Schema Advisor for Hybrid Relational-XML DBMS

Mirella M. Moro [*]
University of California
Riverside, CA, USA
mirella@cs.ucr.edu

Lipyeow Lim
IBM T.J. Watson Research Ctr.
Hawthorne, NY, USA
liplim@us.ibm.com

Yuan-Chi Chang
IBM T.J. Watson Research Ctr.
Hawthorne, NY, USA
yuanchi@us.ibm.com

ABSTRACT

In response to the widespread use of the XML format for document representation and message exchange, major database vendors support XML in terms of persistence, querying and indexing. Specifically, the recently released IBM DB2 9 (for Linux, Unix and Windows) is a hybrid data server with optimized management of both XML and relational data. With the new option of storing and querying XML in a relational DBMS, data architects face the the decision of what portion of their data to persist as XML and what portion as relational data. This problem has not been addressed yet and represents a serious need in the industry. Hence, this paper describes ReXSA, a schema advisor tool that is being prototyped for IBM DB2 9. ReXSA proposes candidate database schemas given an information model of the enterprise data. It has the advantage of considering qualitative properties of the information model such as reuse, evolution and performance profiles for deciding how to persist the data. Finally, we show the viability and practicality of ReXSA by applying it to custom and real usecases.

Categories and Subject Descriptors: H.2 [Database Management]: Logical Design – *Schema and subschema*

General Terms: Design

1. INTRODUCTION

It is no longer a conjecture that XML data and relational data will always co-exist and complement each other in enterprise data management. XML documents and messages pervade enterprise systems such that XML formats have been standardized for data storage and exchange in many industries. While much critical data are still in relational format, practitioners have increasingly turned to XML for storing data that do not fit into the relational model.

In the health care industry, for example, XML is widely used for sharing the metadata of medical records in backend repositories. In one real-world scenario, the schema for the metadata contains over 200 variations in order to support the diverse types of medical documents being persisted and queried. These 200 variant types have a shared common section and specific individual extensions. Persisting such metadata in relational format results in a large number of tables and poor performance. Moreover, adding a new type of document requires at least two weeks for re-engineering the relational schema to accommodate the new type.

In another example, XML is heavily employed in trading systems for representing financial products such as options and derivatives. New types of derivatives are invented every week, which in turn triggers weekly data model changes and hence schema changes. Again, using a relational format requires lengthy database schema changes and data migration, which consequently affect the agility of the business.

As those two real cases, enterprises in various industries have turned to XML for greater flexibility and easier maintenance, when compared to relational representation. On the other hand, much legacy data and transactional data remains highly rigid and well-suited for the relational model. It is clear that neither pure relational nor pure XML data management systems will suffice. Instead, a system that can deal with both types of data is much needed.

Commercial DBMS (Database Management System) vendors have long recognized the need for hybrid relational-XML systems. Hence, all major commercial DBMSs include support for persisting, querying and indexing XML data [18]. In particular, the recently released IBM DB2 9 (for Linux, Unix and Windows) is a hybrid data server with optimized management of both XML and relational data. While XML is now a first-class citizen in the DBMS, data architects are still unsure of how exactly to persist their data. It is still not trivial to decide what portion of the enterprise data to persist as XML and what portion as relational data.

In this paper, we describe ReXSA (*Relational-XML Schema Advisor*), a tool that addresses the challenge of designing hybrid database schemas. Given an annotated information model of the enterprise data, ReXSA evaluates it and recommends a database schema that harmonizes relational and XML data models. ReXSA fills a void in the currently available set of database tools. While tools for designing and generating pure relational schemas and pure XML schemas exist, there is currently no tool available to help data architects design database schemas for both relational and XML data. Moreover, ReXSA complements information modeling tools such as IBM's Rational Data Architect [15], since the information model produced by this tool can be fed as input into ReXSA.

[*]Research done while visiting IBM T J Watson Research Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'07, June 12–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

The problem of generating relational-XML schemas from an information model is particularly challenging for many reasons. First, while major database vendors support XML within their relational products, few industry and academic studies have ventured to suggest methodologies for such new design environment. Second, the parameters and requirements of the problem are many and diverse. Given a particular usage scenario, there are many ways to model the information needs and there are several ways to annotate the model. Similarly, the users (data architects) may also have different priorities for the schema. In some cases, flexibility is of paramount importance, in other cases, query performance is. Third, in general, the solutions are not unique because different relational-XML schemas can satisfy the same set of parameters and requirements.

