A unified approach to structured and XML data modeling and manipulation

Matteo Magnani *

Department of Computer Science, University of Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy

Danilo Montesi

Department of Mathematics and Informatics, University of Camerino,
Via Madonna delle Carceri 9, 62032 Camerino MC, Italy

Abstract

In this paper we propose an approach to defining logical database models, based on the instantiation of a general abstract model, and discuss its application to the management of mixed XML/relational data. Our abstract model is equipped with a parametric query algebra and relational-like algebraic equivalences, that do not have to be redefined when new models are generated. We present an instantiation of our model that unifies the main approaches to represent and manipulate relational and XML data, and in particular SQL, SQL/XML, XQuery, and Oracle's XML Type. Additionally, our algebra can represent queries not expressible by other algebras taken from the literature. Among others, it can represent nested XQuery expressions with no constraints on nesting and on node constructors.

Key words: Data modeling and manipulation, Heterogeneous representation formats, XML, Query translation, SQL/XML, XQuery

1 Introduction

In this paper we propose an approach to defining logical database models, and discuss its application to the management of mixed XML/relational data. In our approach, models are defined through the instantiation of a general abstract model.

* Corresponding author. Tel. +39 051 2094871, Fax +39 051 2094510
Email addresses: matteo.magnani@cs.unibo.it (Matteo Magnani), danilo.montesi@unicam.it (Danilo Montesi).

As an example, we show that the relational and nested relational models, as well as the model underlying the TAX algebra, can be obtained as specializations of our abstract model. The application of our approach to the management of mixed XML/relational data results in a query algebra unifying the main XML query languages currently used in database management systems, i.e., XQuery, SQL/XML and Oracle's XML Type methods. This algebra can represent, among other things, nested XQuery expressions with no constraints on nesting and on node constructors. Moreover, we prove some general algebraic equivalences that, though defined on the abstract model, still hold in its instantiations, enabling logical query rewriting.

Until a few years ago, database management systems were mainly based on a single data model, i.e., the relational one. This was possible because data was simple enough to be stored in tables, and because relational database systems had proved to be very efficient and dependable. Today, many different kinds of data can be stored in digital information systems, e.g., books, phone calls, mails, photos and movies. In the not too distant future, almost everything about our lives will be stored in databases where the relational model captures only a fraction of the relevant data [1]. In this scenario, one of the challenges in the field of data management is the definition of new models, to accommodate new kinds of data and even heterogeneous combinations of existing kinds of data.

The database community has adopted two main approaches to deal with the limitations of the relational model. Industry has been mainly driven by short-term goals: the objective was to quickly develop new dependable systems, with small discomfort for new and existing customers. The best way to achieve both goals consisted of reusing existing models and software as much as possible. All major commercial systems have adopted (at least) this approach, that we shall call *adaptation*. For example, to provide support for XML data, traditional tables are still used as a storage format: trees are shredded into relations, and from relations it is possible to build tree-structured output. In particular, Oracle and IBM have embraced a standard extension of SQL, called SQL/XML, while Microsoft has defined a proprietary extension based on a new FOR XML clause [2]. While fulfilling its short-term goals, this solution can only be used when the difference between XML data and relational tables is mainly syntactical. When XML data is semistructured, i.e., characterized by a sort of schema which is partial, irregular, built a posteriori, wide, or frequently modified, it is not possible to store it inside relations without altering its natural structure.

The second approach, mainly adopted in academia, consists of the definition of new models from scratch – we call this approach *rethinking*. This is motivated by the claim that each different kind of data can be best managed if a model is specifically tailored to it, taking advantage of its specificities. In the long run, this approach promises better results than *adaptation*. Focusing again on XML data, new query languages [3–5] and formal models [6–10] have been proposed, and new systems

have been implemented [1] . While probably fulfilling its long-term goals, within this approach a lot of time can be wasted reinventing or redefining capabilities that end up being like those which we can find in relational systems. Developing new models is not easy, and building and testing new systems based on them may need much effort. For instance, after many years of research, a widely recognized algebra for XML data has not emerged yet. However, if we look at existing proposals, they share many similarities with relational algebra, suggesting that it is possible to reuse what already exists, updating only what is specific to the new data.

In this paper, we define an abstract model generalizing the logical level of database systems, to overcome the drawbacks of the two aforementioned approaches. This generalization separates different data representation and manipulation capabilities that are usually mixed together, such as joining large sets of instances, pruning an XML tree, or extracting a substring from a textual value. In this way, whenever a new data model is needed, it can be produced using our abstract model by tuning only those functionalities that are specific to the new data. Using our abstraction, new models can be easily generated starting from existing ones, with a few changes – thus, partially fulfilling the goals of *adaptation*. At the same time, the new models are specifically tailored on their data – thus, fulfilling the goals of *rethinking*.

The contribution of this paper is twofold. First, we introduce a general abstract data model, that can be used to generate new models for specific application contexts. This model is equipped with a parametric algebra and relational-like algebraic equivalences, that do not have to be redefined when new models are generated. The second contribution is a practical application of our model to relational and XML data management, and in particular to SQL/XML, XQuery, and Oracle's XML Type. By instantiating our parametric algebra, we obtain a very simple language that can represent queries not expressible by other algebras taken from the literature, and by the current implementation of TAX [6,9]. Moreover, because general properties of our abstract model are still valid on its instantiations, algebraic equivalences are automatically inherited by the new query algebra.

The paper is organized in two main parts. In Sections 2, 3, and 4 we introduce our abstract model. First, in Section 2, we present an informal overview, where its main features are justified. In Section 3 we provide formal definitions of our abstract data structures and algebraic operators. In the same section we also present abstract algebraic equivalences, and discuss the definition of constraints on the abstract model. Finally, in Section 4 we show by example that our model can be instantiated to some of the main existing models and languages. In the second part of the paper we present a motivating application of our work. We consider the relevant context of mixed XML/relational systems, and produce a customized model and algebra using our approach. In particular, in Section 5 we show how the new model is obtained step by step, while in Section 6 we present mappings from the major query

[1] http://www.w3.org/XML/Query#products

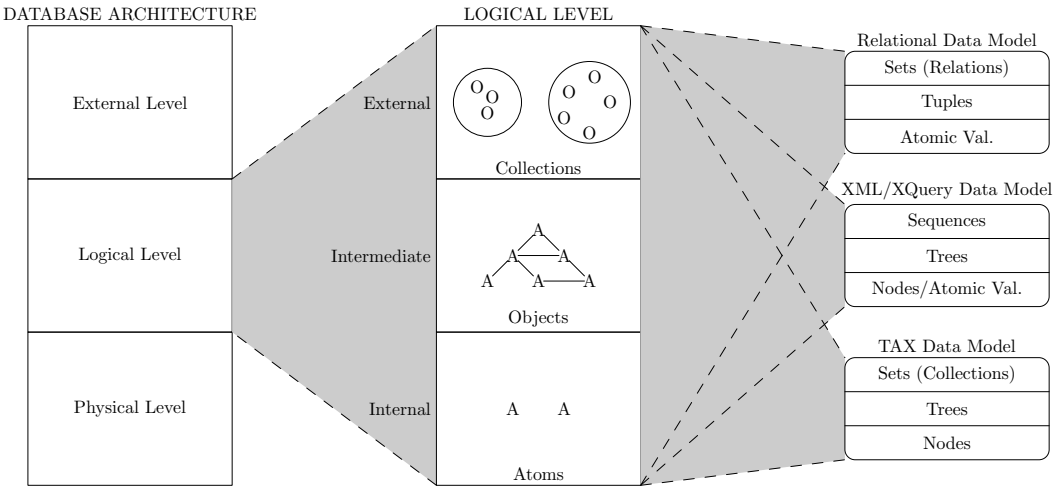


Fig. 1. Our abstract layered logical model. Databases are traditionally organized into three levels: External, Logical, and Physical. We decompose the logical level itself into three other layers: the external logical level, containing collections of objects, the intermediate logical level, representing their structure, and the internal logical level, concerning atoms. The main existing data models can be obtained as instantiations of these abstract layers. The difference between the XQuery model and the model underlying TAX is that TAX manipulates *sets* of *trees*, while XQuery manipulates *sequences* of *trees and atomic values*.

languages currently in use in XML-enabled systems to the query algebra obtained through the instantiation.

2 Informal Overview of the Abstract Model

Relational databases are traditionally organized into three abstraction levels, to achieve what is usually called *data independence*. The logical level provides an abstract representation of databases. At the external levels, many different views can be defined over a logical database, to provide each user with customized access to it. The physical level describes its physical storage structure. In this way, each level can change without affecting the others. New views can be created, without modifying the actual database. New physical storage strategies and algorithms can be implemented, without changing the logical representation of the data.

In this paper, we divide the logical level itself in three layers, to achieve similar advantages. In our abstract model, which is a three-layered version of traditional database logical levels, different granularities of data representation have been identified and separated, as illustrated in Fig. 1. For each level of granularity, a specific query language is defined, as illustrated in Fig. 2. These languages are like Russian *matrioshka* nesting dolls: they are defined so that each layer fits into the upper one, resulting in a single composite query language, where any "dolls" can be easily substituted with others of the same size, leaving the rest unchanged. The architec-

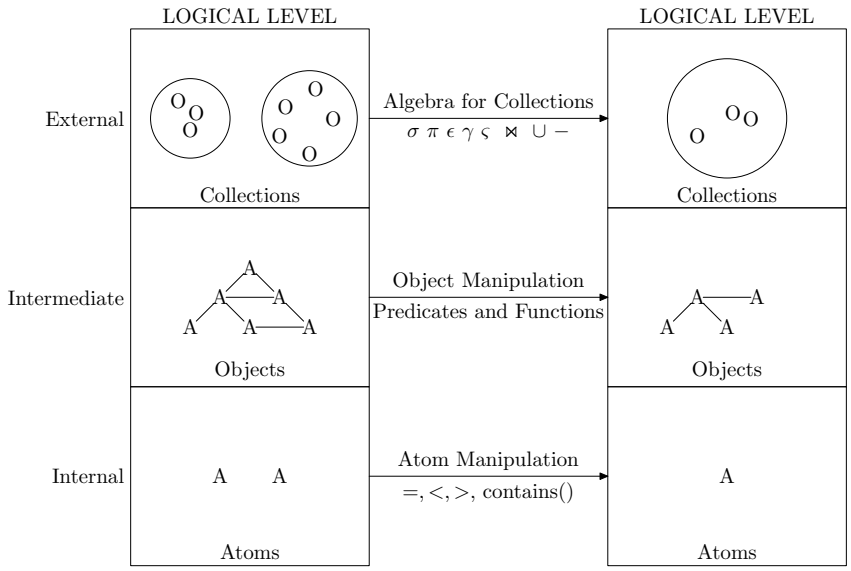


Fig. 2. Three kinds of query languages correspond to our three abstraction layers. They separate different kinds of data manipulation capabilities, such as joining large sets of instances, pruning an XML tree, or extracting a substring from a textual value.

ture of our abstract layered logical model is motivated by the following three basic considerations.

The first basic consideration is that there are operations that we can nearly always perform on collections of data, and can be modeled without paying particular attention to its peculiarities. For instance, consider the following queries:

- Give me all Web Pages where the Title is “home page”.
- Give me all Tuples where the Identifier is “001”.
- Give me all Images where the Prevalent Color is “Green”.

It is easy to notice that, despite their different semantics and application contexts, all these queries can be generalized as: “Give me all *objects* where *a predicate P is satisfied*”. Objects and predicates may vary, but the kind of operation is always the same: check the available objects, and when the predicate is satisfied output them.

The *external logical level* concerns *collections of objects*. A collection can be a set, a bag, or a sequence. For the sake of clarity, the external logical level presented in this paper uses the simplest type of collection, i.e., the set – extensions to bags and sequences can be defined starting from our current work. Depending on applications, objects can be tuples in a relation, files in a folder, objects in an object-oriented repository, text documents in a digital library, or XML documents in a collection. Algebraic operators for the manipulation of collections are parametric extensions of traditional relational operators.

The second basic consideration from which our model originates is that objects are usually internally structured by means of a limited number of aggregation patterns.

For instance, tuples are sets of attribute-name/value pairs, XML documents are trees, and semistructured data instances are graphs. We can represent a color and a web page, and say that a color is composed of its *blue/red/green* components in the same way as a web page is composed of a *head* and a *body*.

The *intermediate logical level* concerns the *internal organization (structure) of objects*. The fact that we allow any complex objects to be used is a key point to enable XML data to be represented homogeneously inside relational tuples. At the intermediate logical level, a specific language to manipulate single objects must be defined. For instance, when objects represent XML trees, typical languages used to manipulate them are path expressions, XPath, and TAX pattern trees [11,5,6].

The third basic consideration is that there are operations that cannot be described by a simple, general, and compact model, as they are meaningful only when applied to particular kinds of data. For instance, the extraction of a color histogram can be performed only on images. Therefore, even in our general abstract model we do not want to explicitly represent all the features of data. When a given level of detail is reached, we hide heterogeneity inside elementary pieces of data, and therefore outside our model.

