeled identically (i.e., similar(*I-Set*) = true) then they can be safely assumed to contain similar information. This means that we have to traverse each substructure and extract the needed information from them. The extracted information per each substructure is represented as its own XML relation. Through union, we express the total information extracted from these substructures. However, each constructed XML relation must be represented as its own element in the result of the union. This is because the semantic relationships among extracted data must be preserved. In other words, a power set, whose elements are XML relations, is produced. Through the generalized concatenation operation, each constructed XML relation is connected to the other extracted and/or restructured information.

CASE 7: This case differs from CASE 6 in the respect that substructures depending on a specific node are not similar with each other (i.e. similar(*I-Set*) = false). In this case, we traverse each substructure, extract the relevant information from them, and concatenate this information together. Substructures are dealt with in the order given by the function next_sibling (defined in Formula 3.6). Finally, *I-Set* contains only one index that is treated based on the means of CASE 1 to CASE 5.

The formal definition of the function extract&restructure is given in Formula 4.1. (The functions leaf, children, e-set, basic, inst, gen, a-name, v-name, to_be_unpivoted, similar, and next_sibling applied in this formula were defined in the previous sections.)

extract&restructure(*X-Rel, TS, I-Set*) =

$$
\begin{cases}
\text{extract\&restructure}\big(\textit{X-Rel, TS,}\ \text{children}(\textit{index})\big), \\
\quad \text{if } \textit{I-Set} = \{index\} \wedge \text{leaf}(index) = \text{false} \ //\ \textbf{CASE 1} \\
\\
\emptyset, \\
\quad \text{if } \textit{I-Set} = \{index\} \wedge \text{leaf}(index) = \text{true} \wedge \{c \mid (c, \_, index)\} \not\subset \text{e\_set}(TS) \ //\ \textbf{CASE 2} \\
\\
\{(c, \text{'e'}, \langle 1 \rangle)\}\ \omega\ \{(v, \text{'v'}, \langle 1 \rangle)\} \\
\quad \text{where } (v, \text{'v'}, \langle index \perp 1 \rangle) \in \textit{X-Rel}, \\
\quad \text{if } \textit{I-Set} = \{index\} \wedge \text{leaf}(index) = \text{true} \wedge \\
\quad \{c \mid (c, \_, index)\} \subset \text{e\_set}(TS) \wedge \text{basic}(c) = \text{true} \ //\ \textbf{CASE 3} \\
\\
\{(v, \text{'e'}, \langle 1 \rangle)\}\ \omega\ \{(t, \text{'v'}, \langle 1 \rangle)\} \\
\quad \text{where } (c, \_, index) \in \textit{X-Rel} \wedge (v, \text{'v'}, \langle index \perp 1 \rangle) \in \textit{X-Rel} \wedge \\
\quad (\text{inst}(expr), \_, ind') \in \textit{X-Rel} : \text{closest}(index, inst(expr)) = ind' \\
\quad \wedge (t, \text{'v'}, \langle ind' \perp 1 \rangle) \in \textit{X-Rel}, \\
\quad \text{if } \textit{I-Set} = \{index\} \wedge \text{leaf}(index) = \text{true} \wedge \\
\quad \exists expr \in \text{e\_set}(TS) : \text{pivot}(expr) = \text{true} \wedge \text{gen}(expr) = c \ //\ \textbf{CASE 4} \\
\\
\{(\text{a-name}(c), \text{'e'}, \langle 1 \rangle)\}\ \omega\ \{(c, \text{'v'}, \langle 1 \rangle)\} <> \{(\text{v-name}(c), \text{'e'}, \langle 1 \rangle)\}\ \omega\ \{(v, \text{'v'}, \langle 1 \rangle)\} \\
\quad \text{where } (c, \_, index) \in \textit{X-Rel} \wedge (v, \_, \langle index \perp 1 \rangle) \in \textit{X-Rel}, \\
\quad \text{if } \textit{I-Set} = \{index\} \wedge \text{leaf}(index) = \text{true} \wedge \text{to\_be\_unpivoted}(c) = \text{true} \ //\ \textbf{CASE 5} \\
\\
\cup_{index \in \textit{I-Set}} \{\text{extract\&restructure}(\textit{X-Rel, TS,}\ \{index\})\}, \\
\quad \text{if } |\textit{I-Set}| > 1 \wedge \text{similar}(\textit{I-Set}) = \text{true} \ //\ \textbf{CASE 6} \\
\\
\text{extract\&restructure}(\textit{X-Rel, TS,}\ \{index\}) <> \text{extract\&restructure}(\textit{X-Rel, TS, I-Set} - \{index\}) \\
\quad \text{where next\_sibling}(\textit{I-Set}) = index, \\
\quad \text{if } |\textit{I-Set}| > 1 \wedge \text{similar}(\textit{I-Set}) = \text{false} \ //\ \textbf{CASE 7}
\end{cases}
$$

(4.1)

The function extract&restructure(*X-Rel, TS, I-Set*) produces the intermediate form represented as the set {*XML-Rel*$_1$, *XML-Rel*$_2$, ... , *XML-Rel*$_n$} where each element *XML-Rel*$_i$ ($1 \le i \ge n$) is an XML relation of the form {(e_1, 'e', ⟨1⟩), (v_1, 'v', ⟨1,1⟩), (e_2, 'e', ⟨2⟩), (v_2, 'v', ⟨2,1⟩), ..., (e_k, 'e' ⟨*k*⟩), (v_k, 'v', ⟨*k*,1⟩)}. Each e_j ($1 \le j \le k$) is an XML element name, which appears as a relational attribute in the result relation to be constructed, and v_j is its value. An XML relation *XML-Rel*$_i$ in the intermediate form contains semantically related information for constructing one tuple (row) for the result relation.

It is possible that an XML relation *XML-Rel*$_i$ does not contain information for all attributes of the result relation. There are two basic reasons for that. First, data in XML sources are often incomplete and their structure may be irregular. For example, when traversing an XML tree we

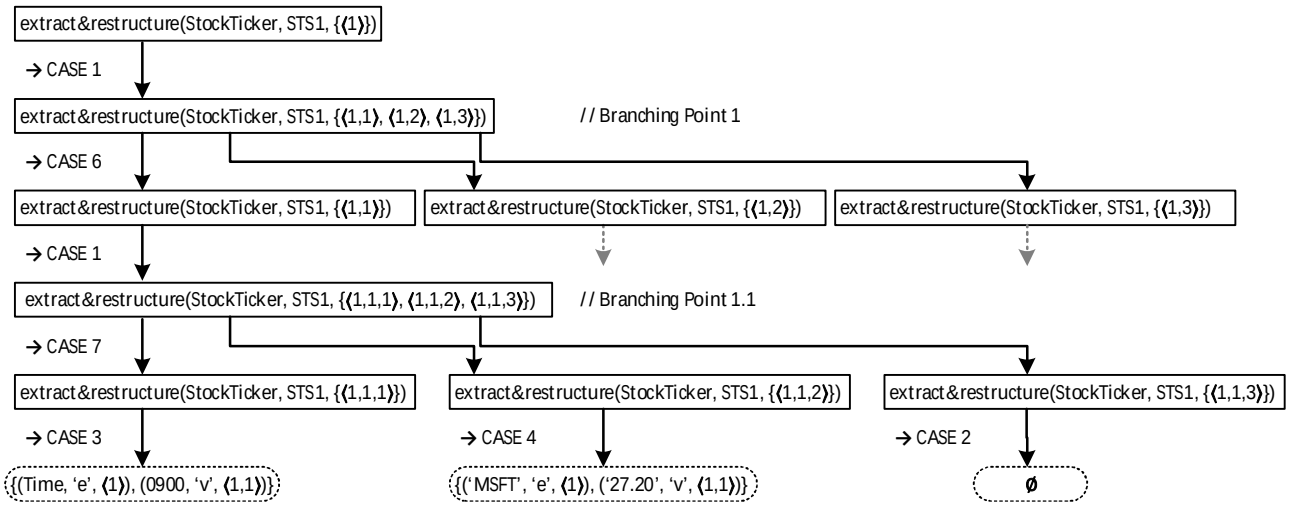


Figure 5. The (partial) evaluation tree related to Phase I of Example 1.

may meet a node (CASE 6 above) whose children have been labeled identically (and thus supposedly contain similar information) but some specific piece of information may appear in the context of some child whereas it is missing from some other child. Second, the values of a leaf node used in the pivoting may differ. This means that a generated data item name will appear only in some XML relations of the intermediate form.

Example 1. In order to ensure that the reader has interpreted the above formulas and definitions correctly, we next provide the detailed evaluation of the Phase I of the XML-to-relation conversion represented in Figure 3. It is worth recalling that this simple example has been used in several contexts dealing with pivot and unpivot operations, and we therefore believe it helps also to proportion our approach to other approaches proposed for these operations. This sample conversion is based on the XML relation representation of the StockTicker XML source given in Figure 2. As we recall from Section 4.1, the related target schema is Result(Time, Company(Price)) (STS1 for short), the predicate r_name(STS1) = Result, and the function e_set(STS1) = {Time, Company(Price)}. The (partial) evaluation tree is given in Figure 5.