Focusing on those challenges, ReXSA is able to generate different candidate schemas for the user to choose, then simplifying the whole process by avoiding the many options that create much confusion. In summary, ReXSA's approach is to pick a default schema for the user, while allowing the more savvy user to examine the different candidate schemas.

A key sub-problem in the hybrid schema design is deciding what portion of the data to be relational and what portion to be XML. Our usecases and practical knowledge tells us that data with high schema variability, data with rapidly evolving schema, and data that are sparse should be ideally persisted in XML [3]. ReXSA's approach is to determine the level of flexibility of each entity in the information model (an entity-relationship model can be assumed for concreteness) and classify each entity as relational or XML according to various requirements. However, this classification is not final. The relationships that an entity participates in can affect its classification. ReXSA then examines each relationship and transforms the entities and relationships into an appropriate collection of candidate schemas.

The contributions of this paper are summarized as follows.

- We address the new problem of hybrid relational-XML schema design, which has not been adequately explored yet, representing a serious need in the industry.
- We present ReXSA, an advisor tool for generating candidate hybrid relational-XML schemas. ReXSA presents the candidate schemas interactively to the user, who simply accept the default mappings or configure new default options.
- A key feature in ReXSA is a method for deciding what portion of the data to persist as relational and what portion as XML. No relational and XML design tools today present any solution for this issue.
- ReXSA considers qualitative properties of modeling elements such as variation and evolution that no other tool considers. Moreover, ReXSA is able to define schemas for hierarchies composed of both XML and relational entities. ReXSA also recommends schema alternatives to address different performance profiles.

The rest of this paper is organized as follows. Section 2 discusses related work and overviews the main XML-related features on IBM DB2. Section 3 discusses how to annotate the information model, which is the input to ReXSA, so as to capture data requirements. Section 4 details the ReXSA prototype, its main functionalities and algorithms. Section 5 discusses the main features of the prototype by employing it for modeling custom and real case scenarios. Finally, section 6 concludes the paper and presents future work.

2. RELATED WORK

The major commercial DBMSs (i.e. IBM DB2, Oracle and Microsoft SQL Server) provide support for storing and querying XML data in addition to relational data [18]. Designing high performance XML-relational DBMSs is a subject that has received a great deal of attention in recent years. However, considerable less efforts are directed toward providing a solid modeling methodology for designing databases on such hybrid data model.

While modeling relational databases is a well understood subject, modeling XML databases has yet to have a standard, well accepted methodology. Most approaches focus on extending traditional modeling techniques in order to support XML data. For example, [4] uses ORM as conceptual model and proposes heuristics for generating the "best" XML Schema, [10] improves the Extended Entity Relationship (EER), and [8, 16] are based on the UML. A more practical approach is [7] that proposes a conceptual model based on UML for defining complex XML schema. The tool also generates the schema based on specific designs (plain design, Russian doll, salami slice, Venetian blinds). Finally, [11] proposes normalization rules (XNF - XML Normal Forms) to DTDs (not XML Schema) focusing on two main properties of maximum connectivity and minimal redundancy.

Regarding commercial tools, several exist for translating ER models and/or UML models to relational database schema, such as: IBM Rational Data Architect, Visual Paradigm's Database Visual Architect, Toad Data Modeler, XCase, Silverrun ModelSphere. For generating XML schemas, IBM [14, 13] and Sparx Systems [19] have software packages or libraries for converting a conceptual model (UML, ECORE, etc.) to XML schemas.

All those previous works focus on designing pure-XML databases by using well known methodologies for modeling relational and object-oriented databases. Our work is independent of which tool or methodology is used for modeling the data. In fact, following the implementation of IBM DB2, we focus on a new context where XML data is part of a relational database by being defined as an atomic type. This way, the problem is not how to model an XML schema but rather how to *incorporate* XML data to a relational DB.