The *internal logical level* concerns *atoms*, which represent the smallest level of detail we can manage within our model. The type of an atom tells us what can and cannot be done on it. We push into typed atoms the specific features of every object. Our atoms have been inspired by the way in which the Oracle database system has been extended to support XML Type columns. Adding support for a new kind of data is as easy as defining a new column type. However, in our model atoms are not constrained to be embedded inside relations, and are used to represent smaller levels of detail than entire XML trees, to reduce the complexity of the definition of new types.

This layered view of the logical level allows us to separate different data manipulation capabilities, as illustrated in Fig. 2. The languages expressing these capabilities can be changed depending on the kind of object we need to manage, to adapt to specific kinds of data. Our algebraic operators are based on predicates and functions for the manipulation of single objects. The language used to manipulate objects depends itself on the language used to manipulate atoms. This abstraction should simplify the application of local optimization strategies based on specific data types. For instance, in the context of XML data management, at the external level algebraic equivalences can be used to anticipate selections, so that XML documents that are of no interest do not slow down query answering. Independently of these equivalences, efficient techniques for the fast evaluation of XPath queries can be developed to speed up the verification of selection predicates. Obviously, the possibility of applying local optimization strategies does not exclude the use of global techniques.

Software systems do not need to explicitly implement our three logical abstraction levels. Our approach is mainly intended to be used to design new models, and these models can be directly implemented without references to the procedure used to generate them. However, our three layers can also be viewed as a data model for new-generation flexible database systems, which are able to switch from one model to another when the data format changes. In modern relational systems, we can change the physical organization of data or external views without the need to change the other abstraction levels. With our model we expect to have similar advantages. For instance, if an information system is extended with a data source containing a new kind of object, we will be able to modify the intermediate logical level and the corresponding language without changing the other two levels. Similarly, in a system for XML data that can process only some kinds of nodes, such as text and JPEG images, we would be able to add support for new data types, just by modifying the language for atoms.

3 Formal Definition of the Abstract Model

In this section we formally define the key concepts of our abstract model. In Sections 3.1 and 3.2 we define collections, objects, and parametric algebraic operators over collections. In Section 3.3 we introduce some algebraic equivalences, while in Section 3.4 we discuss the representation of constraints on collections.

3.1 Collections (External Logical Level)

Collections are composed of *objects*, which are aggregations of *entities*. An equivalence and an order relation on entities are all we need to define an abstract query algebra over collections. In the same way as collections, objects can be sets, bags, or sequences of entities, and this presentation of our model is based on sets – this does not clash with the application to XML data introduced in the second part of the paper, because entities can be internally ordered.

Let **entity** be a set, and $\doteq$ and $\dot{\subseteq}$ two relations on **entity**. $\doteq$ is an equivalence relation, while (**entity**,$\dot{\subseteq}$) is a partially ordered set. In the following, we will call $\dot{\subseteq}$ a *part of* relation. For example, an XML fragment is part of an XML document, and an attribute-name/value pair is part of a relational tuple. A collection C is a set of sets over the domain **entity**. Given two objects c_1 and c_2, $c_1 = c_2$ iff $\forall e \in c_1(\exists e' \in c_2(e \doteq e'))$, and vice-versa. $c_1 \subseteq c_2$ iff $\forall e \in c_1(\exists e' \in c_2(e \doteq e'))$, as usual.

The algebra described in this section focuses on the external logical level defined above. Its main feature is that it is parametric on the language used to manipulate single instances. In this way we distinguish between entity manipulation languages and operators for sets of objects. Every operator is parametric on the kind of predicates and functions it supports. Therefore, our language may be used together with any language for objects satisfying the constraints specified in the following definitions.

Two important functionalities borrowed from the relational world are *selection* (σ) and *projection* (π). They can be easily defined as parametric generalizations of the relational ones:

Definition 1 (selection) *Let* P *be a predicate over a set of objects. Therefore, if* c *is an object,* $\mathrm{P}(c)$ *is either true or false. A* selection *operator* σ *applied to a collection C is defined in the following way:*

$$\sigma_{\mathrm{P}}(C) = \{c \mid c \in C \wedge \mathrm{P}(c)\} \ . \tag{1}$$

Definition 2 (projection) *Let* PF *be a function over a set of objects to a set of objects, where* $\forall e \in \mathrm{PF}(c)\ (\exists e' \in c\ (e \dot{\subseteq} e'))$. $e \dot{\subseteq} e'$ *means that* e *is part of* e'. *A* projection *operator* π *applied to a collection C is defined in the following way:*

$$\pi_{\mathrm{PF}}(C) = \{\mathrm{PF}(c) \mid c \in C\} \ . \tag{2}$$

Projection is an extraction operator, and is complemented by a construction one: An *embedding* (ϵ) extends objects with new data. As the expressive power of these operators is left undefined at this level, projection cannot be considered the inverse of embedding in general. We can only state that they perform opposite tasks.

Definition 3 (embedding) *Let* EF *be a function over a set of objects to a set of objects, where* $\forall e \in c\ (\exists e' \in \mathrm{EF}(c)\ (e \dot{\subseteq} e'))$. *An* embedding *operator* ϵ *applied to a collection C is defined in the following way:*

$$\epsilon_{\mathrm{EF}}(C) = \{\mathrm{EF}(c) \mid c \in C\} \ . \tag{3}$$

Projection and embedding may be seen as horizontal operators, as they shrink or augment single objects, but they do not change the cardinality of a collection. As our model is intrinsically based on aggregation [2], they do not cover all required operations. The *split* (ς) operator decomposes a single object into many objects. This operator does not exist in relational algebra, where structure cannot change.

[2] Objects are not supposed to be atomic.

However, it is fundamental in a semistructured context. For example, a single XML file (entity) may store data about several books. Therefore we need a deconstruction operator to produce a new object for every book. A simple projection would only separate the books, keeping them together inside the same set. Grouping (γ) has a behavior which is similar to that of the split operator, but in the opposite direction. Again, ς is not the inverse of γ in general.

Definition 4 (splitting) *Let* SF *be a function over a set of objects to a set of objects, such that* $\mathrm{SF}(c) \subseteq c$*. A* split *operator* ς *applied to a collection C is defined in the following way:*

$$\varsigma_{\mathrm{SF}}(C) = \{\mathrm{SF}(c) \cup \{e\} \mid c \in C \wedge e \in c \backslash \mathrm{SF}(c)\} \ . \tag{4}$$

Definition 5 (grouping) *Let* SF *be a function over a set of objects to a set of objects, such that* $\mathrm{SF}(c) \subseteq c$*. An* SF*-equivalence class over a collection C is defined in the natural way as* $[c]^C_{\mathrm{SF}} = \{c' \in C \mid \mathrm{SF}(c') = \mathrm{SF}(c)\}$*. A* group *operator* γ *is defined in the following way:*

$$\gamma_{\mathrm{SF}}(C) = \{ \bigcup_{c \in [c']^C_{\mathrm{SF}}} c \mid c' \in C\} \ . \tag{5}$$

When SF is omitted, we assume that it returns the empty set independent of the input object.

The operators presented so far can be used to select, shrink, augment, compose or decompose data. In each case, the original data is not altered. These operators offer a different view of the input objects – for example, projection highlights parts of objects. We still need operators to change the appearance of existing data, and we call them *presentation* operators. However, these operators depend on the specific structure of objects. For instance, if entities are internally ordered, we may need an ordering operator. As a consequence of their dependency on actual data structures, presentation operators must be defined contextually to the definition of objects. At this abstraction level, these operators can be thought of as an embedding which constructs the different representations of entities at the side of the old ones, followed by a projection that keeps only the newly created ones.

Finally, we have traditional *join* and *set oriented* operators. Join, union, and difference are the only binary operators. Join and union are used to compose data previously distributed to different collections, usually for normalization. The former is a sort of horizontal union, while the latter operates vertically, as usual.

Definition 6 (join) *Let* P *be a predicate over pairs of objects. A* join *operator* $\bowtie$ *applied to two collections* C_1 *and* C_2 *is defined in the following way:*

$$C_1 \bowtie_{\mathrm{P}} C_2 = \{c_1 \cup c_2 \mid c_1 \in C_1 \wedge c_2 \in C_2 \wedge \mathrm{P}(c_1, c_2)\} \ . \tag{6}$$

Definition 7 (union) *The* union *($\cup$) of two collections C_1 and C_2 is defined in the following way:*

$$C_1 \cup C_2 = \{c \mid c \in C_1 \vee c \in C_2\} \ . \tag{7}$$

Definition 8 (difference) *The* difference *($-$) of two collections C_1 and C_2 is defined in the following way:*

$$C_1 - C_2 = \{c \mid c \in C_1 \wedge c \notin C_2\} \ . \tag{8}$$

All these operators are closed, i.e., they take collections as input and they return a collection. In this way the integration of presentation operators is straightforward, as they can be interleaved with the other operators of the algebra without any constraints. Intersection can be derived using difference, as usual.

3.3 Algebraic Equivalences

After thirty years of research and applications, many optimization strategies have been developed for relational data. In particular, with regard to the relational logical level, it is well known that algebraic query rewriting can significantly increase the efficiency of database systems. The idea is that every query can be represented by many equivalent algebraic expressions, specifying different *query plans*. Algebraic equivalences can be used to transform expressions into semantically equivalent but more efficient queries. In this paper we focus on rules regarding the selection operator. The proofs of correctness of the following propositions are in Appendix A.

The aim of this section is to define extensions of relational equivalences for our abstract model. In this way, traditional logical optimization techniques can be reused or adapted to new kinds of data, as has already been done in practice. When new models are generated using our approach, these equivalences still hold, and they can be used without the need to redefine them. The possibility of using these rules to perform logical query optimization on new models depends on the companion language. As is intuitively evident, it is better to keep the complexity of this language low. For example, if the language used to express predicates is Turing-complete, the algebraic operators may not converge. However, very simple languages are sufficient to perform significant classes of queries.

Some of the equivalences presented in this section are direct generalizations of those already defined in RA.

Proposition 9 (Atomization of selections)

$$\sigma_{\mathrm{P}_1 \wedge \mathrm{P}_2}(C) = \sigma_{\mathrm{P}_1}(\sigma_{\mathrm{P}_2}(C)) \ .$$

This atomization is typically used as a preliminary transformation. Another preliminary transformation is the inversion of selection and projection.

Proposition 10 (Inversion of selection and projection)

$$\sigma_{\mathrm{P}}(\pi_{\mathrm{PF}}(C)) = \pi_{\mathrm{PF}}(\sigma_{\mathrm{P}'}(C))$$
$$\textit{if } \forall c \in C\ (\mathrm{P}(\mathrm{PF}(c)) = \mathrm{P}'(c))\ .$$

This condition is more general than its relational counterpart (see Example 35).

One of the most typical equivalences of RA is generalized as follows:

Proposition 11 (Pushing selections down into joins)

$$\sigma_{\mathrm{P}}(C_1 \bowtie_{\mathrm{P}'} C_2) = C_1 \bowtie_{\mathrm{P}'} \sigma_{\mathrm{P}}(C_2)$$
$$\textit{if } \forall c' \in C_1, c'' \in C_2\ (\mathrm{P}(c' \cup c'') = \mathrm{P}(c''))\ .$$

The condition specifies that P does not depend on the objects in C_1.

Equivalences similar to those above can be extended to the new operators of the algebra. In particular, the definitions of projection and embedding are very similar, and so are their respective equivalences:

Proposition 12 (Inversion of selection and embedding)

$$\sigma_{\mathrm{P}}(\epsilon_{\mathrm{EF}}(C)) = \epsilon_{\mathrm{EF}}(\sigma_{\mathrm{P}'}(C))$$
$$\textit{if } \forall c \in C\ (\mathrm{P}(\mathrm{EF}(c)) = \mathrm{P}'(c))\ .$$

Interesting things happen when we define transformations concerning selection or projection and splitting or grouping, that do not have a relational counterpart. For example, a splitting gives a vertical view of an object by redistributing its entities to several objects. As a consequence, a selection pushed inside a splitting becomes a projection.

Proposition 13 (Pushing selections down into splittings)

$$\sigma_{\mathrm{P}}(\varsigma(C)) = \varsigma(\pi_{\mathrm{PF}}(C))$$
$$\textit{if } \forall c \in C\ (\mathrm{PF}(c) = \{e \mid e \in c \wedge \mathrm{P}(\{e\})\})\ .$$

This set of rules, together with other similar equivalences that can be defined on the abstract algebraic operators, can be augmented by other rules for presentation operators.

We conclude this section with a discussion of how to define constraints at the external logical level. We proceed in the same way as we have done to define our abstract algebra: constraints may be defined using parametric functions. In particular, we use PF functions, that we introduced in the definition of the projection operator: $\forall e \in \mathrm{PF}(c)\ (\exists e' \in c\ (e \dot{\subseteq} e'))$.