The evaluation starts from the root node (with the index ⟨1⟩), and hence the first function call is extract&restructure(StockTicker, Result(Time, Company(Price)), {⟨1⟩}). This matches CASE 1, since the cardinality of *I-Set* (i.e., {⟨1⟩}) is one and the associated data item is a non-leaf node. Next, we proceed to consider the tuple elements that are the children to the root node (i.e., *I-Set* = {⟨1,1⟩, ⟨1,2⟩, ⟨1,3⟩}). This matches CASE 6, since the cardinality of *I-Set* is greater than one and the function similar(*I-Set*) returns true (see, Formula 3.5). Now, the evaluation splits into three branches, one for each tuple element. Let us call this Branching Point 1.

For the first tuple element, the function call is now extract&restructure(StockTicker, Result(Time, Company(Price)), {⟨1,1⟩}). Here, we face again CASE 1, because the tuple element related to the index ⟨1,1⟩ is a non-leaf node. We therefore proceed to consider its children (i.e., *I-Set* = {⟨1,1,1⟩, ⟨1,1,2⟩, ⟨1,1,3⟩}). In this case, the cardinality of *I-Set* is greater than one but the function similar(*I-*

Set) returns false. It means that we have CASE 7 at hand. The evaluation splits again into three branches that are traversed in the order given by the function next_sibling (defined in Formula 3.6); we refer to this as Branching Point 1.1.

For the Time element related to the index ⟨1,1,1⟩, the evaluation matches CASE 3, since the cardinality of *I-Set* (i.e., {⟨1,1,1⟩}) is one, the associated data item is a leaf node, and the data item's name is a member of the set {Time, Company(Price)} yielded by the function e_set(STS1), and it has a corresponding data item name in the XML source (basic(Time) = true). In this case, the function extract&restructure(StockTicker, Result(Time, Company(Price)), {⟨1,1,1⟩} yields the set {('Time', 'e', ⟨1⟩), ('0900', 'v', ⟨1,1⟩)}, which we get by extracting and re-indexing the original Time element and its value. The evaluation reverts to Branching Point 1.1.

Next, we consider the Company element related to the index ⟨1,1,2⟩. In this case, the cardinality of *I-Set* (i.e., {⟨1,1,2⟩}) is one, the associated data item is a leaf node, and the data item name Company matches the pivot expression Company(Price) in the target schema. In other words, we have CASE 4. Now, the function call extract&restructure(StockTicker, Result(Time, Company(Price)), {⟨1,1,2⟩} yields the tuple {('MSFT', 'e', ⟨1⟩), ('27.20', 'v', ⟨1,1⟩)}. This tuple is formed by turning the value of the Company element (MSFT) into an element (('MSFT', 'e', ⟨1⟩)) and making the value of the Price element closest to the Company element as the value of this newly formed element. The evaluation reverts again to Branching Point 1.1.

In this branch, we have still the element Price related to the index ⟨1,1,3⟩ to consider. It matches CASE 2, since the cardinality of *I-Set* (i.e., {⟨1,1,3⟩}) is one and the associated data item is a leaf node but the last condition, the associated data item name is a part of the set e_set(STS1) = {Time, Company(Price)}, fails. In this case, the function call extract&restructure(StockTicker, Result(Time, Company(Price)), {⟨1,1,3⟩} yields an empty set, and the evaluation reverts to Branching Point 1.1.

When all the branches from Branching Point 1.1 have now been traversed, we concatenate the results obtained through them by the operation <> and get the set {(Time, 'e', ⟨1⟩), (0900, 'v', ⟨1,1⟩), (MSFT, 'e', ⟨2⟩), (27.20, 'v', ⟨2,1⟩)}. Next, the evaluation reverts to Branching Point 1, where we have still two branches to traverse. However, since the evaluation in them is analogous to that of Branching Point 1.1 above, we leave their detailed consideration to the reader. It is here sufficient to say that the branch rooted by the index ⟨1,2⟩ yields the set {(Time, 'e', ⟨1⟩), (0900, 'v', ⟨1,1⟩), (IBM, 'e', ⟨2⟩), (120.00, 'v', ⟨2,1⟩)} and the branch rooted by the index ⟨1,3⟩ the set {(Time', 'e', ⟨1⟩), (0905, 'v', ⟨1,1⟩), (MSFT, 'e', ⟨1⟩), (27.30, 'v', ⟨1,1⟩)}. Now, all the branches leading to Branching Point 1 have been traversed, and the evaluation can be completed by unionizing their results. By doing this, we get the set {{(Time, 'e', ⟨1⟩), (0900, 'v', ⟨1,1⟩), (MSFT, 'e', ⟨2⟩), (27.20, 'v', ⟨2,1⟩)}, {(Time, 'e', ⟨1⟩), (0900, 'v', ⟨1,1⟩), (IBM, 'e', ⟨2⟩), (120.00, 'v', ⟨2,1⟩)}, {(Time', 'e', ⟨1⟩), (0905, 'v', ⟨1,1⟩), (MSFT, 'e', ⟨2⟩), (27.30, 'v', ⟨2,1⟩)}}, which is also the intermediate form based on which the final target relation is constructed. Henceforth, we denote the above intermediate form by IF1.

4.3. Phase II: The Construction of the Target Relation from the Intermediate Form

It is impossible to construct a target relation without knowing the exact relational attributes it contains. As explained in Section 4.1, only a part of attribute names can be specified explicitly in the target schema, whereas the attribute names generated by the pivot operation become known only at run-time. We see this as one of the main reasons why its textual expression and combination with other relational operations have proven so difficult. Next, we show that based on our XML relation representation it is possible to resolve which attribute names will be generated based on each pivot expression. This is because we are able to find out all different values of the data item, which are used as attribute names in the target relation. As mentioned above, all the generated attribute names do not have values in all tuples (rows) in the result relation.

The attribute names in the target schema are expressed explicitly or implicitly. One explicit way is to use a source data item name as such as an attribute name. Another way is to give two new attribute names in the context of each unpivot expression. As explained above, the implicit way is related to pivot expressions. The function target_attribute_names(*X-Rel, TS*) gives the set of relational attribute names, which can be inferred from the underlying XML source *X-Rel* and target schema *TS*. It is defined as follows.

$$
\begin{aligned}
&\text{target\_attribute\_names}(\textit{X-Rel}, TS)\\
&= \{expr \mid expr \in \text{e\_set}(TS)\text{: basic}(expr) = \text{true}\} \text{ // } \textbf{CASE 1}\\
&\cup \bigcup_{expr \in UP} \text{new\_attributes}(expr) \text{ where } UP = \{expr \mid expr \in \text{e\_set}(TS)\text{: unpivot}(expr)\\
&\qquad = \text{true}\} \text{ // } \textbf{CASE 2}\\
&\cup \{value \mid expr \in \text{e\_set}(TS)\text{: pivot}(expr) = \text{true} \wedge (\text{gen}(expr), \_, index)\\
&\qquad \in \textit{X-Rel} \wedge (value, \text{'v'}, \langle index \perp 1\rangle) \in \textit{X-Rel}\} \text{ // } \textbf{CASE 3}
\end{aligned}
\tag{4.2}
$$

In CASE 1 above, the set contains those attribute names which appear as the identical data item names in the XML source. In CASE 2, each unpivot expression gives two new attribute names which do not appear in the XML source (the predicate new_attributes was introduced in Section 4.1). In CASE 3, the set contains the attribute names that will be generated at run-time based on all the pivot expressions in the target schema *TS*.

In the context of Example 1, the function target_attribute_names gives the set {Time, MSFT, IBM}; i.e. CASE 1 and CASE 3 are applied. Once the set of attributes in the target relation has been obtained, we need to construct tuples so that one relational tuple is constructed from one XML relation in the intermediate form. This means essentially two things. First, we have to represent attributes as name–value pairs in a relational tuple (see, Section 3.5), whereas their connection is represented based on indices in the corresponding XML relation. Second, it is possible that an XML relation does not contain information for specific attributes, in which case the attribute

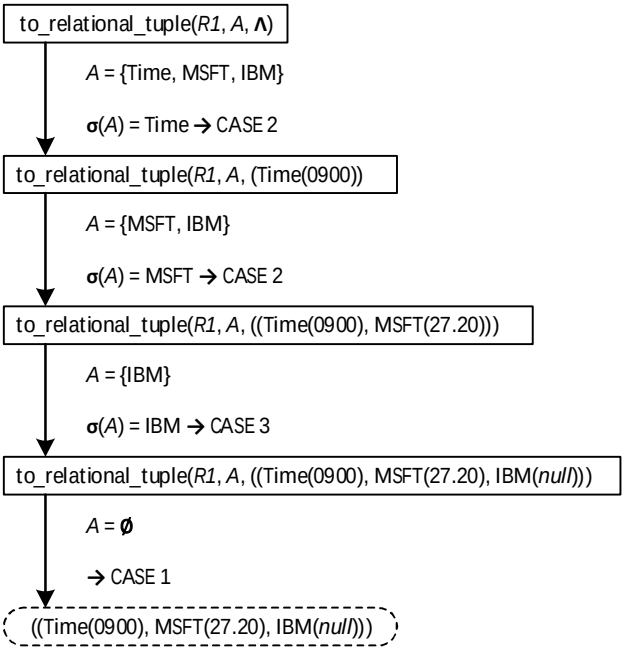


Figure 6. The evaluation tree related to Phase II of Example 1.