On the other hand, there has been much previous work on automatically *shredding* XML data into relational tables. Since there is no unique relational schema equivalent to a given XML schema or dataset, shredding algorithms pick an XML-to-Relational mapping based on some cost model. Various considerations in logical to physical schema mapping lend difficulties in defining quality metrics that consistently reflect the trade-offs among workloads, performance, flexibility and evolution. Our work differs from shredding in two important aspects. First, we map from a given logical model to a physical model that consists of relational tables that may have XML columns. Second, our approach relies on human experts' intuition to choose among the recommended mapping alternatives.

Earlier work also considered of workload-aware and workload-ignorant mappings from XML to relational. For example, [2] ignored the workload to discover regularities in XML data and store the 'structured' regular elements in a relation, while leaving 'irregular' fragments in XML. Alternatively, [5] reported cost-based schema mapping from XML data to relational tables, by estimating query performance for a given workload. It must be recognized, however, that

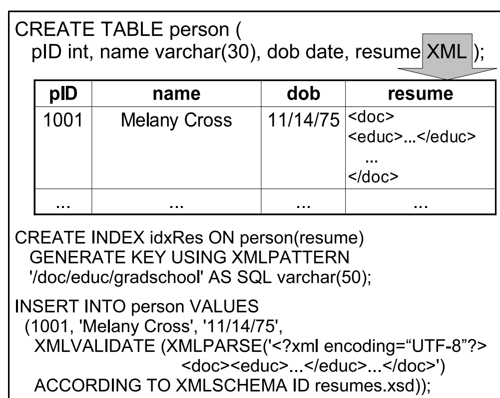


Figure 1: Table with a column of data type XML

further tuning of relational databases (e.g. with index, partitioning and materialized views) can lead the performance away from the initial cost estimates. The multiple latitudes in tuning hybrid databases aggravate the challenge further in defining a workload-based cost measure.

Hybrid Relational-XML DBMSs. The recently released IBM DB2 9 (for Linux, Unix and Windows) is a hybrid data server with optimized management of both relational and XML data, where XML is treated as a first-class data type [3, 17]. Moreover, it supports XML data natively, i.e. XML documents are internally stored in the tree format, following the XML data model. DB2's query processor handles both SQL and XQuery in a single framework, without translating XQuery statements to SQL [3]. When a document is inserted in a table, it may be associated with a schema. The same column may have many XML documents associated with different schemas. DB2 also provides specialized indexes for improving the performance of queries on values, paths and full text [17].

For example, Figure 1 shows a *create table* statement for personal information and resume, which is a column of data type XML. After creating the table, an index is defined on the value of the element *gradschool*, which is a subelement of *educ* within the resume document. Then, an *insert* statement populates the table with some values for the relational columns and a document for the XML column. Note that the document is validated against the schema *resume.xsd*.

3. THE INFORMATION MODEL

ReXSA is based on model-driven approach. The user first creates an *annotated information model* using a visual editor (e.g. IBM Rational Data Architect) for capturing the data requirements. ReXSA then analyzes the model and recommends candidate schemas. Before detailing all the prototype functionalities, this section describes the data features and annotations that the information model supports.

The information model is presented in a standard format such as UML or EER (i.e. ER diagrams with entity inheritance). An information model is a conceptual specification of the data and is the result of the first phase in the traditional three step database design methodology (i.e. conceptual design, logical design and physical design). The basic features of a model are well-known [9] and include: entities, attribute names and types, relationships between entities and semantic constraints (e.g. cardinality and participation constraints). ReXSA assumes that the model is *annotated* with other features relevant to the hybrid design: business artifacts, XML messages, schema variability, schema evolution, data versioning, and performance requirements.

Business Artifacts. In enterprise applications, it is common to find data elements naturally grouped into "documents" that are artifacts of business processes. Usually, these artifacts need to be processed as a single unit (e.g. travel expense forms, insurance claims, merchandise invoices) and hence should *not* be normalized across different storage units. Moreover, they often contain both structured and free text fields. Considering those factors, there are two main approaches for defining business artifacts within the information model. First, an artifact is modeled as an atomic attribute, preserving its details implicitly within the DBMS. Second, the data elements and relationships within a business artifact are modeled explicitly. The main advantage of the latter is that the logical and physical designs can be optimized for accessing the data elements within the artifact. ReXSA supports both ways of modeling business artifacts, but requires entities and relationships that are associated with the same artifact to be explicitly annotated so that the artifact boundaries are defined. The boundaries are specified by assigning a name to the artifact, so that XML entities (an their relationships) that model the same artifact are mapped to only one table with one XML schema.