Functional dependencies are very important in relational databases. They are used both to define the constraints of key and super-key, and as a basis for the theory of normalization. Functional dependencies have been traditionally defined between relational attributes, and can be easily generalized to *parts of* objects, of which attributes are a specific case. If C is a collection, PF_B functionally depends on PF_A, written $\mathrm{PF}_A \rightarrow \mathrm{PF}_B$, iff $\forall c_1, c_2 \in C\ (\mathrm{PF}_A(c_1) = \mathrm{PF}_A(c_2) \implies \mathrm{PF}_B(c_1) = \mathrm{PF}_B(c_2))$.

Another important concept in the relational model is that of the *referential integrity constraint*, which is used to ensure correspondences between attributes from different relations through the definition of foreign keys. In the same way as functional dependencies, referential integrity constraints are generalized to parts of objects. If C_1 and C_2 are collections, referential integrity constraints can be specified through algebraic projections of the form: $\pi_{\mathrm{PF}_A}(C_1) \subseteq \pi_{\mathrm{PF}_B}(C_2)$.

4 Intermediate and Internal Logical Levels: Examples of Instantiations

The abstract model presented in the previous section can be used to produce concrete models through the instantiation of its lower levels, by following three steps. First, we must define the internal structure of the objects, together with the two relations $\doteq$ and $\dot{\subseteq}$, and possible restrictions on the way we group them into sets. Then, we must instantiate the predicates and functions to manipulate the objects, i.e., $\mathrm{P}, \mathrm{PF}, \mathrm{EF}$, and SF. Finally, we can define presentation operators, if needed. In general, the intermediate logical level may itself be based on lower-level functionalities, embedded inside functions on atoms. However, in the following examples we will not use particular types of atoms or introduce specific functions, for the sake of clarity.

4.1 Relational Model

Let **attnames** be a set of attribute names, and **dom** be a set of values. An entity is a pair $(A : v)$ with $A \in$ **attnames** and $v \in$ **dom**. An object (tuple) is a set of entities

A	B	C
a	e	c
a	e	b
a	d	c

C1

Fig. 3. A relation C1, whose logical representation is: $\{\{(A:a),(B:e),(C:c)\},\{(A:a),(B:e),(C:b)\},\{(A:a),(B:d),(C:c)\}\}$.

with unique attribute names. Let t be a tuple. We define $\mathrm{sort}(t) = \{A \mid (A:v) \in t\}$. A relation instance r is a collection of tuples with the same sort. The sort of r is the same as that of its tuples. We notate $t[A_1,\ldots,A_k] = \{(A:v) \in t \mid A \in \{A_1,\ldots,A_k\}\}$ [3], $\mathrm{name}((A:v)) = A$, and $\mathrm{val}((A:v)) = v$.

In the relational model entities are very simple, and so are the notions of equality and part of. $(A_1:v_1) \doteq (A_2:v_2)$ iff $\mathrm{name}((A_1:v_1)) = \mathrm{name}((A_2:v_2))$ and $\mathrm{val}((A_1:v_1)) = \mathrm{val}((A_2:v_2))$, while $\dot{\subseteq}$ coincides with $\doteq$, as entities have no structure. An example of relation is presented in Fig. 3, together with its logical representation.

We refer to Named Relational Algebra (RA) as presented in [12]. Unary predicates P have the form $A = c$ or $A = B$. When applied to a tuple t, $A = c$ is true iff $\mathrm{val}(t[A]) = c$, and $A = B$ is true iff $\mathrm{val}(t[A]) = \mathrm{val}(t[B])$. The resulting instantiation, in the case of a predicate $A = c$, is:

$$\sigma_{A=c}(R) = \{t \mid t \in R \wedge \mathrm{val}(t[A]) = c\} \ .$$

Projection functions PF have the form $A_1,\ldots,A_k$. When applied to a tuple t, they return $t[A_1,\ldots,A_k]$. The instantiation of π is thus:

$$\pi_{A_1,\ldots,A_k}(R) = \{t[A_1,\ldots,A_k] \mid t \in R\} \ .$$

Binary predicates $\mathrm{P}(t_1,t_2)$ are defined as follows. Let Y be the sort of r_1 and Z be the sort of r_2, with $X = Y \cap Z$. Given two tuples $t_1 \in r_1$ and $t_2 \in r_2$, $\mathrm{P}(t_1,t_2)$ is true iff $t_1[X] = t_2[X]$. We do not specify this predicate after the $\bowtie$ symbol, as it is fixed.

$$\begin{aligned} R_1 \bowtie R_2 = &\{t_1 \cup t_2 \mid t_1 \in R_1 \wedge t_2 \in R_2 \wedge t_1[X] = t_2[X]\} = \\ &\{t \text{ over } Y \cup Z \mid t[Y] = t_1 \in R_1 \wedge\ t[Z] = t_2 \in R_2\} \ . \end{aligned}$$

Renaming is the only presentation operator, and it is borrowed from RA. We do not need to define EF and SF functions, as embedding, splitting and grouping are

[3] In the case of a single attribute, we will abuse notation and use $t[A]$ to represent both the object $\{(A:v)\}$ and the entity $(A:v)$.

not necessary to simulate RA. Union and difference are defined as in our general algebra.

4.2 Nested Relational Model

As in the previous example, a tuple is a set of name/value pairs, but values may be either elements of **dom** or tuples, and we allow repetitions of attribute names inside a tuple when the entities with the same name contain themselves tuples. A nested relation corresponds to one or more entities with the same name, each containing a tuple. Figure 4 shows an example of nested relations, together with their logical representations.

Constraints on the sorts are based on the concept of *nesting level*. An entity has the level of nesting of the tuple containing it. If a tuple is contained inside an entity, it has the level of nesting of that entity plus one. Tuples which are elements of a collection have nesting level 0. If we start from a tuple and go back to the most external tuple containing it, we may keep trace of the traversed entities to define its *nesting chain*. For example, the tuple $\{(C : b)\}$ of the nested relation C3 of Fig. 4 has nesting level 2 and nesting chain (D, E), because it is contained inside a column D, which is itself contained inside a column E. To be a nested relation, a collection must contain tuples that have the same sort whenever they have the same nesting chain. If one entity contains a tuple, all other entities with the same name and nesting chain must contain a tuple. It follows that there cannot be empty nested relations. Moreover, we do not consider the use of *null* values.

The relation $\doteq$ is defined recursively. If v_1, v_2 are elementary values, $(A_1 : v_1) \doteq (A_2 : v_2)$ iff name$((A_1 : v_1))$ = name$((A_2 : v_2))$, and val$((A_1 : v_1))$ = val$((A_2 : v_2))$. Else, $(A_1 : v_1) \doteq (A_2 : v_2)$ iff name$((A_1 : v_1))$ = name$((A_2 : v_2))$ and $v_1 = v_2$[4]. Additionally, $(A_1 : v_1) \dot{\subseteq} (A_2 : v_2)$ iff $(A_1 : v_1) \doteq (A_2 : v_2)$ or $(A_1 : v_1) \in v_2$. This formula means that a tuple inside an entity is considered part of that entity.

We refer to Nested Relational Algebra (NRA) as presented in [13]. Union, set difference, Cartesian product, projection, and selection are defined analogously to those presented in the previous example on RA (Cartesian product is a simple variation of Join). The distinguishing operators of NRA are nest (ν) and unnest (μ). To simulate them, we need an entity constructor and an entity deconstructor. Given a tuple t and an attribute A, we define them as:

$$\mathrm{Con}(A, t) = (A : t)$$
$$\mathrm{Des}((A : t)) = t \ .$$

[4] Checking this equality corresponds to verifying the $\doteq$ equality on the entities belonging to v_1 and v_2.

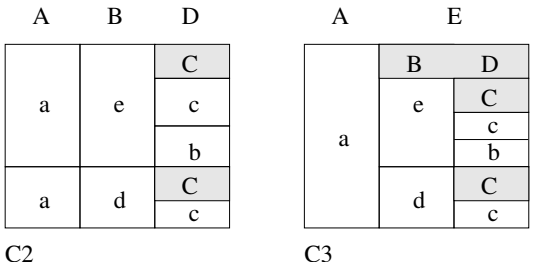


Fig. 4. Two nested relations. The logical representation of C2 is: $\{\{(A:a),(B:e),(D:\{(C:c)\}),(D:\{(C:b)\})\},\{(A:a),(B:d),(D:\{(C:c)\})\}\}$. The logical representation of the nested relation C3 is: $\{(A:a),(E:\{(B:e),(D:\{(C:c)\}),(D:\{(C:b)\})\}),(E:\{(B:d),(D:\{(C:c)\})\})\}$.

These functions are used to define two additional PF and EF parameters. Let t be a tuple of sort $\{C_1,\ldots,C_m,B\}$. PF can take the form just defined for RA, or μB, returning $t[C_1,\ldots,C_m]\cup \mathrm{Des}(t[B])$ when applied to t. Now, let t be a tuple with sort $\{C_1,\ldots,C_m,B_1,\ldots,B_k\}$. The EF function $\nu B = B_1,\ldots,B_k$ applied to t returns $t[C_1,\ldots,C_m]\cup \mathrm{Con}(B,t[B_1,\ldots,B_k])$. SF functions have the form $A_1,\ldots,A_k$, and when applied to a tuple t they return $t[A_1,\ldots,A_k]$.

Let R be a relation of sort $\{C_1,\ldots,C_m,B_1,\ldots,B_k\}$. We can express nesting and unnesting as:

$$\nu_{B=(B_1,\ldots,B_k)}(R) = \gamma_{C_1,\ldots,C_m}(\epsilon_{\nu B=B_1,\ldots,B_k}(R))$$
$$\mu_{B=(B_1,\ldots,B_k)}(R) = \pi_{\mu B}(\varsigma_{C_1,\ldots,C_m}(R)) \ .$$

As an example, notice that C1=$\mu_{D=(C)}$(C2), C2 = $\nu_{D=(B)}$(C1) = $\mu_{E=(B,D)}$(C3), C3 = $\nu_{E=(B,D)}$(C2). The fact that nesting corresponds to two operators of our algebra indicates that it expresses two elementary functionalities: creation (to create tuples/relations) and aggregation (to collect them inside a same relation). A similar remark can be done about unnesting.

4.3 Tree Algebra for XML (TAX) Model

In the case of XML in general, and of TAX in particular, **entity** is a set of ordered labeled trees. We can express the operators of TAX as compositions of our algebraic operators, with the possibility of temporary objects being composed of more than one tree, but producing single-tree objects in the end.

We refer to TAX as presented in [6]. As usual, some operators are specific to the kind of objects managed by the algebra, i.e., trees, so they are defined independently on our operators for the external logical level. These are renaming, reordering, and value-update. Set operators are the same as usual. Then there are some construc-

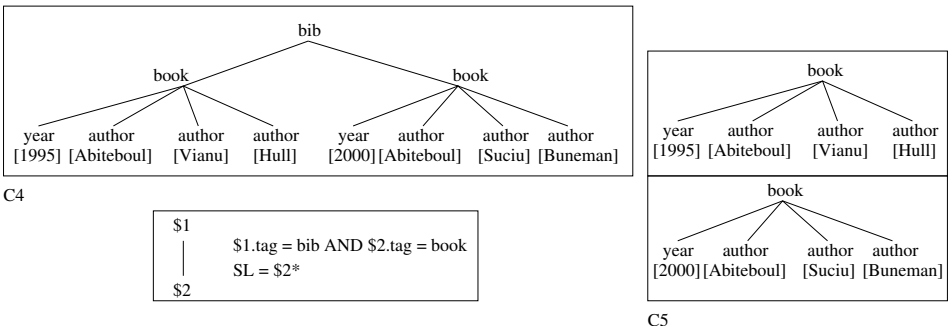


Fig. 5. An example of the TAX model. Collection C4 contains one data tree. The application of a selection, with pattern tree and selection list as specified in the box on the bottom, produces the collection C5.

tive operators, that are node-insertion and copy-and-past. These can be directly expressed as embeddings. Conversely, node-deletion can be expressed as a projection. To simulate TAX selection and projection, we can use a projection based on the following functions: $\mathrm{PF}_{\mathcal{P},\mathrm{SL}}$ and $\mathrm{PF}_{\mathcal{P},\mathrm{PL}}$ [5]. They are used to extract parts of the input trees (forests). Then, single-tree objects are obtained by means of a splitting. The fact that both selection and projection correspond to a projection in our algebra comes from the fact that in TAX these operators are very similar, and not orthogonal.

$$\mathrm{TAX}\sigma_{\mathcal{P},\mathrm{SL}}(C) = \varsigma(\pi_{\mathrm{PF}_{\mathcal{P},\mathrm{SL}}}(C))$$
$$\mathrm{TAX}\pi_{\mathcal{P},\mathrm{PL}}(C) = \varsigma(\pi_{\mathrm{PF}_{\mathcal{P},\mathrm{PL}}}(C))$$

Join, Cartesian product and grouping in TAX are a potpourri of functionalities: for example, new node need to be created because of the single-tree data model, and not because of the expected logical behavior of these operators. To perform a join using our operators, we must create a tax_prod_root node on top of the trees from the first collection, then join them with the other collection, then connect the tax_prod_root node to the newly added tree. To perform a grouping, we must obtain the grouping value by means of a projection, embed it inside a tax_grouping_list node (this can need an embedding of the original tree), then group the collection on tax_grouping_list, and embed the result to create the tax_group_subroot and tax_group_root nodes.