values are assigned to *null,* indicating the missing information. The function to_relational_tuple(*X-Rel, A, tuple*) instantiates recursively from the XML relation *X-Rel* each relational attribute in *A* one by one. The argument *tuple* expands as the instantiation of attributes proceeds. When the function is initially invoked, the argument *tuple* is the empty tuple Λ and, in the end, all attributes in *A* have been instantiated. Because we represent a relational tuple based on attribute name–attribute value pairs, the order among these pairs is unimportant. Therefore, in the construction of a tuple we apply the selection function σ to obtain undeterministically one attribute name from the set *A*. Based on the notations and functions introduced above, the function to_relational_tuple is defined as follows.

$$\text{to\_relational\_tuple}(\textit{X-Rel}, A, tuple) = \begin{cases} tuple, & \text{if } A = \emptyset \ \text{// CASE 1} \\ \text{to\_relational\_tuple}(\textit{X-Rel}, A - \{\sigma(A)\}, \text{concatenate}(a(v), tuple) & \\ \quad \text{where } \sigma(A) = a, & \\ \text{if } A \neq \emptyset \wedge (a,' \text{e}', index) \in \textit{X-Rel} \wedge (v,'\text{v}', \langle index \perp 1 \rangle) \in \textit{X-Rel} \ \text{// CASE 2} & \\ \text{to\_relational\_tuple}(\textit{X-Rel}, A - \{\sigma(A)\}, \text{concatenate}(a(null), tuple) & \\ \quad \text{where } \sigma(A) = a, & \\ \text{if } A \neq \emptyset \wedge \nexists (a,' \text{e}', index) \in \textit{X-Rel} \ \text{// CASE 3} & \end{cases} \quad (4.3)$$

Let us now consider how a single relational tuple is evaluated from the XML relation {(Time, 'e', ⟨1⟩), (0900, 'v', ⟨1,1⟩), (MSFT, 'e', ⟨2⟩), (27.20, 'v', ⟨2,1⟩)} of the intermediate form IF1 constructed in Example 1. In this context, *A* = {Time, MSFT, IBM}. The evaluation tree related to the the above

XML relation, denoted by R1, is shown in Figure 6. The remaining two XML relations in the intermediate form IF1 are treated in an analogous way.

By applying the functions target_attribute_names and to_relational_tuple in the context of the result of Phase I (i.e., the intermediate form), we can define Phase II of our conversion operation. It constructs the target relation organized according to the given target schema *TS* from the intermediate form *IF,* which, in turn, has been constructed from the underlying XML source *X-Rel* in Phase I. Phase II is implemented by the function intermediate_to_target(X-Rel, IF, TS) and is defined as follows

intermediate_to_target($X\text{-}Rel, IF, TS$) = $\cup_{r \in IF}$\{to_relational_tuple(r, A, Λ)\} (4.4)

where A = target_attribute_names($X\text{-}Rel, TS$).

In the context of Example 1, Phase II can be implemented by the function intermediate_to_target(StockTicker, IF1, Result(Time, Company(Price))). The evaluation of this function gives the set

{(Time(0900), MSFT(27.20), IBM(*null*)),
(Time(0900), MSFT(*null*), IBM(120.00)),
(Time(0905), MSFT(27.30), IBM(*null*))}.

This set expresses the tuples of which the sample target relation Result (= r_set(Result(Time, Company(Price)))) consists; i.e., the desired target relation has been constructed. The tabular visualization of this target relation was given in Figure 3.

The combination of Phases I and II contains the processing mechanism that is needed to construct a target relation from the information in the XML source at hand. This processing mechanism includes complex data restructuring, such as pivoting and unpivoting, which are beyond the conventional relational operations. The whole conversion process based on the relation representation of an XML source denoted by *X-Rel* and a given target schema denoted by *TS* can be implemented by the function conversion(*X-Rel, TS*), which is defined as follows

conversion($X\text{-}Rel, TS$) = intermediate_to_target($X\text{-}Rel, IF, TS$) (4.5)

where *IF* = extract&restrusture(*X-Rel, TS,* {⟨1⟩}).

5. Sample Evaluation

In the previous section, we defined our conversion tool and demonstrated its use in the context of a simple and well-structured XML source. We borrowed our example from [HPTa-08] where it was used to demonstrate pivot and unpivot operations in relational databases. In this paper, we represented the source information as XML. We believe that this compact example suffices to show the processing principles of our conversion tool. However, in this section we provide a

more comprehensive example that further illuminates the capability of our tool to treat both pivot and unpivot operations at the same time.

Example 2. Appendix A shows the IndustryStatistics XML source containing information on industry sectors in the USA and in Finland (denoted by FI). The elements names Textile, Forest, and IT stand for industry sectors, whereas their values express the profit produced by them (in an unspecified time period). The element Number expresses the total number of enterprises belonging to a specific sector. The costs are given per personnel category, which are Mgmt (for Management), Admin (for Administration), Worker, and Expert.

Although the sample XML source is an artificial one, it contains aspects typical of Open Data. For example, the number of enterprises is expressed only for the Textile sector in the USA and for the Forest sector in Finland. Further, different industry sectors have different personnel categories. Likewise, in the source the USA has tree industry sectors (Textile, Forest, and IT), while Finland has only two (Forest and IT). In other words, the fragments of the IndustryStatistics XML source do not contain regular information.

Let us now assume that we want to form such a relation (table) expressing the profits of industry sectors, costs per personnel category, and the number of enterprises belonging to a specific industry sector in a specific country. In our conversion tool, this relation can be constructed by applying the function conversion(IndustryStatistics, Result(Name, Sector([Textile, Forest, IT], Profit), Number, Type(Cost)))). In the context of the given target schema (STS2 for short), r_name(STS2) = Result and e_set(STS2) = {Name, Sector([Textile, Forest, IT], Profit), Number, Type(Cost)}. The evaluation of the above function starts by constructing first the intermediate form based on the function extract&restructure, defined in Formula 4.1. Its (partial) evaluation tree is shown in Appendix B. The evaluation goes as follows.

The first function call, extract&restructure(IndustryStatistics, STS2, {⟨1⟩}), matches CASE 1, since *I-Set* consists of the index of the root node which is a non-leaf node. The evaluation next proceeds to consider the children of the root node, i.e., the Country elements with the indices ⟨1,1⟩ and ⟨1,2⟩. This time we face CASE 6, since the cardinality of *I-Set* (= {⟨1,1⟩, ⟨1,2⟩}) is greater than one and the function similar({⟨1,1⟩, ⟨1,2⟩}) yields true. The evaluation splits now into two branches. Since we will later return to this point of evaluation, we call it Branching Point 1.

Let us take a closer look at the branch rooted by (Country, 'e', ⟨1,1⟩). Here, the evaluation first matches CASE 1 as *I-Set* = {⟨1,1⟩} and (Country, 'e', ⟨1,1⟩) is a non-leaf node, and we proceed to its children (Name, 'a', ⟨1,1,1⟩) and (Sectors, 'e', ⟨1,1,2⟩). The first child matches CASE 3, and the XML relation {(Name, 'e', ⟨1⟩), (USA, 'v', ⟨1,1⟩)} is constructed. The second child, on the other hand, matches CASE 1 and the evaluation proceeds to its children, the three SectorInfo elements with the indices ⟨1,1,2,1⟩, ⟨1,1,2,2⟩, and ⟨1,1,2,3⟩. Here, we face CASE 6. Since the evaluation now splits into three branches and we will later return to this point, we call this Branching Point 1.1.

Let us now consider the evaluation related to the SectorInfo element with the index ⟨1,1,2,1⟩. The element matches CASE 1, and the evaluation proceeds to its children. Its first child, (Textile, 'e', ⟨1,1,2,1,1⟩), matches CASE 5, since the cardinality of *I-Set* is one, the element at hand is a leaf node, and the predicate to_be_unpivot(Textile) is true. In this case, the predicate a-name(Textile) yields Sector and the predicate v-name(Textile) yields Profit. Once these values are assigned to the relevant places in Formula 4.1 (CASE 5), we are able to construct the XML relation {(Sector, 'e', ⟨1⟩), (Textile, 'v', ⟨1,1⟩), (Profit, 'e', ⟨2⟩), (27, 'v', ⟨2,1⟩)} by concatenating the XML relations {(Sector, 'e', ⟨1⟩), (Textile, 'v', ⟨1,1⟩)} and {(Profit, 'e', ⟨1⟩), (27, 'v', ⟨1,1⟩)}. Next, the evaluation backtracks to consider the remaining children of the SectorInfo element with the index ⟨1,1,2,1⟩.