XML Messages. Many information systems exchange data via XML messages using a service-oriented architecture (SOA). These XML messages often need to be persisted for audit trail. In terms of conceptual modeling, XML messages are somewhat similar to business artifacts with two important differences. First, data architects often prefer to have the option of persisting these messages *as-is* in their original XML format. Second, when these messages are to be persisted as XML, their original XML schema needs to be used. Therefore, ReXSA allows a business artifact to be annotated with its own XML Schema, so that it does not define yet another one (since the original schema for the messages *must* be used for compatibility reasons).

Schema Variability. A data element whose content takes on many variant structural forms is said to have high *schema variability*. In an information model, schema variability is closely associated with the presence of sparse attributes, optional values, multiple inheritance[1] and derivation. Modeling such information explicitly is important for ReXSA, because schema variability is a key factor on deciding whether an entity should be persisted as relational or XML. Modeling optional attributes in relational tables requires columns that support NULL values. Moreover, the presence of sparse attributes in the table often results in poor space utilization and sub-optimal query optimization. On the other hand, XML allows those attributes to be modeled and stored easily by including their values only when necessary. In other words, instead of storing NULL values, the elements are just absent in the XML data. Schema variability does not require any additional annotation in the information model other than the minimum cardinality of attributes and entities (0 if it is optional).

Schema Evolution. Very often, it is necessary to adapt the information model for considering changes on the ap-

[1]At the current stage, ReXSA does not support multiple inheritance, which is left for future work.

plication and user requirements. Usually, an update in the model generates changes on the database schema, then constituting a *schema evolution* processing. For relational data, schema evolution can trigger adding and deleting entities (tables), attributes (columns), updating data types and names, and relationship constraints. Updating the relational schema of a populated DB is not an easy task because it usually requires costly data migration. For XML data, schema evolution often results in changes to the XML schema. In a DBMS that supports XML-schema-agnostic columns (i.e. each document in a column is registered to a different schema or no schema at all), changes in the XML schema do not necessarily require any data migration. Rather, the application needs to be robust over data from different XML schema versions. For ReXSA, highly evolving entities need to be explicitly annotated in the information model, so that evolving schemas can be chosen for such entities. In principle, any entity may have its design changed, making it hard to predict which entities will evolve. However, on some applications, it is possible to know at modeling time which entities will evolve for sure. For example, the entity for derivatives in a financial application. Hence, those entities (whose evolution is certain and frequent) must be annotated in the information model.

Data Versioning. Schema evolution deals with changes in the *structure* of data elements. *Data versioning* deals with changes within the *content* of the data itself. Traditional transactional database applications are only concerned with the most up-to-date content. However, many enterprises (e.g. financial applications) require the history of the content changes to be kept for auditing purposes. Efficient versioning control is still a challenge for most commercial relational DBMS, in spite of the amount of research already published on the subject. On the other hand, with the advance of XML, new, less complex and more performance effective techniques have appeared (e.g. [6]). Entities or attributes to be versioned need to be explicitly annotated so that ReXSA can generate the appropriate schemas with additional versioning metadata such as a version identifier.

Performance Critical Elements. Often some critical data elements need to be accessed or updated with much stricter level of response time requirements. Entities involved in such cases may be explicitly annotated in the information model. There are several ways that such annotations can be used. An index advisor can use such annotations as hints for recommending indexes. ReXSA can use such annotations to decide whether to recommend additional materialization or denormalization. In a DBMS where XML access is not as high performing as relational, our tool can recommend that these entities be materialized in relational format for better partial update performance.

In summary, ReXSA considers as input an information model with the following concepts.

- **Entities** are defined by a name and an optional set of attributes. An entity may be annotated as evolving, versioned and performance critical.
- **Attributes** are associated with an entity or a relationship and are defined by a name and a data type. An attribute may also be annotated as optional, multi-valued, versioned or artifact. Multi-valued attributes may also be modeled as independent entities connected to the main entity by one-to-many relationships.