We do not delve into the formal representation of these operators in the context of our general algebra, as the model we develop in the second part of the paper has many similarities with TAX. However, the model presented in Section 5 is not restricted to XML and XQuery, but applies also to relational and mixed XML and relational data, and SQL/XML. While we keep the intuition that the elements of a model for XML (and not only for XML) can be trees, we will abandon the two

[5] $\mathcal{P}$ is a pattern tree, SL and PL are selection and projection lists, as described in [6]. An example of pattern tree and selection list is illustrated in Fig. 5

most important assumptions of TAX. First, objects are not single trees – we use forests, which allow cleaner operators, orthogonal projection and selection, and the representation of nested queries with no constraints on nesting. Second, trees are not necessarily manipulated using pattern trees – our algebra is independent on the entity manipulation language, and we can use any compatible language for objects. In Section 5 we will show how to instantiate the algebra with a simple path language, and that it is sufficient to express queries equivalent to XQuery-like expressions with unconstrained nesting of subexpressions.

5 An Application to Mixed XML and Relational data

In this section we present a complete application of our abstract model to the representation and manipulation of mixed XML and relational data. Today, a very large number of applications use XML as the language to interoperate with other programs, and as an internal data representation format. At the same time, the majority of information systems exchanging XML data were previously based on relational software. As a result, many current systems manipulate a mix of XML and relational data.

In the remainder of the paper, our aim is to produce a model supporting the main existing approaches in a unified way, as an instantiation of our abstract model. As for the examples of Section 4, this can be done following three main steps. First, we must define entities that can easily represent both attribute-name/value pairs and XML trees. At the same time, we must define equality and part of relationships on them. Second, we must define a language to express predicates and functions on these entities. Finally, we must introduce some presentation operators, to complete the instantiated algebra. Some of the following concepts have been presented in [10], independently of the general theory developed in the first part of the paper.

5.1 Entities for Mixed XML and Relational Data: Data Trees

In this specialization, the elements of **entity** are called *data trees*, while objects (sets of entities) are called *data graphs*.

Definition 14 (Data Tree) *A* data tree *is a tuple* $t = (V, E, \preceq, \lambda, \tau, \delta)$*, where:*

(1) $V = \{v_1, \ldots, v_n\}$ *is a finite set of vertices.*
(2) $E \subset \{(v_i, v_j) \mid v_i, v_j \in V\}$.
(3) (V, E) *is a directed tree.*
(4) $\preceq$ *is a possibly empty partial order on* V.
(5) $\lambda : V \to (L \cup \{\text{null}\})$*, where* L *is a set of names (labels).*

(6) $\tau : V \rightarrow T$*, where T is a set of types.*

(7) $\forall v \in V \left(\delta(v) = \begin{cases} \oplus(c(v)) & \textit{if outdegree(v)} \geq 1 \\ \delta'(v) & \textit{o.w.} \end{cases} \right)$

where $\oplus$ *is a parametric concatenation operator,* $c(v)$ *is the set of* v*'s children, and* δ' *is a content function defined on leaf nodes.*

Given a data tree t, we can define the following three derived functions: $r(t) = \{v \in V \mid \text{indegree}(v) = 0\}$ (root node), $c(v) = \{v_i \in V \mid (v, v_i) \in E\}$ (children of node v), and $l(t) = \{v \in V \mid \text{outdegree}(v) = 0\}$ (leaf nodes).

The intuition about edges is that "v_j *is part of* v_i" if $(v_i, v_j) \in E$[6]. Nodes may have labels (λ) and they may be ordered ($\preceq$). Every node has a type (τ) and a content (δ). The content of internal nodes need not be explicitly specified, but can be computed from the content of the leaves of the sub-trees rooted at them. For example, in an XML document the string content of a node is the concatenation of the string contents of all the leaves of its subtree. The corresponding content function for data trees is [14]:

$$\begin{aligned} \oplus(\{v_1, \ldots, v_m\}) &= \delta(v_{j_1}) \cdot \ldots \cdot \delta(v_{j_m}) \\ \delta'(v) &= \text{fn:string}(v) \end{aligned} \tag{9}$$

where $\forall i, p \in [1, m]$ $(i \leq p \Rightarrow v_{j_i} \preceq v_{j_p})$. $\preceq$ is the document order, $\cdot$ is the string concatenation operator, $\sqcup$ is the blank character and fn:string(v) returns the content of v converted as string. The organization of the type system is orthogonal to the definition of the model.

Example 15 (relational database) *Let* $\text{DB}(R_1(A, B), R_2(A, C))$ *be a relational database schema, with instance* $db(r_1, r_2)$*, as illustrated in Fig. 6. This can be represented as a database with schema* $\text{DB}'(C_1, C_2)$ *where every tuple* $\langle t_1, t_2 \rangle \in r_1$ *corresponds to a* data graph $d \in C_1, d = \{x_1, x_2\}$*, with* x_1 *defined as:*

- $V = \{v, u\}$
- $E = \{(v, u)\}$
- $\preceq = \emptyset$
- $\lambda(u) = null, \lambda(v) = A$
- $\tau(v)$ = *column,* $\tau(u) = \text{Dom}(A)$
- $\delta(u) = \delta(v) = t_1$

and x_2 *defined as:*

- $V = \{v, u\}$

[6] A data graph is a particular case of a semantic network, with a limited semantics but much easier to manipulate. Possible extensions of this model can be obtained by adding more semantics, i.e., new kinds of arcs and less restrictions on the graph structure $(E, \preceq)$.

- $E = \{(v, u)\}$
- $\preceq = \emptyset$
- $\lambda(u) = null, \lambda(v) = B$
- $\tau(v) =$ *column*, $\tau(u) = \mathrm{Dom}(B)$
- $\delta(u) = \delta(v) = t_2$

Other tuples are represented analogously.

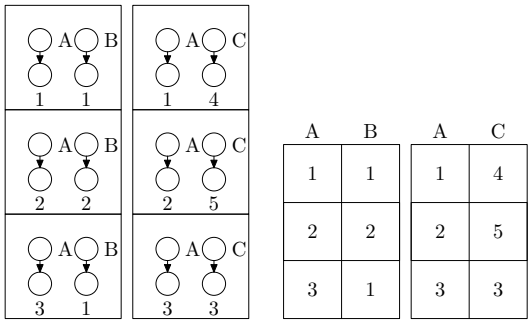


Fig. 6. A database composed of two relational tables. On the left we represent the corresponding collections, with labels and contents. This logical view of the database may seem redundant. However, at the physical level it is not necessary to replicate the attribute names.

Example 16 (XML document repository) *Let x be the following XML document:*

```
<book>
   <author>Hugo</author>
   <title>Notre-Dame de Paris</title>
</book>
```

We can model it using a data graph containing the following data tree:

- $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$
- $E = \{(v_0, v_1), (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_4, v_5)\}$
- $\preceq = \{(v_i, v_j) \mid i \leq j\}$
- $\lambda(v_0) = null, \lambda(v_1) =$ *book*, $\lambda(v_2) =$ *author*, $\lambda(v_4) =$ *title*, $\lambda(v_3) = \lambda(v_5) = null$
- $\tau(v_0) =$ *document*, $\tau(v_1) = \tau(v_2) = \tau(v_4) =$ *element*, $\tau(v_3) = \tau(v_5) =$ *text (or other types defined by a DTD/XMLSchema)*
- $\delta(v_0) = \delta(v_1) =$ *'HugoNotre-Dame de Paris'*, $\delta(v_2) = \delta(v_3) =$ *'Hugo'*, $\delta(v_4) = \delta(v_5) =$ *'Notre-Dame de Paris'*.

Here we have used the content aggregation equation (9). Many XML documents may be contained inside a collection, as illustrated in Fig. 7.

Example 17 (mixed model) *A relational model extended with an XML Type, as in the Oracle database system, can be represented by merging the modelings of Examples 15 and 16. In particular, we consider the relation represented in Fig. 8, with*

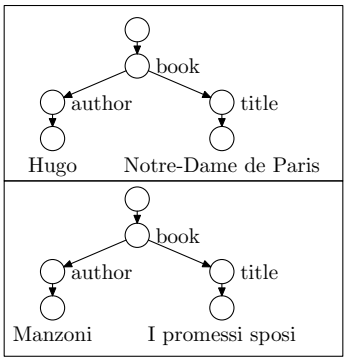


Fig. 7. An XML document repository. Roots are document nodes.

two string attributes and an XML Type column. The first tuple may be represented by a data graph $d_1 = \{t_{11}, t_{12}, t_{13}\}$*, where* t_{11} *and* t_{12} *correspond to the ID and Cl attributes, and* t_{13} *stores the XML data of the XData column.* t_{11} *is defined as:*

- $V = \{v, u\}$
- $E = \{(v, u)\}$
- $\preceq = \emptyset$
- $\lambda(u) = null, \lambda(v) =$ ***ID***
- $\tau(v) = \text{column}, \tau(u) =$ ***CHARACTER (2)***
- $\delta(u) = \delta(v) =$ ***012*** .

t_{12} *is defined as:*

- $V = \{v, u\}$
- $E = \{(v, u)\}$
- $\preceq = \emptyset$
- $\lambda(u) = null, \lambda(v) =$ ***Cl***
- $\tau(v) = \text{column}, \tau(u) =$ ***CHARACTER (6)***
- $\delta(u) = \delta(v) =$ ***HUG453*** .

t_{13} *is:*

- $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$
- $E = \{(v_0, v_1), (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_4, v_5)\}$
- $\preceq = \{(v_i, v_j) \mid i \leq j\}$
- $\lambda(v_0) = null,\ \lambda(v_1) =$ ***book***$,\ \lambda(v_2) =$ ***author***$,\ \lambda(v_4) =$ ***title***$, \lambda(v_3) = \lambda(v_5) = null$
- $\tau(v_0) =$ ***column***$,\ \tau(v_1) = \tau(v_2) = \tau(v_4) =$ ***element***$,\ \tau(v_3) = \tau(v_5) =$ ***text*** ***(or other types defined by a DTD/XMLSchema)***
- $\delta(v_0) = \delta(v_1) =$ ***'HugoNotre-Dame de Paris'***$, \delta(v_2) = \delta(v_3) =$ ***'Hugo'***$, \delta(v_4) = \delta(v_5) =$ ***'Notre-Dame de Paris'***.

Other tuples may be represented in a similar way. Figure 9 shows the table as represented in our model.

ID	Cl	XData
012	HUG453	<book><author>Hugo</author> <title>Notre-Dame de Paris</title></book>
013	MAN123	<book><author>Manzoni</author> <title>I Promessi Sposi</title></book>

Fig. 8. A mixed model, where relational and XML data coexist, represented as a table.

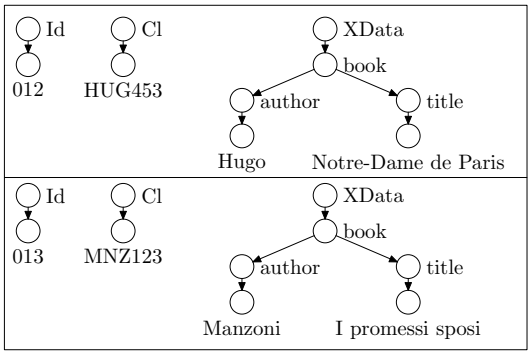


Fig. 9. A mixed model, where relational and XML data coexist, represented as a collection of data graphs.

5.2 Equality and Part Of Relations

Two data trees are equal when they have the same values, structured in the same way. This definition corresponds to a mathematical isomorphism, where we do not consider types:

Definition 18 (equality) *Two data trees* $t_1 = (V_1, E_1, \preceq_1, \lambda_1, \tau_1, \delta_1)$ *and* $t_2 = (V_2, E_2, \preceq_2, \lambda_2, \tau_2, \delta_2)$ *are equal, written* $t_1 \doteq t_2$*, if there is a structural isomorphism* i *between* V_1 *and* V_2 *which preserves* $E_1, \preceq_1, \lambda_1$*, and* δ_1*.*

We can obtain other notions of equality by restricting the isomorphism to a subset of the data representation dimensions. In particular, it may be useful to verify if two data trees are equal without considering the order. If this is the case, we say they are permutations of each other.