Next, it is the turn of the element (Number, 'e', ⟨1,1,2,1,2⟩) that matches CASE 3, and the XML relation {(Number, 'e', ⟨1⟩), (120, 'v', ⟨1,1⟩)} is constructed. We then proceed to (Categories, 'e', ⟨1,1,2,1,3⟩), which satisfies CASE 1, and progress to its children, the three Category elements with the indices ⟨1,1,2,1,3,1⟩, ⟨1,1,2,1,3,2⟩, and ⟨1,1,2,1,3,3⟩. They each match CASE 1, and the evaluation moves on to their respective children. Let us now consider the evaluation related to the children of the Category element with the index ⟨1,1,2,1,3,1⟩. Its first child, the element (Type, 'e', ⟨1,1,2,1,3,1,1⟩), matches CASE 4, as the cardinality of *I-Set* is one, the element at hand is a leaf node, and it participates in a pivot expression in the target schema *TS*, so that its value is used as an attribute name in the result relation. In other words, the value of the Type element (Mgmt) is pivoted as an attribute name in the target relational schema, and the value of the Cost element that is most closely associated with it is assigned as its value (in this case, the value 3 of the Cost element with the index ⟨1,1,2,1,3,1,2⟩). As a result, the XML relation {(Mgmt, 'e', ⟨1⟩), (3, 'v', ⟨1,1⟩)} is constructed. The evaluation next proceeds to element (Cost, 'e', ⟨1,1,2,1,3,1,2⟩). The element matches CASE 2, meaning that it does not appear at the target relational schema (although its value occurs in the result relation), and an empty XML relation is returned. Now the evaluation reverts to consider the remaining two children. Their evaluation is analogous to the evaluation related to the Category element with the index ⟨1,1,2,1,3,1⟩, resulting in the construction of the XML relations {(Admin, 'e', ⟨1⟩), (5, 'v', ⟨1,1⟩)} and {(Worker, 'e', ⟨1⟩), (25, 'v', ⟨1,1⟩)}.

The evaluation related to the SectorInfo element with the index ⟨1,1,2,1⟩ is now complete, and we backtrack to Branching Point 1.1. As a result, we have constructed a set consisting of three XML relations: {{(Sector, 'e', ⟨1⟩), (Textile, 'v', ⟨1,1⟩), (Profit, 'e', ⟨2⟩), (27, 'v', ⟨2,1⟩), (Number, 'e', ⟨3⟩), (120, 'v', ⟨3,1⟩), (Mgmt, 'e', ⟨4⟩), (3, 'v', ⟨4,1⟩)}}, {(Sector, 'e', ⟨1⟩), (Textile, 'v', ⟨1,1⟩), (Profit, 'e', ⟨2⟩), (27, 'v', ⟨2,1⟩), (Number, 'e', ⟨3⟩), (120, 'v', ⟨3,1⟩), (Admin, 'e', ⟨4⟩), (5, 'v', ⟨4,1⟩)}},{(Sector, 'e', ⟨1⟩), (Textile, 'v', ⟨1,1⟩), (Profit, 'e', ⟨2⟩), (27, 'v', ⟨2,1⟩), (Number, 'e', ⟨3⟩), (120, 'v', ⟨3,1⟩), (Worker, 'e', ⟨4⟩), (25, 'v', ⟨4,1⟩)}}. The evaluation related to the remaining two branches, rooted by (SectorInfo, 'e', ⟨1,1,2,2⟩) and (SectorInfo, 'e', ⟨1,1,2,3⟩) is analogous to the case above. The evaluation related to the index ⟨1,1,2,2⟩ produces the set:

{{(Sector, 'e', ⟨1⟩), (Forest, 'v', ⟨1,1⟩), (Profit, 'e', ⟨2⟩), (20, 'v', ⟨2,1⟩), (Mgmt, 'e', ⟨3⟩), (2, 'v', ⟨3,1⟩)}},

{(Sector, ’e’, ⟨1⟩), (Forest, ’v’, ⟨1,1⟩), (Profit, ’e’, ⟨2⟩), (20, ’v’, ⟨2,1⟩), (Worker, ’e’, ⟨3⟩), (25, ’v’, ⟨3,1⟩)}}}.

Similarly, the evaluation of the index ⟨1,1,2,3⟩ produces the set:

{{(Sector, ’e’, ⟨1⟩), (IT, ’v’, ⟨1,1⟩), (Profit, ’e’, ⟨2⟩), (200, ’v’, ⟨2,1⟩), (Mgmt, ’e’, ⟨3⟩), (10, ’v’, ⟨3,1⟩)}},
{(Sector, ’e’, ⟨1⟩), (IT, ’v’, ⟨1,1⟩), (Profit, ’e’, ⟨2⟩), (200, ’v’, ⟨2,1⟩), (Admin, ’e’, ⟨3⟩), (30, ’v’, ⟨3,1⟩)}},
{(Sector, ’e’, ⟨1⟩), (IT, ’v’, ⟨1,1⟩), (Profit, ’e’, ⟨2⟩), (200, ’v’, ⟨2,1⟩), (Expert, ’e’, ⟨3⟩), (50, ’v’, ⟨3,1⟩)}}}.
The evaluation related to Branching Point 1.1 is complete when we unionize the above three sets. The evaluation then reverts to Branching Point 1 with the concatenation of the XML relation {(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩)} with each member of the above set (they are XML relations). For example, the concatenation of the XML relation {(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩)} with the XML relation {(Sector, ’e’, ⟨1⟩), (Textile, ’v’, ⟨1,1⟩), (Profit, ’e’, ⟨2⟩), (27, ’v’, ⟨2,1⟩), {(Number, ’e’, ⟨3⟩), (120, ’v’, ⟨3,1⟩), (Mgmt, ’e’, ⟨4⟩), (3, ’v’, ⟨4,1⟩)} produces the following new XML relation:

{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (Textile, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (27, ’v’, ⟨3,1⟩), (Number, ’e’, ⟨4⟩), (120, ’v’, ⟨4,1⟩), (Mgmt, ’e’, ⟨5⟩), (3, ’v’, ⟨5,1⟩)}.

Now, the evaluation related to the branch rooted by (Country, ’e’, ⟨1,1⟩) is complete. We have still the branch rooted by (Country, ’e’, ⟨1,2⟩) to consider, but since its evaluation is analogous to the above we leave it to the reader. The evaluation of the function extract&restructure(IndustryStatistics, STS2, {⟨1⟩}) produces the set consisting of the following XML relations:

{{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (Textile, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (27, ’v’, ⟨3,1⟩), (Number, ’e’, ⟨4⟩), (120, ’v’, ⟨4,1⟩), (Mgmt, ’e’, ⟨5⟩), (3, ’v’, ⟨5,1⟩)},
{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (Textile, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (27, ’v’, ⟨3,1⟩), (Number, ’e’, ⟨4⟩), (120, ’v’, ⟨4,1⟩), (Admin, ’e’, ⟨5⟩), (5, ’v’, ⟨5,1⟩)},
{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (Textile, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (27, ’v’, ⟨3,1⟩), (Number, ’e’, ⟨4⟩), (120, ’v’, ⟨4,1⟩), (Worker, ’e’, ⟨5⟩), (25, ’v’, ⟨5,1⟩)},
{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (Forest, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (20, ’v’, ⟨3,1⟩), (Mgmt, ’e’, ⟨4⟩), (2, ’v’, ⟨4,1⟩)},
{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (Forest, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (20, ’v’, ⟨3,1⟩), (Worker, ’e’, ⟨4⟩), (25, ’v’, ⟨4,1⟩)},
{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (IT, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (200, ’v’, ⟨3,1⟩), (Mgmt, ’e’, ⟨4⟩), (10, ’v’, ⟨4,1⟩)},
{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (IT, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (200, ’v’, ⟨3,1⟩), (Admin, ’e’, ⟨4⟩), (30, ’v’, ⟨4,1⟩)},
{(Name, ’e’, ⟨1⟩), (USA, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (IT, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (200, ’v’, ⟨3,1⟩), (Expert, ’e’, ⟨4⟩), (50, ’v’, ⟨4,1⟩)},
{(Name, ’e’, ⟨1⟩), (FI, ’v’, ⟨1,1⟩), (Sector, ’e’, ⟨2⟩), (Forest, ’v’, ⟨2,1⟩), (Profit, ’e’, ⟨3⟩), (22, ’v’, ⟨3,1⟩), (Number, ’e’, ⟨4⟩), (4, ’v’, ⟨4,1⟩), (Mgmt, ’e’, ⟨5⟩), (3, ’v’, ⟨5,1⟩)},