Definition 19 (permutation) *A data tree* $t_1 = (V_1, E_1, \preceq_1, \lambda_1, \tau_1, \delta_1)$ *is a* permutation *of* $t_2 = (V_2, E_2, \preceq_2, \lambda_2, \tau_2, \delta_2)$ *if there is a structural isomorphism* i *between* V_1 *and* V_2 *which preserves* E_1, λ_1*, and* δ_1*. We write* $t_1 \stackrel{!}{=} t_2$ *when* t_1 *is a permutation of* t_2*.*

Being part of something ($\dot{\subseteq}$) is a transitive relation, but transitivity is only implicitly represented in our model. Therefore, new arcs may appear on parts of data trees.

Definition 20 (part of) *A* part of *a data tree* $t = (V, E, \preceq, \lambda, \tau, \delta)$ *is a data tree* $t' = (V', E', \preceq', \lambda_{|V'}, \tau_{|V'}, \delta')$*, written* $t' \dot{\subseteq} t$*, with:*

(1) $V' \subseteq V$
(2) $E' \subseteq E^{tr}$ *(*E^{tr} *is the transitive closure of* E*)*
(3) $\preceq' \subseteq \preceq$
(4) $\lambda_{|V'}, \tau_{|V'}$ *indicate the restriction of* λ *and* τ *to the nodes in* V'
(5) $\forall v \in V' \left(\delta'(v) = \begin{cases} \oplus(c(v)) & \textit{if outdegree(v)} \geq 1 \\ \delta(v) & \textit{o.w.} \end{cases} \right)$.

5.3 Predicates and Functions for Data Graphs

In the previous section, we have shown that collections of data graphs can represent data relevant to our application context. Now, we need to provide predicates and functions for data graphs to instantiate the external algebra.

Underlying our predicates and functions, there is the functionality of node marking. After a set of nodes has been identified (marked), they can be projected, modified, or used to check a selection predicate. The process of node marking works as follows: first, we identify some root nodes. Then, from the identified nodes, we can descend into the trees to mark nodes at other levels.

Let Σ be $L \cup \mathrm{Var} \cup \{*\}$, where L is a set of labels and Var is a (disjoint) set of variable names. Root Node Markers are defined in the following way:

- $x \in \Sigma$ is a Root Node Marker (RNM).
- (ϕ, ϕ) is a RNM if ϕ is a RNM.
- $\overline{\phi}$ is a RNM if ϕ is a RNM.

The elements of $L \cup \mathrm{Var}$ identify root nodes based on their label, e.g., a identifies the root node whose label is a. $*$ identifies all the root nodes. A negated expression $\overline{\phi}$ identifies all the root nodes not identified by ϕ, while a list of comma-separated expressions takes the union of the nodes identified by them. For instance, $(a, \overline{b})$ identifies the root nodes labeled a and those not labeled b. In particular, the a part is redundant, as nodes labeled a are already identifieed by $\overline{b}$.

After having identified root nodes by means of RNMs, we can descend into their trees, by means of node markers (NMs).

- A RNM ϕ is a NM.
- $(\psi)/x$ is a NM if ψ is a NM and $x \in \Sigma$.
- $(\psi)//x$ is a NM if ψ is a NM and $x \in \Sigma$.
- $(\psi; \psi)$ is a NM if ψ are NMs.

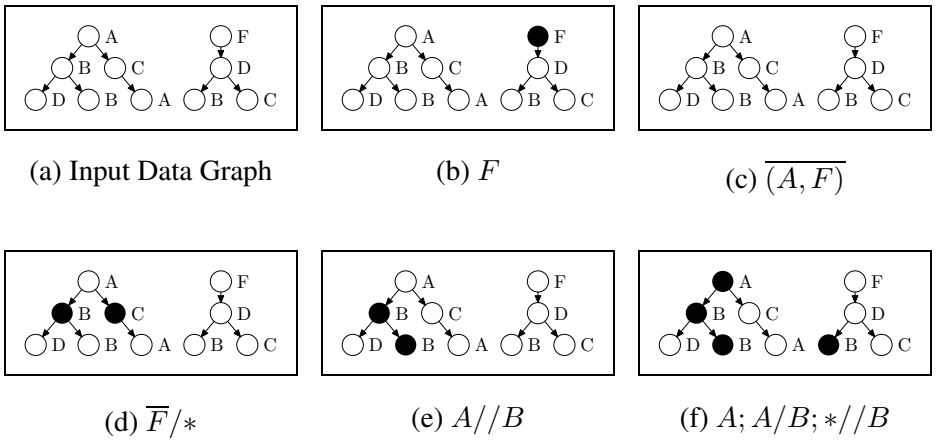


Fig. 10. Some examples of Node Markers. They have been applied to the data graph in Fig. 10(a), and the marked nodes have been represented by filled circles. Notice that in Fig. 10(f) the A/B part of the NM is redundant.

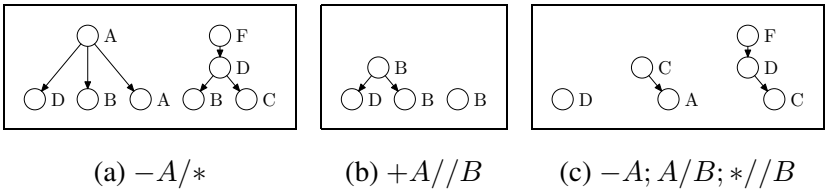
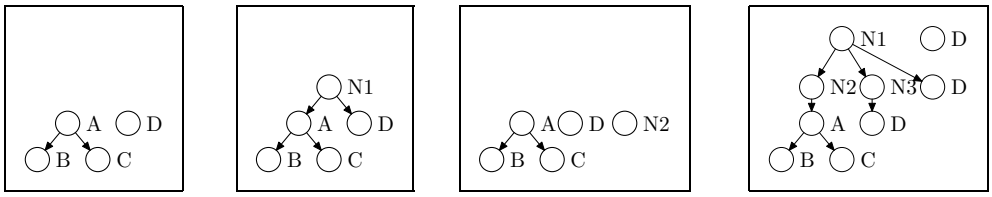


Fig. 11. Examples of projection functions, applied to the data forest in Fig. 10(a).

$/$ and $//$ are used to mark respectively, children and descendants of the nodes previously identified, if their label is x. A semi-colon takes the union of the nodes marked by its arguments – we will omit parentheses when unnecessary. Figure 10 shows some examples of node markers.

The other building blocks of our predicates and functions are node constructors and node comparison operators. Comparison operators may vary depending on the types of the nodes allowed at the internal logical abstraction level – we may expect to have powerful operators for complex types like images or MS Office documents. As we deal only with textual nodes in this paper, we will use the following constructors: $<l>$, to create a node with label l, and $'x'$, to create a node with content x.

Now we can define predicates and functions for data graph manipulation. There are two kinds of projection functions (PF): $+$ and $-$, both based on a node marker. The former extracts the subtrees rooted at the marked nodes. The latter deletes the marked nodes. Some projection functions are illustrated in Fig. 11. A subset function (SF) is just a $+$ projection function based on a RNM. An embedding function (EF) is a node constructor, followed by an optional list of children. Its children can be existing trees, marked by RNMs, or other embedding functions. Examples of



(a) Input (b) $<N1>$ $(*)$ (c) $<N2>$ (d) $< N1 >$ $(< N2 >$ (A),$<N3>$ $(D), +D)$

Fig. 12. Examples of embedding functions. They can be used to create new nodes on top of existing data forests (Fig. 12(b)) or next to them (Fig. 12(c)). When a $+$ is specified before a tree selection expression, the corresponding tree will be replicated and not just embedded.

embedding functions are provided in Fig. 12. Finally, a selection predicate (P) is a comparison of sets of nodes or data graphs, identified by node markers, extracted by projection functions, or created by means of node constructors. More formally,

- $+\psi$ and $-\psi$ are projection functions (respectively, positive and negative) if ψ is a node marker
- $+\phi$ is a subset function if ϕ is a RNM.
- $\mathrm{nc}(\phi, \ldots, \phi)$ is an embedding function if nc is a node constructor and ϕ are embedding functions or RNMs. If a RNM is anticipated by a $+$, the corresponding trees will belong to the result, i.e., they are also replicated (we will omit parentheses when unnecessary).
- $\psi\ \theta\ \psi$ is a selection predicate if ψ are node markers, projection functions, or node constructors, and θ is a node comparison operator.

The semantics of selection predicates (P) depends on the comparison operators.

5.4 Presentation operators

To complete the algebra instantiated by the predicates and functions previously defined, we still need presentation operators. Looking at the definition of data trees, we can identify two missing operators, used to change order and labels.

Definition 21 (ordering) *Let* OF *be a function which takes a data graph* d *as input and returns a data graph* d' *such that* $\forall t \in d\ (\exists t' \in d'(t \stackrel{!}{=} t))$ *and vice-versa*[7]*. An* ordering *operator* $\omega_{\mathrm{OF}}(C)$ *applied to a collection C returns a new data graph* $\mathrm{OF}(d)$ *for each* $d \in C$*.*

Definition 22 (renaming) *Let NM be a node marker. A* renaming *operator* $\rho_{\mathrm{name} \leftarrow \mathrm{NM}}$ *applied to a collection C returns a new data graph for each* $d \in C$*, where* $\lambda(v) =$

[7] For $\stackrel{!}{=}$ see Definition 19.

name for all nodes v in d that have been marked by NM.

The choice of the OF function depends on applications. As an example, we have shown a possible instantiation based on path expressions in Fig. 13. In particular, in Fig. 13(h) the first path expression (bib) identifies the nodes whose children must be ordered, and the second (book/date), applied to each child of bib, selects a single node whose content δ is used as the order criterion. In Fig. 13 we have also shown an example for each algebraic operator, as instantiated by the predicates and functions introduced in Section 5.3. The integration of presentation operators in the algebra is straightforward, as they take collections as input and return collections.

As a final remark, we point out that the set of equivalences available for the algebra can be extended with the introduction of new presentation operators. For example, we can easily define a rule for the inversion of renaming and selection, so that labels do not have to be changed in data graphs that will not be selected. However, a general discussion of these rules is not possible, as they depend on the internal organization of objects, which is left undefined at the external logical level.

6 Algebraic Representation of XML and Relational Queries

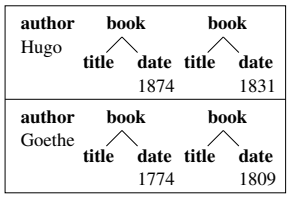
To conclude our example, we show how existing user-level languages for the manipulation of mixed data can be mapped to the algebra obtained as an instantiation of our abstract model. In particular, we consider SQL/XML, XQuery, and a proprietary approach based on an object/relational system.

6.1 Algebraic Representation of SQL/XML Queries

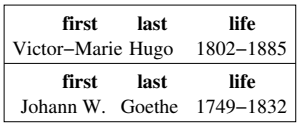
SQL/XML is the standard extension of SQL to extract XML data from relational tables, and it is used by the Oracle and DB2 database systems [2]. Consider the relations EMP and DEP represented in Fig. 15.

Example 23 *The SQL part of the input query is translated as usual: all relations in the FROM clause are joined together, and the WHERE clause is mapped to one or more selections. Then, the target list is mapped to a projection, and AS instructions are translated to renamings. The following query outputs, for all employees, the name and address of its department. The query uses a theta-join, and column names are considered different if they belong to different relations – EMP.name is different from DEP.name. The following queries have been automatically translated by a program written in Java, using JavaCC.*

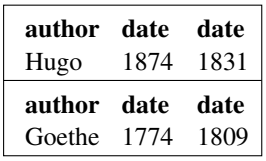
```
select EMP.name, surname, DEP.name AS dept, DEP.address
```



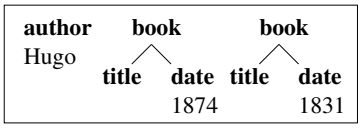
(a) B



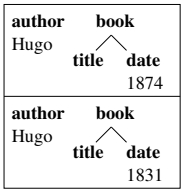
(b) A



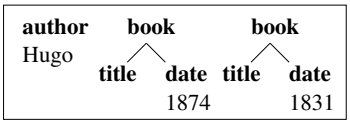
(c) $\pi_{+\text{author;book/date}}(\text{B})$



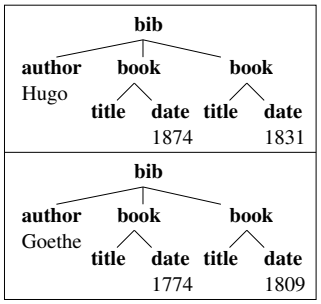
(d) $\text{S} = \sigma_{\text{author}='\text{Hugo}'}(\text{B})$



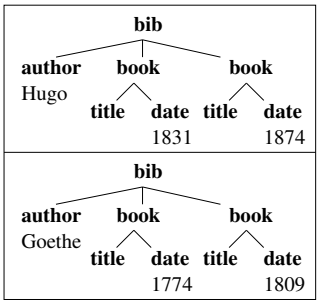
(e) $\text{S}' = \varsigma_{\text{author}}(\text{S})$



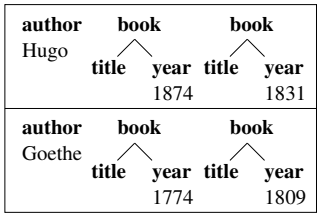
(f) $\gamma_{\text{author}}(\text{S}')$



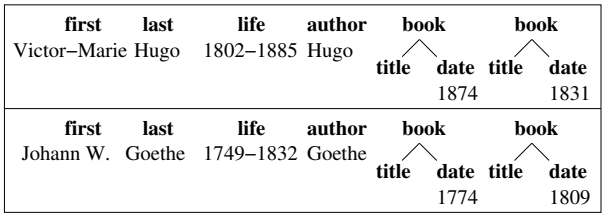
(g) $\text{E} = \epsilon_{<\text{bib}>(*)}(\text{B})$



(h) $\omega_{\text{bib;book/date}}(\text{E})$



(i) $\rho_{\text{year}\leftarrow\text{book/date}}(\text{B})$



(j) $\text{A} \bowtie_{\text{last}=\text{author}} \text{B}$

Fig. 13. Examples of algebraic operators. Labeled nodes are represented in bold font, while text nodes are in normal font.

```
from EMP,DEP
where salary = 2000 and EMP.dep = DEP.name;
```

$\rho_{\text{department}\leftarrow\text{DEP.name}}(\pi_{+\text{EMP.name,surname,DEP.name,DEP.address}}($
$\sigma_{\text{EMP.dep}/*=\text{DEP.name}/*}(\sigma_{\text{salary}/*='2000'}((\text{EMP}) \bowtie_{\text{true}} (\text{DEP})))))$

```
CompilationUnit ::= SelectStatement ";"
TableColumn ::= Name ( "." Name )?
Name ::= ( <S_IDENTIFIER> | <S_QUOTED_IDENTIFIER> )
TableReference ::= Name
SelectStatement ::= "SELECT" SelectList FromClause
                    ( WhereClause )? ( GroupByClause )?
SelectList ::= ( "*" | SelectItem ( "," SelectItem )* )
SelectItem ::= (SQLPrimaryExpression|XMLElement|XMLAgg)(As)?
As ::= "AS" <S_IDENTIFIER>
FromClause ::= "FROM" TableReference ( "," TableReference )*
WhereClause ::= "WHERE" SQLExpression
GroupByClause ::= "GROUP BY" TableColumn ( "," TableColumn )*
SQLExpression ::= SQLComparisonExpression
                  ( "AND" SQLComparisonExpression )*
SQLComparisonExpression ::= SQLPrimaryExpression <Relop>
                            SQLPrimaryExpression
SQLPrimaryExpression::= (TableColumn | <S_NUMBER> |
                         <S_CHAR_LITERAL>)
XMLElement ::= "XMLELEMENT(NAME" <S_QUOTED_IDENTIFIER> ","
               ElementContent ( "," ElementContent )* ")"
ElementContent ::= (Name|XMLElement|XMLAgg|<S_CHAR_LITERAL>)
XMLAgg ::= "XMLAGG(" XMLElement ( "," XMLElement )* ")"
```

Fig. 14. Grammar of the SQL/XML-like language that can be translated to our algebra.

Id	**Name**	**Surname**	**Dep**	**Salary**
emp01	John	Foo	Sales	20000
emp02	Jean	Toto	Sales	18000
emp03	Gianni	Pippo	PR	15000

Name	**Address**
Sales	Queen's Road
PR	Queen's Gate

Fig. 15. Relations EMP and DEP, containing data about employees and departments.

Result
Employee <id>emp01</id> is <name>John</name><surname>Foo</surname>
Employee <id>emp02</id> is <name>Jean</name><surname>Toto</surname>
Employee <id>emp03</id> is <name>Gianni</name><surname>Pippo</surname>

Fig. 16. Result of Example 24.

Example 24 *The following query uses XMLELEMENT functions and literals to produce a human-readable description of the employees. XMLELEMENT functions correspond to an embedding, followed by a projection. The embedding creates the new element with the children specified in the content of the XMLELEMENT function. These children can be other XML elements, literals, or column references. The new element is itself embedded inside a temporary node, that will be used as a reference to it. Then, a projection deletes temporary and column nodes, so that their children become direct descendants of the new element. Its result is represented in Fig. 16.*

```
select XMLELEMENT(NAME "emp",
          'Employee ',
          XMLELEMENT(NAME "id", idemp),
          ' is ',
          XMLELEMENT(NAME "name", name),
          XMLELEMENT(NAME "surname", surname))
       AS result
from EMP;
```

$$\rho_{\text{result}\leftarrow\$0}(\pi_{+\$0}(\pi_{-\$0/\text{emp}/*}(\epsilon_{<\$0>(<\text{emp}>(<\$1>(\text{'Employee '}),\$2,<\$3>(\text{' is '}),\$4,\$5))}($$
$$\pi_{-\$5/\text{surname}/*}(\epsilon_{<\$5>(<\text{surname}>(+\text{surname}))}(\pi_{-\$4/\text{name}/*}(\epsilon_{<\$4>(<\text{name}>(+\text{name}))}($$
$$\pi_{-\$2/\text{id}/*}(\epsilon_{<\$2>(<\text{id}>(+\text{idemp}))}(\text{EMP}))))))))))$$

Example 25 *The final example shows how the XMLAGG function is mapped to our algebra. XMLAGG functions and their contents are processed before external XMLELEMENT functions. Then, for all tuples, the resulting XML content is embedded inside temporary nodes, one for each XMLAGG function. At this point, tuples are grouped together on the columns specified in the GROUP BY clause. Finally, all the other (external) XMLELEMENT functions can be translated, as shown in the previous example. When some XMLAGG nodes appear as content of an XMLELEMENT function, we use a reference to the corresponding temporary node previously created. Its result is illustrated in Fig 17.*

Result
<dep><name>Sales</name> <emp>emp01</emp> <emp>emp02</emp> </dep>
<dep><name>PR</name> <emp>emp03</emp> </dep>

Fig. 17. Result of Example 25.

```
SELECT XMLELEMENT(NAME "dep",
            XMLELEMENT(NAME "name", dep),
            XMLAGG(XMLELEMENT(NAME "emp", id))
        ) AS result
FROM EMP
GROUP BY dep;
```

$$\rho_{\text{result}\leftarrow\$2}(\pi_{+\$2}(\pi_{-\$2/\text{dep}/*}(\epsilon_{<\$2>(<\text{dep}>(\$3,\$0))}(\pi_{-\$3/\text{name}/*}(\epsilon_{<\$3>(<\text{name}>(^{+}\text{dep}))}($$
$$\gamma_{\text{dep}}(\pi_{-\$0/*}(\epsilon_{<\$0>(\$1)}(\pi_{-\$1/\text{emp}/*}(\epsilon_{<\$1>(<\text{emp}>(^{+}\text{id}))}(\text{EMP}))))))))))$$

6.2 XQuery

The subset of XQuery that we have mapped to our algebra includes its main functionalities, i.e., path expressions, For-Let-Where-Return expressions, constructors, and arbitrarily nested sub-queries. The flexibility of the model and the operators for embedding, splitting and grouping are sufficient to allow unconstrained nesting of XQuery-like expressions inside For, Let, Where and Return clauses, and also inside element constructors. On the other hand, it has many limitations, which in our opinion are not relevant to this discussion and would compromise the clarity of the explanation. However, we briefly cite the main limitations. Path expressions are very simple, but additional features could be added painlessly to the companion language. We have not included duplicates, as our model is based on sets. However, to avoid the elimination of duplicate XML fragments with different node identifiers, we can represent the identifiers explicitly as nodes with a reserved type. Order is represented inside data trees. The choice of not representing it between data graphs is to enhance the clarity of the presentation, and it is not a characterizing feature of our model. However, it should be noticed that the physical optimization of ordered data is very hard to achieve, and order between data graphs is not essential because

we do represent order inside data trees. Finally, we have not considered generic functions: they may be implemented outside the algebraic representation of the language, as happens in the relational context. The grammar of the XQuery-like language under consideration is illustrated in Fig. 18.

6.3 Algebraic Representation of XQuery Expressions

The following queries have been automatically translated by a program written in Java, using JavaCC. To abbreviate the resulting algebraic expressions, we will represent with the symbol ϵ^- an embedding of subexpressions followed by a deletion of temporary nodes. For instance, we will write:

$$\epsilon^-_{<\mathrm{TagQName}>(\mathrm{Var}_1;\ldots;\mathrm{Var}_n)}(B)$$

meaning that we embed $\mathrm{Var}_1;\ldots;\mathrm{Var}_n$ inside TagQName and erase the nodes $\mathrm{Var}_1;\ldots;\mathrm{Var}_n$. We will also use an outer join and a cross product operator, which have not been defined in previous sections and have their usual semantics.

Example 26 (Input function call with path expression)

```
collection('coll')//author
```

$\pi_{+*//\mathrm{author}}(\mathrm{'coll'})$

Example 27 (Nested constructors)

```
<greetings>Hello <planet>World</planet></greetings>
```

$\epsilon^-_{<\mathrm{greetings}>(\mathrm{'Hello\ ',\$0})}(\epsilon_{<\$0>(*)}(\epsilon_{<\mathrm{planet}>(\mathrm{'World'})}([\,])))$

Example 28 (Literal)

```
'hello'
```

$\epsilon_{\mathrm{'hello'}}([\,])$

Example 29 (For-Return expression)

```
for $b in collection('books')
for $a in $b//author
return $a
```

$\gamma(\pi_{+\overline{\$a,\$b}}(\pi_{+\$a;\$b;\$a/*}(\epsilon_{<\$a>(\overline{\$b})}(\varsigma_{\$b}(\pi_{+\$b;\$b/*//\mathrm{author}}(\epsilon_{<\$b>(*)}(\varsigma(\mathrm{'books'}))))))))$

Example 30 (For-Let-Return expression)

```
for $b in collection('books')
let $a := $b//author
return <authors>{$a}</authors>
```

```
Expr ::= InputExpr | FLWORExpr |
         Literal | Constructor
InputExpr ::= (InputFunctionCall | VarRef)(PathExpr)?
InputFunctionCall ::= ( "doc(" | "collection(" )
                      <StringLiteral> ")"
VarRef ::= <VarName>
PathExpr ::= ( ChildStep | DescendantStep )
             ( ChildStep | DescendantStep )*
ChildStep ::= "/" NameTest
DescendantStep ::= "//" NameTest
NameTest ::= <QName> | Wildcard
Wildcard ::= "*" | <NCName> ":*" | "*:" <NCName>
FLWRExpr ::=  (ForClause | LetClause)
              (WhereClause)? "return" Expr
ForClause ::= "for" <VarName> "in" Expr
LetClause ::= "let" <VarName> ":=" Expr
WhereClause ::= "where" Expr <CompOp> Expr
Constructor ::= "<" <TagQName> ("/>" |
                (">" ElementContent*
                 "<" <TagQName> ">"))
ElementContent ::= <ElementContentChar> |
                   Constructor | EnclosedExpr
Literal ::= NumericLiteral | <StringLiteral>
NumericLiteral ::= <IntegerLiteral> |
                   <DecimalLiteral> |
                   <DoubleLiteral>
EnclosedExpr ::= "{" Expr "}"
```

Fig. 18. Grammar of the XQuery-like language that can be translated into our algebra. The syntax of <QName>, <IntegerLiteral>, etc. is the same as that described in the XQuery specification [3].