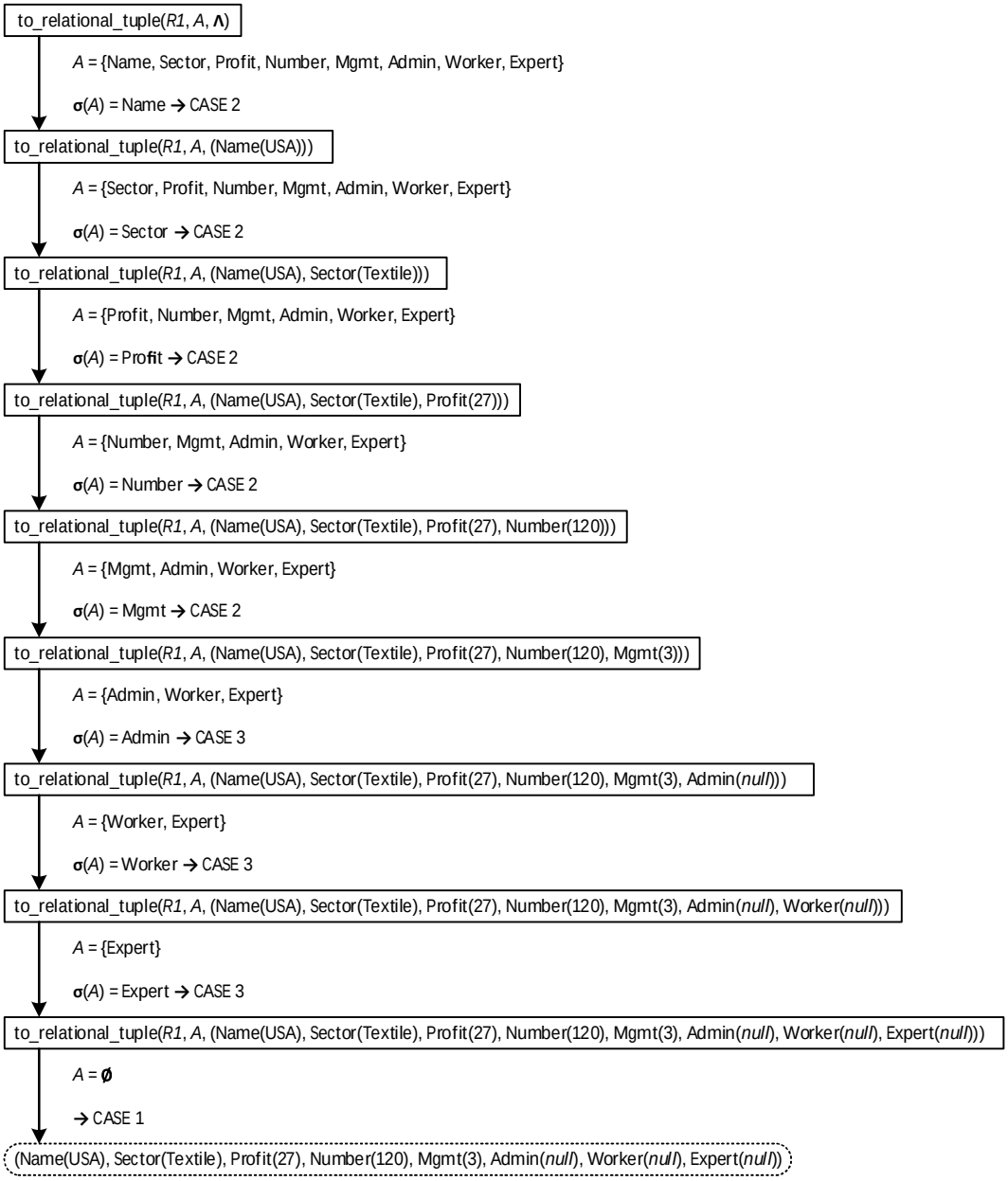


Figure 7. An evaluation tree related to Phase II of Example 2.

{(Name, 'e', ⟨1⟩), (Fl, 'v', ⟨1,1⟩), (Sector, 'e', ⟨2⟩), (Forest, 'v', ⟨2,1⟩), (Profit, 'e', ⟨3⟩), (22, 'v', ⟨3,1⟩), (Number, 'e', ⟨4⟩), (4, 'v', ⟨4,1⟩), (Worker, 'e', ⟨5⟩), (30, 'v', ⟨5,1⟩)},
{(Name, 'e', ⟨1⟩), (Fl, 'v', ⟨1,1⟩), (Sector, 'e', ⟨2⟩), (IT, 'v', ⟨2,1⟩), (Profit, 'e', ⟨3⟩), (80, 'v', ⟨3,1⟩), (Mgmt, 'e', ⟨4⟩), (8, 'v', ⟨4,1⟩)},
{(Name, 'e', ⟨1⟩), (Fl, 'v', ⟨1,1⟩), (Sector, 'e', ⟨2⟩), (IT, 'v', ⟨2,1⟩), (Profit, 'e', ⟨3⟩), (80, 'v', ⟨3,1⟩), (Admin, 'e', ⟨4⟩), (15, 'v', ⟨4,1⟩)},
{(Name, 'e', ⟨1⟩), (Fl, 'v', ⟨1,1⟩), (Sector, 'e', ⟨2⟩), (IT, 'v', ⟨2,1⟩), (Profit, 'e', ⟨3⟩), (80, 'v', ⟨3,1⟩), (Expert, 'e', ⟨4⟩), (20, 'v', ⟨4,1⟩)}}.

The set consisting of the above XML relations is the intermediate form, denoted by IF2. This is also the output of Phase I of our conversion operation.

Phase II constructs the desired target relation from the intermediate form IF2 by the function intermediate_to_target(IndustryStatistics, IF2, STS2). It, in turn, applies the function *to_relational_tuple* to each XML relation in the intermediate form and unionizes the resulting tuples. Figure 7 shows the evaluation tree for the XML relation {(Name, 'e', ⟨1⟩), (USA, 'v', ⟨1,1⟩), (Sector, 'e', ⟨2⟩), (Textile, 'v', ⟨2,1⟩), (Profit, 'e', ⟨3⟩), (27, 'v', ⟨3,1⟩), (Number, 'e', ⟨4⟩), (120, 'v', ⟨4,1⟩), (Mgmt, 'e', ⟨5⟩), (3, 'v', ⟨5,1⟩)}, denoted by R1. In our example, the auxiliary function target_attribute_names produces the set {Name, Sector, Profit, Number, Mgmt, Admin, Worker, Expert}, denoted by A.

The remaining XML relations in the intermediate form IF2 are evaluated in an analogous fashion. The evaluation of all the XML relations in IF2 produces the set:

{(Name(USA), Sector(Textile), Profit(27), Number(120), Mgmt(3), Admin(*null*), Worker(*null*), Expert(*null*)),
(Name(USA), Sector(Textile), Profit(27), Number(120), Mgmt(*null*), Admin(5), Worker(*null*), Expert(*null*)),
(Name(USA), Sector(Textile), Profit(27), Number(120), Mgmt(*null*), Admin(*null*), Worker(25), Expert(*null*)),
(Name(USA), Sector(Forest), Profit(20), Number(*null*), Mgmt(2), Admin(*null*), Worker(*null*), Expert(*null*)),
(Name(USA), Sector(Forest), Profit(20), Number(*null*), Mgmt(*null*), Admin(*null*), Worker(25), Expert(*null*)),
(Name(USA), Sector(IT), Profit(200), Number(*null*), Mgmt(*null*), Admin(*null*), Worker(*null*), Expert(*null*)),
(Name(USA), Sector(IT), Profit(200), Number(*null*), Mgmt(10), Admin(*null*), Worker(*null*), Expert(*null*)),
(Name(USA), Sector(IT), Profit(200), Number(*null*), Mgmt(*null*), Admin(30), Worker(*null*), Expert(50)),
(Name(FI), Sector(Forest), Profit(22), Number(4), Mgmt(3), Admin(*null*), Worker(*null*), Expert(*null*)),
(Name(FI), Sector(Forest), Profit(22), Number(4), Mgmt(*null*), Admin(*null*), Worker(*null*), Expert(30)),
(Name(FI), Sector(IT), Profit(80), Number(*null*), Mgmt(8), Admin(*null*), Worker(*null*), Expert(*null*)),
(Name(FI), Sector(IT), Profit(80), Number(*null*), Mgmt(*null*), Admin(15), Worker(*null*), Expert(*null*)),
(Name(FI), Sector(IT), Profit(80), Number(*null*), Mgmt(*null*), Admin(*null*), Worker(*null*), Expert(20))}.

Figure 8 represents the result of Example 2 as a typical tabular visualization. This table can be made even more compact through some post-processing (e.g., representing the contents of several rows in one row) in relational database environments.

6. Discussion

During the last years, it has been noticed in enterprises and other organizations that it is not sufficient to analyze only Closed Data, i.e., data in databases controlled by them. Due to the

Result

Name	Sector	Profit	Number	Mgmt	Admin	Worker	Expert
USA	Textile	27	120	3	*null*	*null*	*null*
USA	Textile	27	120	*null*	5	*Null*	*null*
USA	Textile	27	120	*null*	*null*	25	*null*
USA	Forest	20	*null*	2	*null*	*null*	*null*
USA	Forest	20	*null*	*null*	*null*	25	*null*
USA	IT	200	*null*	10	*null*	*null*	*null*
USA	IT	200	*null*	*null*	30	*null*	*null*
USA	IT	200	*null*	*null*	*null*	*null*	50
FI	Forest	22	4	3	*null*	*null*	*null*
FI	Forest	22	4	*null*	*null*	30	*null*
FI	IT	80	*null*	8	*null*	*null*	*null*
FI	IT	80	*null*	*null*	15	*null*	*null*
FI	IT	80	*null*	*null*	*null*	*null*	20

Figure 8. Tabular visualization of the result of Example 2.

market situations in different countries/areas, competitors, consumer sentiment etc.). By combining this kind of information with data internal to the enterprise, business analytics can be enriched considerably. We see that the main reason for the lack of tools capable for analyzing Open and Closed Data together is that Open Data and Closed Data are based on different data formats. In this paper, we introduce an approach and give a conversion tool in terms of which Open Data of interest can be transferred to a part of Closed Data so that all data are represented based on the same data format used in the closed environment. After conversion, it is possible through tools (e.g. query languages) available in the closed environment to organize underlying data into such structures that support subsequent data analysis. The actual data analysis is beyond this paper.

There are several different data formats both for Open Data and for Closed Data. Our starting point and limitation for the developed conversion tool is that Open Data are assumed to be based on XML and Closed Data on the relational model. As discussed above, many data sources in Open Data are either XML-based or easily convertible into XML whereas the majority of enterprises is used relational databases to store and manipulate internal data. We believe that the developed conversion tool can be utilized in many practical analyzing cases related to combination of Open and Closed Data because many analytics tools presuppose that the underlying data have been organized relationally. The use of our conversion tool presupposes that the user knows XML data sources to be converted. It is not clear that the user masters the data sources in Open Data intended for ad hoc analytics similarly with data sources in Closed Data. The idea of the novel data management trend, called dataspaces, is to increase incrementally the user's knowledge about the underlying data that are beforehand unknown to the user. In [NäNi-12, MNNK-14], we have developed an XML-based dataspace system, which, among others, helps the user to find out the contents, structures and semantics of beforehand unknown XML sources. Thus, we can assume that, at least after the use of our XML-based dataspace system, the user knows in detail the XML data sources to be converted. As explained above, in our conversion tool the user uses the names of the XML data items (i.e., attribute or element names) as the attribute names of the target relation. Therefore, the user must know in

globalization, there is often a need to combine Closed Data internal to a specific organization with Open Data that is freely available on the Web. For example, one of the greatest shortcomings in business intelligence is that there are no tools for ad hoc analytics based on data sources in Open Data and Closed Data [VTBL-13, ETBL-13]. From the viewpoint of a specific enterprise, Open Data may contain critical information (e.g. on changes in trends, detail the information contents and semantics of XML sources.

It is typical of the data exchange approach that it is based on schema mappings between the underlying source and target schema. In it, the target data are materialized by realizing the corresponding changes at the instance level. In the conventional data exchange approach, the source and target schemas are based on the same data model. From the viewpoint of our contribution, the closest work of the data exchange approach has been represented in [PVMH+02, HPTa-08], in which restructuring between flat and regularly structured hierarchical relations can be specified based on the nested relational model. In our conversion approach, it is however important that the platform of stored data also changes, i.e., a textual XML data format is transformed into a tabular (relational) format. Unlike in the data exchange approach, we do not require that the XML source to be converted have an attached schema (e.g. DTD). In the paper, we show that by utilizing the self-description property of semi-structured XML data in the context of the XML relation representation it is possible to analyze structural relationships among XML data without any separate schema. Our conversion tool analyzes automatically, on behalf of the user, the structural relationships among data of the XML source at hand. Therefore, the use of our conversion tool is very declarative. The data restructuring facility is also an essential part of the conversion tool. In the paper, we show that the constructor algebra developed for manipulating the XML relation representation is expressive enough to perform demanding data restructuring including data-to-metadata and metadata-to-data translations.

In our approach, the XML relation representation has an essential role as an intermediate form in constructing the desired target relation from a textual XML source. This is due to the following facts. First, any textual XML source can be converted unambiguously, i.e., without losing any information, into the XML relation representation that is structurally compatible with the conventional relation in spite of their fundamental differences. Among others, this means that we can utilize the efficient storing methods of RDBMSs as such although XML relations have to be manipulated by the constructor algebra instead of the relational algebra intended to manipulate conventional relations. Second, in the paper we show how information based on the XML relation representation can be transformed into the conventional relation representation.

We have developed an XML-based dataspace system [NäNi-12] based on the XML relation representation. Its prototype has been implemented on top of a PostgreSQL relational database system. We have implemented the bidirectional conversion tool between the textual and XML relation representations, and by using this tool we convert any textual XML source into its corresponding XML relation which is stored under the PostgreSQL relational database system. In our prototype, the constructor algebra has been implemented as user-defined SQL functions

written in C. This algebra has been applied in implementing our RXQL query language in [NMNi-11], which has a central role in manipulating the heterogeneity factors among XML sources. It is possible that the same data in an Open XML source and in the available relational database are represented in a different way. For example, the names of persons can expressed in the order last name–first name in an Open XML source, whereas they are represented in the opposite order in the underlying relational database. Likewise, the values in Open Data and Closed Data can be based on the different units of measurement (e.g. monetary units). In the conversion tool specified in this paper, we did not pay any attention to that the values can be represented in a heterogeneous way among Open and Closed Data. Of course, we have to remove this kind of heterogeneity before data can be utilized in the closed environment. In [MNNK-14], we have implemented a dataspace system with a user-friendly interface, which also contains visual primitives for removing heterogeneity related to representations of values. In future, our aim is to implement and extend the conversion tool of this paper so that by utilizing our dataspace system it is also capable to represent values of data in a uniform way with Closed Data.

7. Conclusions

Combining XML-based Open Data and relational Closed Data is needed in many contexts. In this paper, we introduce and specify a conversion tool that is able to construct a relation of a relational database (Closed Data) from desired data items of an XML data source in Open Data. A user-friendly interface of the tool presupposes a high degree of automation, which, in turn, requires a great analyzing power. In the paper, we show that the XML relation representation developed by us affords the possibility of analyzing contents and structural relationships among XML data sources without user interactions. We also show how the XML relation representation can be used as an intermediate form in changing the textual XML format into the relational (tabular) format. Our conversion tool has also to have a great restructuring power because, in addition to restructuring of hierarchical XML data to flat relational data, it is often necessary to transfer data at the instance level to data at the schema level (so called data-to-metadata translations) and vice versa (so called metadata-to-data translations). In our tool this kind of translations can be made through *pivot* and *unpivot* restructuring operations. We show that the needed great restructuring power can be implemented by the constructor algebra attached to the XML relation representation.

References

[ArLi-08] Marcelo Arenas, Leonid Libkin. XML data exchange: Consistency and query answering. Journal of the ACM 2008; 55(2).

[Barc-09] Pablo Barceló. Logical foundations of relational data exchange. SIGMOD Record 2009; 38(1): 49-58.

[BETL-12] Katrin Braunschweig, Julian Eberius, Maik Thiele, Wolfgang Lehner. OPEN: Enabling non-expert users to extract, integrate, and analyze Open Data. Datenbank-Spektrum 2012; 12(2): 121-130.

[BPEF+10] Khalid Belhajjame, Norman W. Paton, Suzanne M. Embury, Alvaro A. A. Fernandes, Cornelia Hedeler. Feedback-based annotation, selection and refinement of schema mappings for dataspaces. In: Proceedings of the 13th International Conference on Extending Database Technology, 2010: 573-584.

[CCSi-00] Vassilis Christophides, Sophie Cluet, Jérôme Siméon. On wrapping query languages and efficient XML integration. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 2000: 141-152.

[CDNa-11] Surajit Chaudhuri, Umeshwar Dayal, Vivek R. Narasayya. An overview of business intelligence technology. Communications of the ACM 2011; 54(8): 88-98.

[EDBT+13] Julian Eberius, Patrick Damme, Katrin Braunschweig, Maik Thiele and Wolfgang Lehner. Publish-Time Data Integration for Open Data Platforms. In: Proceedings of the 2nd International Workshop on Open Data, 2013.

[ETBL-12] Julian Eberius, Maik Thiele, Katrin Braunschweig, Wolfgang Lehner. DrillBeyond: Enabling business analysts to explore the Web of Open Data. Proceedings of the VLDB Endowment 2012; 5(12): 1978-1981.

[ETBL-13] Julian Eberius, Maik Thiele, Katrin Braunschweig, Wolfgang Lehner. DrillBeyond: Open-world SQL queries using Web tables. In: Proceedings of the 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme", 2013: 523-526.

[EWTB+13] Julian Eberius, Christoper Werner, Maik Thiele, Katrin Braunschweig, Lars Dannecker, Wolfgang Lehner. DeExcelerator: A framework for extracting relational data from partially structured documents. In: Proceedings of the 22nd ACM International Conference on Information and Knowledge Management, 2013: 2477-2480.

[FFMR+01] Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, Philip Wadler. The XML Query Algebra. Available at http://www.w3.org/TR/query-algebra/ (2001; accessed October 31, 2014).

[FHMa-05] Michael J. Franklin, Alon Y. Halevy, David Maier. From databases to dataspaces: A new abstraction for information management. SIGMOD Record 2005; 34(4): 27-33.

[FKMP-05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, Lucian Popa. Data exchange: Semantics and query answering. Theoretical Computer Science 2005; 336(1): 89-124.

[FKPo-05] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa. Data exchange: getting to the core. ACM Transactions on Database Systems 2005; 30(1): 174-210.

[FlKo-99] Daniela Florescu, Donald Kossmann. Storing and Querying XML Data using an RDMBS. IEEE Data Engineering Bulletin 1999; 22(3): 27-34.

[HFMa-06] Alon Y. Halevy, Michael J. Franklin, David Maier. Principles of dataspace systems. In: Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 2006: 1-9.

[HJLP+04] Alan Halverson, Vanja Josifovski, Guy M. Lohman, Hamid Pirahesh, Mathias Mörschel, ROX: Relational over XML. In: Proceedings of the 30th International Conference on Very Large Data Bases, 2004: 264-275.