$\gamma(\pi_{+\overline{\$a,\$b}}(\epsilon^{-}_{<\text{authors}>(\$0)}(\epsilon_{<\$0>(\overline{\$b,\$a})}(\pi_{+\$a;\$b;\$a/*}($
$\epsilon_{<\$a>(\overline{\$b})}(\pi_{+\$b;\$b/*//\text{author}}(\epsilon_{<\$b>(*)}(\varsigma('\text{books}'))))))))))$

Example 31 (FLWR expression)

```
for $b in collection('books')
let $a := $b//author
where $b/title = 'Moby Dick'
return <authors>{$a}</authors>
```

$\gamma(\pi_{+\overline{\$a,\$b}}(\epsilon^{-}_{<\text{authors}>(\$0)}(\epsilon_{<\$0>(\overline{\$b,\$a})}(\pi_{+\$a;\$b;\$a/*}(\sigma_{\$b/*/\text{title}/*='\text{Moby Dick}'}($
$\epsilon_{<\$a>(\overline{\$b})}(\pi_{+\$b;\$b/*//\text{author}}(\epsilon_{<\$b>(*)}(\varsigma('\text{books}'))))))))))$

Example 32 (FLWR expression, constructor and nested variable reference)

```
for $b in collection('books')
let $a := $b//author
where $b/title = 'Moby Dick'
return <info>The aut. are <aut>{$a}</aut></info>
```

$\gamma(\pi_{+\overline{\$a,\$b}}(\epsilon^{-}_{<\text{info}>('\text{The authors are }',\$1)}(\epsilon_{<\$1>(\overline{\$b,\$a})}(\epsilon^{-}_{<\text{authors}>(\$0)}($
$\epsilon_{<\$0>(\overline{\$b,\$a})}(\pi_{+\$a;\$b;\$a/*}(\sigma_{\$b/*/\text{title}/*='\text{Moby Dick}'}(\epsilon_{<\$a>(\overline{\$b})}(\pi_{+\$b;\$b/*//\text{author}}($
$\epsilon_{<\$b>(*)}(\varsigma('\text{books}'))))))))))))$

Example 33 (Nested FLWR expression)

```
for $b in collection('books')
return
<book>
   {for $a in $b/author
    return <name>{$a/name}</name>}
</book>
```

$\gamma(\pi_{+\overline{\$b}}(\epsilon^{-}_{<\text{book}>('\hookleftarrow\ ',\$1,'\hookleftarrow')}(\epsilon_{<\$1>(\overline{\$b})}(\gamma_{\$b}(\pi_{+\overline{\$a}}(\epsilon^{-}_{<\text{name}>(\$0)}($
$\epsilon_{<\$0>(\overline{\$b,\$a})}(\pi_{+\$a;\$b;\$a/*/\text{name}}(\epsilon_{<\$a>(\overline{\$b})}(\varsigma_{\$b}(\pi_{+\$b;\$b/*/\text{author}}(\epsilon_{<\$b>(*)}(\varsigma('\text{books}'))))))))))))))$

Example 34 (Multiple Parallel Nesting)

```
for $a in collection('Authors')
return
<author>
   $a/name
   {for $b in collection('Books')
   where $b/author/* = $a/id/*
   return $b/title}
   {for $j in collection('Journals')
   where $j/author/* = $a/id/*
   return $j/title}
</author>
```

$\gamma(\pi_{+\overline{\$a}}(\epsilon^{-}_{<\text{author}>('\hookleftarrow \quad \$a/\text{name}\hookleftarrow \quad ',\$0,'\hookleftarrow \quad ',\$1,'\hookleftarrow')}(\epsilon_{<\$1>(\overline{\$a,\$0})}($
$(\text{B3}) =\!\bowtie (\gamma_{\$a,\$0}(\pi_{+\overline{\$j}}(\pi_{+\$j;\$0;\$a;\$j/*/\text{title}}(\sigma_{\$j/*/\text{author}/*=\$a/*/\text{id}/*}(\epsilon_{<\$j>(\overline{\$a,\$0,\$0})}(\varsigma_{\$a,\$0}($
$'\text{Journals}' \times \epsilon_{<\$0>(\overline{\$a})}((\text{B2}) =\!\bowtie (\gamma_{\$a}(\pi_{+\overline{\$b}}(\pi_{+\$b;\$a;\$b/*/\text{title}}(\sigma_{\$b/*/\text{author}/*=\$a/*/\text{id}/*}($
$\epsilon_{<\$b>(\overline{\$a})}(\varsigma_{\$a}('\text{Books}' \times \epsilon_{<\$a>(*)}(\varsigma('\text{Authors}'))))))))))))))))))))$

B2 and B3 are the tables of bindings before the execution of the two nested FLWR expressions.

Example 35 (Query rewriting) *As a final example, we show how some general equivalences presented in Section 3.3 can be applied after the definition of a specific intermediate logical level.*

```
for $b in collection('books')
let $c := $b//chapter
where $b/title = 'XQuery'
return $c/title
```

The algebraic translation of this query is:
$\gamma(\pi_{+\overline{\$c,\$b}}(\pi_{+\$c;\$b;\$c/*/\text{title}}(\sigma_{\$b/*/\text{title}='\text{XQuery}'}($
$\epsilon_{<\$c>(\overline{\$b})}(\pi_{+\$b;\$b/*//\text{chapter}}(\epsilon_{<\$b>(*)}(\varsigma('\text{books}')))))))))$
We may rewrite it by using Propositions 12, 10, and again Proposition 12:

$$\sigma_{\$b/*/\text{title}='\text{XQuery}'}(\epsilon_{<\$c>(\overline{\$b})}(.)) = \epsilon_{<\$c>(\overline{\$b})}(\sigma_{\$b/*/\text{title}='\text{XQuery}'}(.))$$
$$\sigma_{\$b/*/\text{title}='\text{XQuery}'}(\pi_{+\$b;\$b/*//\text{chapter}}(.)) = \pi_{+\$b;\$b/*//\text{chapter}}(\sigma_{\$b/*/\text{title}='\text{XQuery}'}(.))$$
$$\sigma_{\$b/*/\text{title}='\text{XQuery}'}(\epsilon_{<\$b>(*)}(.)) = \epsilon_{<\$b>(*)}(\sigma_{*/\text{title}='\text{XQuery}'}(.)) \ .$$

Finally, we keep together all consecutive projections and embeddings, obtaining the following equivalent query:

$$\pi_{+\overline{\$c,\$b}}(\pi_{+\$b,\$c;\$c/*/\text{title}}(\epsilon_{<\$c>(\overline{\$b})}(\pi_{+\$b;\$b/*//\text{chapter}}(\epsilon_{<\$b>(*)}(\text{T2})))))$$
$$\text{T2} = \sigma_{*/\text{title}='\text{XQuery}'}(\text{T1})$$
$$\text{T1} = \varsigma('\text{books}') \ .$$

This last example shows two potential improvements of the query execution plan under concern. First, as in the relational context, anticipating selections may be crucial. In this case, we do not need to evaluate the path expression $\$b//\text{chapter}$ and to bind $\$c$ for books that do not satisfy the selection condition. Second, consecutive lists of embeddings and projections allow us to automatically identify lists of operators which can be computed in a single step [8] . In the current example, there would be no need to build tables of bindings or secondary memory variables. However, as

[8] This is a totally reasonable assumption for embeddings and projections, and it can also be extended to other operators – this point is not addressed in the paper.

we do not make assumptions on the physical organization of data, and this paper focuses on the logical level, we do not delve into this subject.

As a final remark, notice that the translation of XQuery-like expressions can generate a lot of algebraic operators. As the preceding example has shown, this does not mean that the corresponding algebraic queries are complex. Many operators can be grouped together, following simple aggregation patterns (for example, all consecutive embeddings and projections). At the same time, the presence of many elementary operators allows a fine-grained control over query execution plans.

6.4 Algebraic Representation of XML Type Methods

Let DEPS_AND_EMPS and DEP_INFO be the result of the following Oracle10*g* SQL statements, corresponding to the creation and population of the relations represented in Figs. 19 and 20:

```
create table DEPS_AND_EMPS
       (XDATA sys.XMLTYPE);
create table DEP_INFO
       (Name VARCHAR2(10), Info VARCHAR2(80));

insert into DEPS_AND_EMPS(XDATA) values (
 sys.XMLTYPE.createXML(
 ' <dep>
      <id>0001</id>
      <name>Sales</name>
      <emp>emp01</emp>
      <emp>emp02</emp>
   </dep>
 '));
insert into DEPS_AND_EMPS(XDATA) values (
 sys.XMLTYPE.createXML(
 ' <dep>
      <id>0002</id>
      <name>PR</name>
      <emp>emp03</emp>
   </dep>
 '));

insert into DEP_INFO values
  ('Sales', 'Info on sales department.');
insert into DEP_INFO values
  ('PR', 'Info on PR department.');
insert into DEP_INFO values
  ('Production', 'Info on production department.');
```

XDATA
<dep> <id>0001</id> <name>Sales</name> <emp>emp01</emp> <emp>emp02</emp> </dep>
<dep> <id>0002</id> <name>PR</name> <emp>emp03</emp> </dep>

Fig. 19. Relation DEPS_AND_EMPS.

Name	Info
Sales	Info on sales department.
PR	Info on PR department.
Production	Info on production department.

Fig. 20. Relation DEP_INFO.

As we have shown in the Example 17, these tables can be easily represented in our model. Now, consider the following queries, that use XML Type methods. The former extracts all XML fragments in column XDATA whose element `id` child of a root element `dep` has string value `0001`. The latter performs a join between two mixed relations.

```
select D.XDATA.extract('/dep/name').getClobVal()
from DEPS_AND_EMPS D
where D.XDATA.extract('/dep/id/text()')
             .getStringVal() = '0001';

select D.XDATA, I.Info
from DEPS_AND_EMPS D, DEP_INFO I
where D.XDATA.extract('/dep/name/text()')
             .getStringVal() = I.Name;
```

Our algebra can represent these queries in the following way:

$$\pi_{\mathrm{XDATA/dep/name}}(\sigma_{\mathrm{XDATA/dep/id='0001'}}(\mathrm{DEPS\_AND\_EMPS})) \tag{10}$$

$$\pi_{\mathrm{XDATA,Info}}(\mathrm{DEPS\_AND\_EMPS} \bowtie_{\mathrm{XDATA/dep/name=Name}} \mathrm{DEP\_INFO}) \tag{11}$$

XML and relational/XML databases are only two examples of the applicability of our model. With the availability of specific languages for the manipulation of atoms, which constitute the third abstraction layer of our model, we can expect to use our algebra to represent queries manipulating several kinds of heterogeneous objects in a homogeneous way.

7 Related Work

The model generated using our approach and presented in Section 5 is based on trees. Using a relational terminology, trees are not used to represent an entire database, single relations, or tuples, as in other recent works reviewed in this section. In our model, tuples are forests, so trees are the components of tuples. Tree and graph-based data models were the first to be studied by database researchers and industry (hierarchical and network models). Then, they have been superseded by the relational and subsequently object/relational ones. This was mainly due to the absence of abstraction, not to their structural features. In fact, many different hierarchical models have been recently proposed, in different contexts. We briefly review some of them.

To overcome the limitations of the relational model, at the end of the past century the database community studied semistructured data, and provided the first theories and tools to manage them [15–17]. A revisited hierarchical model to query network directories has been defined as an extension of the LDAP query language, but it is very specific and difficult to adapt to the context of general data management [18]. The Oracle corporation has released an API to manage folder-based heterogeneous data [19]. However, this is only an interface, without a corresponding formal model – it is still based on a relational storage system.

The best known example of hierarchical data is certainly XML. The popularity of this language has appealed to a lot of researchers. For this reason, a great number of algebras for hierarchical data can be found in this context [6–9,20]. The World Wide Web Consortium has recently published the data model and the formal semantics of XQuery [21,22]. However, the concepts described in these documents have not been proposed to optimize query execution, and they have not been used with this aim in existing implementations.

The most well known algebra for XML data is TAX (Tree Algebra for XML), which does not use tables but provides a tree-based model [6]. The main innova-

tion of TAX is the concept of *pattern tree*, which is a way of matching parts of XML trees. The aim of pattern trees is the same as that of path expressions, which can be written using XPath, and have been introduced in the LOREL language for semistructured data [11]. While TAX implicitly suggests that relational-like operators can be used in several different contexts, it redefines them using pattern trees. In contrast, our operators do not depend on a specific language for the manipulation of trees. Another difference between the model presented in Section 5 and TAX is that we use data graphs where TAX uses single trees. This allows our selection and projection to be orthogonal. Finally, and most importantly, an algorithm to translate XQuery expressions to TAX queries has been provided, but it can only be applied to expressions with single nesting inside the Return clause. Our algebra allows unconstrained nesting of subexpressions, which is a central feature of XQuery [23].

A different proposal is NAL, a nested algebra coming from object oriented database theory [9]. In this work, the authors propose algebraic equivalences for query unnesting and they show enormous performance improvements by unnesting some example queries. However, this work has three significant limitations. First, as the authors say, the equivalences assume that the nested queries do not construct XML fragments. This is a strong limitation, not shared by our algebra. Additionally, the algebra is nested, i.e., we can recursively include algebraic expressions inside predicates, possibly leading to very complex expressions. In contrast, our algebra is simple, although it has no limitations on nesting and XML fragment construction. Finally, it is noteworthy than the performance improvements obtained seem not to be related to the unnesting equivalences, as claimed by the authors, but to an ad hoc feature of the examples, where the naive execution reads the same document many times from secondary memory.

For the sake of completeness, we conclude mentioning some of the main general data models defined in the context of data integration [24–30]. These models have not been proposed with the aim of defining new context-dependent models. In contrast, our approach concerns data representation and manipulation, and not model transformation and integration. Therefore, a direct comparison is not applicable.

8 Final Remarks and Open Issues

In this paper, we have presented an approach for the generation of data models. The main motivation of this work is the high diversity in existing digital data, and the consideration that every different kind of data and even every combination of heterogeneous data can be best managed using a specialized model that takes advantage of its specificities. An abstract model allows us to describe general data manipulation functionalities, that do not have to be reinvented every time a new model is proposed.

We have shown that the relational, nested relational, and TAX models can be obtained as specializations of ours. Moreover, we have generated a new model for mixed XML and relational data, that can be used as a unifying representation and manipulation level for the main existing approaches used in real systems. We have also presented mappings from XQuery, SQL/XML and Oracle's XML Type methods to the algebra generated using our abstract model. This algebra can represent, among the others, nested XQuery expressions with no constraints on nesting and on node constructors, and inherits general algebraic equivalences that can be used to perform logical query rewriting.