[HNWe-06] Jan Hegewald, Felix Naumann, Melanie Weis. XStruct: Efficient schema extraction from multiple and large XML documents. In: Proceedings of the 22nd International Conference on Data Engineering Workshops, 2006: 81.

[HPTa-08] Mauricio A. Hernández, Paolo Papotti, Wang Chiew Tan. Data exchange with data-metadata translations. Proceedings of the VLDB Endowment 2008; 1(1): 260-273.

[HRGa-10] Matthias Hert, Gerald Reif, Harald Gall. Updating relational data via SPARQL/update. In: Proceedings of the 2010 EDBT/ICDT Workshops, 2010.

[JLST-01] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, Keith Thompson. TAX: A tree algebra for XML. In: Proceedings of the 8th International Workshop on Database Programming Languages, 2001: 149-164.

[Kola-05] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In: Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 2005: 61-75.

[LMGC+09] Fenareti Lampathaki, Spiros Mouzakitis, George Gionis, Yannis Charalabidis, Dimitris Askounis. Business to business interoperability: A current review of XML data integration standards. Computer Standards & Interfaces 2009; 31(6): 1045-1055.

[LPVe-00] Bertram Ludäscher, Yannis Papakonstantinou, Pavel Velikhov. Navigation-driven evaluation of virtual mediated views. In: Proceedings of the 7th International Conference on Extending Database Technology, 2000: 150-165.

[LSSu-96] Laks V. S. Lakshmanan, Fereidoon Sadri, Iyer N. Subramanian. SchemaSQL: A language for interoperability in relational multi-database systems. In: Proceedings of 22th International Conference on Very Large Data Bases, 1996: 239-250.

[MACh-03] Jun-Ki Min, Jae-Yong Ahn, Chin-Wan Chung. Efficient extraction of schemas for XML documents. Information Processing Letters 2003; 85(1): 7-12.

[MHHe-00] Renée J. Miller, Laura M. Haas, Mauricio A. Hernández. Schema mapping as query discovery. In: Proceedings of 26th International Conference on Very Large Data Bases, 2000: 77-88.

[MNNK-14] Katja Moilanen, Timo Niemi, Turkka Näppilä, Mikko Kuru. A visual XML dataspace approach for satisfying ad hoc information needs. Journal of the Association for Information Science and Technology 2014; 17 pages (accepted for publication).

[NCJo-12] Raghunath Nambiar, Ramesh Chitor, Ashok Joshi. Data management: A look back and a look ahead. In: Proceedings of the 1st and 2nd Workshops on Big Data Benchmarking, 2012 (LNCS 8163): 11-19.

[NJJä-14] Timo Niemi, Marko Junkkari, Kalervo Järvelin. Concept-based query language approach to enterprise information systems. Enterprise Information systems 2014; 8(1): 26-66.

[NMMi-11] Turkka Näppilä, Katja Moilanen, Timo Niemi. A query language for selecting, harmonizing, and aggregating heterogeneous XML data. International Journal of Web Information Systems 2011; 7(1): 62-99.

[NNJä-09] Timo Niemi, Turkka Näppilä, Kalervo Järvelin. A relational data harmonization approach to XML. Journal of Information Science 2009; 35(5): 571-601.

[NNPC+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury. ORDPATHs: Insert-friendly XML node labels. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2004: 903-908.

[NäNi-12] Turkka Näppilä, Timo Niemi. An approach for developing a schemaless XML dataspace profiling system. Journal of Information Science 2012; 38(3): 234-257.

[PCSS+04] Shankar Pal, Istvan Cseri, Gideon Schaller, Oliver Seeliger, Leo Giakoumakis, Vasili Vasili Zolotov. Indexing XML data stored in a relational database. In: Proceedings of the 30th International Conference on Very Large Data Bases, 2004: 1134-1145.

[PVMH+02] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, Ronald Fagin. Translating Web data. In: Proceedings of 28th International Conference on Very Large Data Bases, 2002: 598-609.

[RCHa-12] Seán O'Riain, Edward Curry, Andreas Harth. XBRL and open data for global financial ecosystems: a linked data approach. International journal of Accounting Information Systems 2012; 13: 141-162.

[RLBP-10] Vatcharaphun Rajsiri, Jean-Pierre Lorré, Frédérick Bénaben, Hervé Pingaud. Knowledge-based system for collaborative process specification. Computers in Industry 2010; 61(2): 161-175.

[STZH+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In: Proceedings of 25th International Conference on Very Large Data Bases, 1999: 302-314.

[TaGr-10] Andrea Tagarelli, Sergio Greco. Semantic clustering of XML documents. ACM Transactions on Information Systems 2010; 28(1): Article 3.

[TDMS+10] James F. Terwilliger, Lois M. L. Delcambre, David Maier, Jeremy Steinhauer, Scott Britell. Updatable and evolvable transforms for virtual databases. Proceedings of the VLDB Endowment 2010; 3(1): 309-319.

[TZWS-13] Zijing Tan, Liyong Zhang, Wei Wang, Baile Shi. XML data exchange with target constraints. Information Processing and Management 2013; 49(2): 465-483.

[Ullm-88] Jeffrey D. Ullman. Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press: 1988.

[WBBD+03] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Sheng, Sankar Subramanian. Spreadsheets in RDBMS for OLAP. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 2003: 52-63.

[VTBL-13] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, Wolfgang Lehner. Leveraging flexible data management with graph databases. In: Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems, 2013.

[WyRo-05a] Catharine M. Wyss, Edward L. Robertson. A formal characterization of PIVOT/UNPIVOT. In: Proceedings of the 14th ACM International Conference on Information and Knowledge Management, 2005: 602-608.

[WyRo-05b] Catharine M. Wyss, Edward L. Robertson. Relational languages for metadata integration. ACM Transactions on Database Systems 2005; 30(2): 624-660.

Appendix A

```
<IndustryStatistics>
  <Country Name="USA">
    <Sectors>
      <SectorInfo>
        <Textile>27</Textile>
        <Number>120</Number>
        <Categories>
          <Category>
            <Type>Mgmt</Type>
            <Cost>3</Cost>
          </Category>
          <Category>
            <Type>Admin</Type>
            <Cost>5</Cost>
          </Category>
          <Category>
            <Type>Worker</Type>
            <Cost>25</Cost>
          </Category>
        </Categories>
      </SectorInfo>
      <SectorInfo>
        <Forest>20</Forest>
        <Categories>
          <Category>
            <Type>Mgmt</Type>
            <Cost>2</Cost>
          </Category>
          <Category>
            <Type>Worker</Type>
            <Cost>25</Cost>
          </Category>
        </Categories>
      </SectorInfo>
      <SectorInfo>
        <IT>200</IT>
        <Categories>
          <Category>
            <Type>Mgmt</Type>
            <Cost>10</Cost>
          </Category>
          <Category>
            <Type>Admin</Type>
            <Cost>30</Cost>
          </Category>
          <Category>
            <Type>Expert</Type>
            <Cost>50</Cost>
          </Category>
        </Categories>
      </SectorInfo>
    </Sectors>
  </Country>
  <Country Name="FI">
    <Sectors>
      <SectorInfo>
        <Forest>22</Forest>
        <Number>4</Number>
        <Categories>
          <Category>
            <Type>Mgmt</Type>
            <Cost>3</Cost>
          </Category>
          <Category>
            <Type>Worker</Type>
            <Cost>30</Cost>
          </Category>
        </Categories>
      </SectorInfo>
      <SectorInfo>
        <IT>80</IT>
        <Categories>
          <Category>
            <Type>Mgmt</Type>
            <Cost>8</Cost>
          </Category>
          <Category>
            <Type>Admin</Type>
            <Cost>15</Cost>
          </Category>
          <Category>
            <Type>Expert</Type>
            <Cost>20</Cost>
          </Category>
        </Categories>
      </SectorInfo>
    </Sectors>
  </Country>
</IndustryStatistics>
```

Figure A1. The IndustryStatistics sample XML document.

C	T	I
IndustryStatistics	'e'	⟨1⟩
Country	'e'	⟨1,1⟩
Name	'a'	⟨1,1,1⟩
USA	'v'	⟨1,1,1,1⟩
Sectors	'e'	⟨1,1,2⟩
SectorInfo	'e'	⟨1,1,2,1⟩
Textile	'e'	⟨1,1,2,1,1⟩
27	'v'	⟨1,1,2,1,1,1⟩
Number	'e'	⟨1,1,2,1,2⟩
120	'v'	⟨1,1,2,1,2,1⟩
Categories	'e'	⟨1,1,2,1,3⟩
Category	'e'	⟨1,1,2,1,3,1⟩
Type	'e'	⟨1,1,2,1,3,1,1⟩
Mgmt	'v'	⟨1,1,2,1,3,1,1,1⟩
Cost	'e'	⟨1,1,2,1,3,1,2⟩
3	'v'	⟨1,1,2,1,3,1,2,1⟩
Category	'e'	⟨1,1,2,1,3,2⟩
Type	'e'	⟨1,1,2,1,3,2,1⟩
Admin	'v'	⟨1,1,2,1,3,2,1,1⟩
Cost	'e'	⟨1,1,2,1,3,2,2⟩
5	'v'	⟨1,1,2,1,3,2,2,1⟩
Category	'e'	⟨1,1,2,1,3,3⟩
Type	'e'	⟨1,1,2,1,3,3,1⟩
Worker	'v'	⟨1,1,2,1,3,3,1,1⟩
Cost	'e'	⟨1,1,2,1,3,3,2⟩
25	'v'	⟨1,1,2,1,3,3,2,1⟩
SectorInfo	'e'	⟨1,1,2,2⟩
Forest	'e'	⟨1,1,2,2,1⟩
20	'v'	⟨1,1,2,2,1,1⟩
Categories	'e'	⟨1,1,2,2,2⟩
Category	'e'	⟨1,1,2,2,2,1⟩
Type	'e'	⟨1,1,2,2,2,1,1⟩
Mgmt	'v'	⟨1,1,2,2,2,1,1,1⟩
Cost	'e'	⟨1,1,2,2,2,1,2⟩
2	'v'	⟨1,1,2,2,2,1,2,1⟩
Category	'e'	⟨1,1,2,2,2,2⟩
Type	'e'	⟨1,1,2,2,2,2,1⟩
Worker	'v'	⟨1,1,2,2,2,2,1,1⟩
Cost	'e'	⟨1,1,2,2,2,2,2⟩
25	'v'	⟨1,1,2,2,2,2,2,1⟩
SectorInfo	'e'	⟨1,1,2,3⟩
IT	'e'	⟨1,1,2,3,1⟩
200	'v'	⟨1,1,2,3,1,1⟩
Categories	'e'	⟨1,1,2,3,2⟩
Category	'e'	⟨1,1,2,3,2,1⟩
Type	'e'	⟨1,1,2,3,2,1,1⟩
Mgmt	'v'	⟨1,1,2,3,2,1,1,1⟩
Cost	'e'	⟨1,1,2,3,2,1,2⟩
10	'v'	⟨1,1,2,3,2,1,2,1⟩
Category	'e'	⟨1,1,2,3,2,2⟩

C	T	I
(Continued)		
Type	'e'	⟨1,1,2,3,2,2,1⟩
Admin	'v'	⟨1,1,2,3,2,2,1,1⟩
Cost	'e'	⟨1,1,2,3,2,2,2⟩
30	'v'	⟨1,1,2,3,2,2,2,1⟩
Category	'e'	⟨1,1,2,3,2,3⟩
Type	'e'	⟨1,1,2,3,2,3,1⟩
Expert	'v'	⟨1,1,2,3,2,3,1,1⟩
Cost	'e'	⟨1,1,2,3,2,3,2⟩
50	'v'	⟨1,1,2,3,2,3,2,1⟩
Country	'e'	⟨1,2⟩
Name	'a'	⟨1,2,1⟩
FI	'v'	⟨1,2,1,1⟩
Sectors	'e'	⟨1,2,2⟩
SectorInfo	'e'	⟨1,2,2,1⟩
Forest	'e'	⟨1,2,2,1,1⟩
22	'v'	⟨1,2,2,1,1,1⟩
Number	'e'	⟨1,2,2,1,2⟩
4	'v'	⟨1,2,2,1,2,1⟩
Categories	'e'	⟨1,2,2,1,3⟩
Category	'e'	⟨1,2,2,1,3,1⟩
Type	'e'	⟨1,2,2,1,3,1,1⟩
Mgmt	'v'	⟨1,2,2,1,3,1,1,1⟩
Cost	'e'	⟨1,2,2,1,3,1,2⟩
3	'v'	⟨1,2,2,1,3,1,2,1⟩
Category	'e'	⟨1,2,2,1,3,2⟩
Type	'e'	⟨1,2,2,1,3,2,1⟩
Worker	'v'	⟨1,2,2,1,3,2,1,1⟩
Cost	'e'	⟨1,2,2,1,3,2,2⟩
30	'v'	⟨1,2,2,1,3,2,2,1⟩
SectorInfo	'e'	⟨1,2,2,2⟩
IT	'e'	⟨1,2,2,2,1⟩
80	'v'	⟨1,2,2,2,1,1⟩
Categories	'e'	⟨1,2,2,2,2⟩
Category	'e'	⟨1,2,2,2,2,1⟩
Type	'e'	⟨1,2,2,2,2,1,1⟩
Mgmt	'v'	⟨1,2,2,2,2,1,1,1⟩
Cost	'e'	⟨1,2,2,2,2,1,2⟩
8	'v'	⟨1,2,2,2,2,1,2,1⟩
Category	'e'	⟨1,2,2,2,2,2⟩
Type	'e'	⟨1,2,2,2,2,2,1⟩
Admin	'v'	⟨1,2,2,2,2,2,1,1⟩
Cost	'e'	⟨1,2,2,2,2,2,2⟩
15	'v'	⟨1,2,2,2,2,2,2,1⟩
Category	'e'	⟨1,2,2,2,2,3⟩
Type	'e'	⟨1,2,2,2,2,3,1⟩
Expert	'v'	⟨1,2,2,2,2,3,1,1⟩
Cost	'e'	⟨1,2,2,2,2,3,2⟩
20	'v'	⟨1,2,2,2,2,3,2,1⟩

Figure A2. The XML relation representation of the XML document in Figure A1.

Appendix B

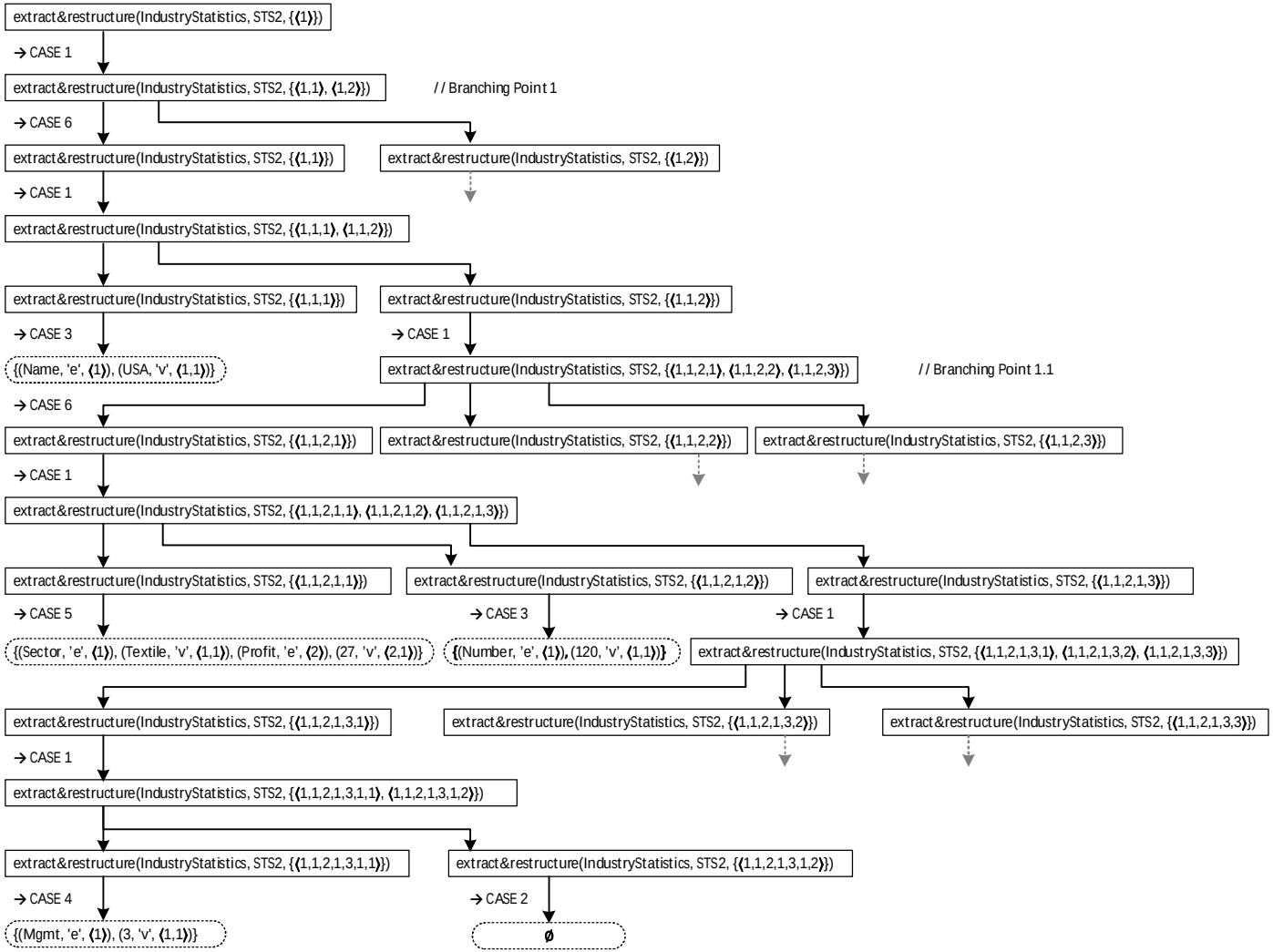


Figure B1. The (partial) evaluation tree related to Phase I of Example 3.