The approach for the generation of new models defined in this paper has been presented from a theoretical point of view. In practice, it can be used just to define new models, as in Section 5, or possibly in the design of software systems whose architecture reflects our three layers. These systems would inherit the benefits of our approach, in particular the ability to change their logical models in an easy and scalable way. However, implementing theoretical models often presents new practical issues, that will be objects of further research.

In particular, we have not discussed the relationship of our abstract model with the other two abstraction levels of relational database systems, i.e., the external and physical levels. While it seems easy to define views over collections, because views are expressed as queries, it should be investigated whether it is possible and reasonable to define three analogous abstraction layers at the physical level, corresponding to those defined in this paper.

As a final remark, it should be noted that we have focused on very simple atoms and languages for atom manipulation. The integration of very complex atoms, like those found in multimedia systems, will probably lead to an extension of our model, where concepts like *ranking* and *approximate queries* are also generalized at the external logical level.

A Proofs of algebraic equivalences

Proof of Proposition 9 By definition of σ,

$$\sigma_{\mathrm{P}_1}(\sigma_{\mathrm{P}_2}(C)) = \{c \mid c \in \sigma_{\mathrm{P}_2}(C) \wedge \mathrm{P}_1(c)\}$$

As $\sigma_{\mathrm{P}_2}(C) = \{c \mid c \in C \wedge \mathrm{P}_2(c)\}$, we obtain by substitution:

$$c \in \sigma_{\mathrm{P}_1}(\sigma_{\mathrm{P}_2}(C)) \Leftrightarrow c \in \{c \mid \underbrace{c \in C \wedge \mathrm{P}_2(c)}_{c \in \sigma_{\mathrm{P}_2}(C)} \wedge \mathrm{P}_1(c)\}$$

This is the definition of $\sigma_{\mathrm{P}_1 \wedge \mathrm{P}_2}(C)$. Therefore, $\sigma_{\mathrm{P}_1}(\sigma_{\mathrm{P}_2}(C)) = \sigma_{\mathrm{P}_1 \wedge \mathrm{P}_2}(C)$.

Proof of Proposition 10 By definition of σ,

$$\sigma_{\mathrm{P}}(\pi_{\mathrm{PF}}(C)) = \{c \mid c \in \pi_{\mathrm{PF}}(C) \wedge \mathrm{P}(c)\}$$

By substituting $\pi_{\mathrm{PF}}(C)$, we can state that:

$$\sigma_{\mathrm{P}}(\pi_{\mathrm{PF}}(C)) = \{c \mid \underbrace{c' \in C \wedge c = \mathrm{PF}(c')}_{c \in \pi_{\mathrm{PF}}(C)} \wedge \mathrm{P}(c)\}$$

Notice that $c = \mathrm{PF}(c')$. We may thus rewrite the equivalence as:

$$\sigma_{\mathrm{P}}(\pi_{\mathrm{PF}}(C)) = \{c \mid c' \in C \wedge c = \mathrm{PF}(c') \wedge \underbrace{\mathrm{P}(\mathrm{PF}(c'))}_{\text{As } c=\mathrm{PF}(c')}\}$$

The proposition assumes that $\forall c \in C\ (P'(c) = \mathrm{P}(\mathrm{PF}(c)))$, so we obtain:

$$\sigma_{\mathrm{P}}(\epsilon_{\mathrm{PF}}(C)) = \{\mathrm{PF}(c') \mid c' \in C \wedge \underbrace{\mathrm{P}'(c')}_{\mathrm{P}'=\mathrm{P}(\mathrm{PF}(c'))}\}$$

Now, the generic element of $\sigma_{\mathrm{P}'}(C)$ can be written as: $c' \in C \wedge \mathrm{P}'(c')$. Therefore,

$$\sigma_{\mathrm{P}}(\epsilon_{\mathrm{PF}}(C)) = \{\mathrm{PF}(c') \mid \underbrace{c' \in \sigma_{\mathrm{P}'}(C)}_{c' \in C \wedge \mathrm{P}'(c')}\}$$

This is the definition of $\pi_{\mathrm{PF}}(\sigma_{\mathrm{P}'}(C))$, therefore: $\sigma_{\mathrm{P}}(\pi_{\mathrm{PF}}(C)) = \pi_{\mathrm{PF}}(\sigma_{\mathrm{P}'}(C))$.

Proof of Proposition 11 By definition of σ,

$$\sigma_{\mathrm{P}}(C_1 \bowtie_{\mathrm{P}'} C_2) = \{c \mid c \in (C_1 \bowtie_{\mathrm{P}'} C_2) \wedge \mathrm{P}(c)\}$$

We may substitute $c \in (C_1 \bowtie_{\mathrm{P}'} C_2)$ using the definition of join, obtaining:

$$\sigma_{\mathrm{P}}(C_1 \bowtie_{\mathrm{P}'} C_2) = \{c \mid \underbrace{c' \in C_1 \wedge c'' \in C_2 \wedge \mathrm{P}'(c', c'') \wedge c = c' \cup c''}_{c \in (C_1 \bowtie_{\mathrm{P}'} C_2)} \wedge \mathrm{P}(c)\}$$

Notice that $c = c' \cup c''$. We may thus rewrite $P(c)$ as $P(c' \cup c'')$. Moreover, by assumption, $P(c' \cup c'') = P(c'')$. This means that the selection predicate only concerns the part of every object derived from C_2. We obtain:

$$\sigma_{\mathrm{P}}(C_1 \bowtie_{\mathrm{P}'} C_2) = \{c \mid c' \in C_1 \wedge c'' \in C_2 \wedge \mathrm{P}'(c', c'') \wedge c = c' \cup c'' \wedge\ \mathrm{P}(c'')\}$$

Finally, as $c'' \in C_2 \wedge \mathrm{P}(c'')$ is the generic element of $\sigma_{\mathrm{P}}(C_2)$, we can write the previous equivalence in the following way:

$$\sigma_{\mathrm{P}}(C_1 \bowtie_{\mathrm{P}'} C_2) = \{c' \cup c'' \mid c' \in C_1 \wedge \underbrace{c'' \in \sigma_{\mathrm{P}}(C_2)}_{c'' \in C_2\ \wedge \mathrm{P}(c'')} \wedge \mathrm{P}(c', c'')\}$$

which is the definition of $C_1 \bowtie_{\mathrm{P}'} \sigma_{\mathrm{P}}(C_2)$. Therefore, $\sigma_{\mathrm{P}}(C_1 \bowtie_{\mathrm{P}'} C_2) = C_1 \bowtie_{\mathrm{P}'} \sigma_{\mathrm{P}}(C_2)$.

Proof of Proposition 12 Analogously to the proof of Proposition 10.

Proof of Proposition 13 By definition of σ,

$$\sigma_{\mathrm{P}}(\varsigma(C)) = \{c \mid c \in \varsigma(C) \wedge \mathrm{P}(c)\}$$

The substitution of $\varsigma(C)$ with its definition leads to:

$$\sigma_{\mathrm{P}}(\varsigma(C)) = \{c \mid \underbrace{c' \in C \wedge e \in c' \wedge c = \{e\}}_{c \in \varsigma(C)} \wedge \mathrm{P}(\{e\})\}$$

By assumption, $\mathrm{PF}(c) = \{e \mid e \in c \wedge \mathrm{P}(\{e\})\}$. Therefore, we can substitute the condition $e \in c' \wedge \mathrm{P}(\{e\})$ with the condition $e \in \mathrm{PF}(c')$. We obtain:

$$\sigma_{\mathrm{P}}(\varsigma(C)) = \{\{e\} \mid c' \in C \wedge e \in \mathrm{PF}(c')\}$$

After a simple substitution,

$$\sigma_{\mathrm{P}}(\varsigma(C)) = \{\{e\} \mid c' \in C \wedge e \in c \wedge c = \mathrm{PF}(c')\}$$

$\{c \mid c' \in C \wedge c = \mathrm{PF}(c')\}$ is the generic element of $\pi_{\mathrm{PF}}(C)$, therefore,

$$\sigma_{\mathrm{P}}(\varsigma(C)) = \{\{e\} \mid c \in \pi_{\mathrm{PF}}(C) \wedge e \in c\}$$

By definition of ς, we conclude that $\sigma_{\mathrm{P}}(\varsigma(C)) = \varsigma(\pi_{\mathrm{PF}}(C))$.

References

[1] Gemmel, J., Bell, G., Lueder, R., Drucker, S., Wong, C.: MyLifeBits: Fulfilling the memex vision. In: Multimedia'02, Juan-les-Pins, France, ACM (2002) 235–238

[2] Chaudhri, A.B., Rashid, A., Zicari, R., eds.: XML Data Management: Native XML and XML-Enabled Database Systems. Addison Wesley Professional (2003)

[3] Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML query language. Technical report, W3C (2003)

[4] Clark, J.: XSL transformations (XSLT) version 1.0. Technical report, W3C (1999) http://www.w3.org/TR/xslt.

[5] Clark, J., DeRose, S.: XML path language (XPath) version 1.0. Technical report, W3C (1998)

[6] Jagadish, H., Lakshmanan, L., Srivastava, D., Thompson, K.: TAX: A tree algebra for XML. In: International Workshop on Database Programming Languages (DBPL). (2001)

[7] Frasincar, F., Houben, G.J., Pau, C.: XAL: an algebra for XML query optimization. In: Proceedings of the thirteenth Australasian conference on Database technologies, Australian Computer Society, Inc. (2002) 49–56

[8] Sartiani, C., Albano, A.: Yet another query algebra for XML data. In: International Database Engineering and Applications Symposium (IDEAS), Edmonton, Canada (2002)

[9] May, N., Helmer, S., Moerkotte, G.: Nested queries and quantifiers in an ordered context. In: 20th International Conference on Data Engineering (ICDE). (2004) 239–250

[10] Magnani, M., Montesi, D.: XML and relational data: towards a common model and algebra. In: International Database Engineering and Applications Symposium (IDEAS). (2005)

[11] Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. International Journal on Digital Libraries **1** (1997) 68–88

[12] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc. (1995)

[13] Roth, M.A., Korth, H.F., Silberschatz, A.: Extended algebra and calculus for nested relational databases. ACM Transactions on Database Systems **13** (1988) 389–417

[14] Malhotra, A., Melton, J., Walsh, N.: XQuery 1.0 and XPath 2.0 functions and operators. Technical report, W3C (2003)

[15] Abiteboul, S.: Querying semi-structured data. In: International Conference on Data Base Theory (ICDT), Delphi, Greece (1997) 1–18

[16] Buneman, P.: Semistructured data. In: ACM Symposium on Principles of Database Systems (PODS), Tucson, Arizona (1997) 117–121

[17] Suciu, D.: Semistructured data and XML. In: International Conference on Foundations of Data Organization (FODO), Kobe, Japan (1998)

[18] Jagadish, H.V., Lakshmanan, L.V.S., Milo, T., Srivastava, D., Vista, D.: Querying network directories. In: ACM SIGMOD Conference, ACM Press (1999) 133–144

[19] Azriel, S.: Oracle content management SDK: Concepts and architecture. White Paper (2003) http://otn.oracle.com/products/ifs/pdf/OracleCMSDKArchitectureWP.pdf.

[20] Beeri, C., Tzaban, Y.: SAL: An algebra for semistructured data and XML. In: ACM SIGMOD Workshop on The Web and Databases (WebDB), Philadelphia, Pennsylvania (1999) 37–42

[21] Mary Fernández, A.M., Marsh, J., Nagy, M., Walsh, N.: XQuery 1.0 and XPath 2.0 data model. Technical report, W3C (2003)

[22] Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 formal semantics. Technical report, W3C (2004)

[23] Magnani, M., Montesi, D.: Mapping XQuery to algebraic expressions. Technical Report UBLCS-2004-11, University of Bologna (2004)

[24] Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: International Conference on Very Large Data Bases, Morgan Kaufmann (1996) 251–262

[25] Poulovassilis, A., McBrien, P.: A general framework for schema transformation. Data & Knowledge Engineering **28** (1998) 47–71

[26] Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A vision for management of complex models. SIGMOD Record **29** (2000) 55–63

[27] Castano, S., Antonellis, V.D., di Vimercati, S.D.C.: Global viewing of heterogeneous data sources. IEEE Transactions on Knowledge and Data Engineering **13** (2001) 277–297

[28] Bergamaschi, S., Castano, S., Vincini, M., Beneventano, D.: Semantic integration of heterogeneous information sources. Data & Knowledge Engineering **36** (2001) 215–249

[29] Calvanese, D., Giacomo, G.D., Lenzerini, M.: Description logics for information integration. In: Computational Logic (Kowalski Festschrift). Volume 2048 of LNAI., Berlin Heidelberg, Springer-Verlag (2002) 41–60

[30] Rosaci, D., Terracina, G., Ursino, D.: A framework for abstracting data sources having heterogeneous representation formats. Data & Knowledge Engineering **48** (2004) 1–38