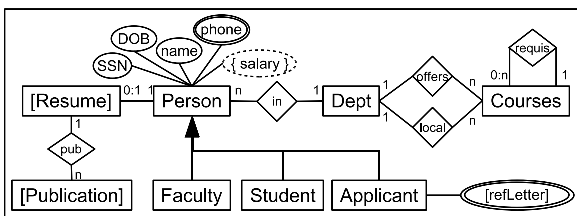


Figure 2: Simple academic model: entities in rectangles, attributes in ovals, multi-valued attributes in double-ovals (*phone*, *refLetter*), relationships in diamonds, inheritance with arrow, business artifacts within brackets (*Resume*, *Publication*, *refLetter*), versioned attribute within curly brackets (*salary*).

- **Business artifacts** are modeled as regular entities or attributes annotated as artifact. These artifacts may be modeled as (i) atomic entities or attributes with no detail specified, or (ii) decomposed entities and relationships for explicit modeling. In the first case, if artifacts are tightly associated with only one entity, then they should be modeled as attributes of that entity. Otherwise, they should be individual entities.
- **Relationships** connect two entities and have name, cardinalities for the involved entities, and an optional set of attributes.
- **Hierarchies** connect a parent entity to a derived child entity. The child entity inherits all attributes and relationships of the parent entity.
- **Annotations** are used to tag elements of the information model with additional information (e.g. keywords and/or values) that captures additional user requirements. For example, an entity annotated as *evolving* and an attribute annotated as *versioned*.

Figure 2 illustrates a simple academic design as a an example of an annotated information model. For clarity reasons, not all attributes are shown and graphic notations are used for identifying annotations. A hierarchy is defined with *Person* as parent entity and *Faculty*, *Student* and *Applicant* as its children. Attribute *salary* in *Person* is optional and annotated as versioned. Entities *Person* and *Applicant* have each one multi-valued attribute, *phone* and *refLetter* (reference letter), where the latter is also annotated as a business artifact. Entities *Resume* and *Publication* are also defined as a business artifacts. This example clarifies the choice of leaving the details of a business artifact implicitly, as done in *refLetter*, and defining the details explicitly, as done in *Resume*. Such decision is up to the designer and the consequences of this choice will be clarified later on.

4. SCHEMA ADVISOR

Having discussed the information model and the annotations used for capturing data requirements, we now describe ReXSA. Section 4.1 gives an overview of the main algorithm for mapping the information model to DDL statements. Section 4.2 details the scoring function that uses a measure of flexibility and other annotations for deciding how the entity will be modeled. Sections 4.3 and 4.4 detail how relationships and hierarchies are handled in ReXSA.

Algorithm 1 GENERATEDBSCHEMA($\mathcal{M}$, t)

Input: A Conceptual Model $\mathcal{M}$, and a threshold t
Output: A Set of DDL Statements

```
1: for all entities E ∈ M do // phase 1
2:    score ←SCOREENTITY(E, M)
3:    if score > t then
4:       type(E) ← XML
5:       schema(E) ← defSchema(E)
         // defSchema returns the DDL statements for E
         // if type(E)=XML, it assigns XML schema:
         // it creates a new schema if none annotated yet
6:    else
7:       type(E) ← Relational
8:       schema(E) ← defSchema(E)
9: for all hierarchies H ∈ M do // phase 2a
10:    Process H according to section 4.4
11:    Output schema(H)
12:    for all entities E ∈ H do
13:       done.add(E)
14: for all entities E ∉ done do // phase 2b
15:    Output schema(E)
16:    done.add(E)
17: for all relationships R ∈ M do // phase 3
18:    Output DDL according to Table 1
```

Algorithm 2 SCOREENTITY(E,$\mathcal{M}$)

Input: An entity E in a Conceptual Model $\mathcal{M}$
Output: Returns a score for E

```
1: E.attribs ← countAttribs(E)
2: E.flex ← countOptAttribs(E)
3: E.spec ← countSpecialAttribs(E)
4: if specialAnnot(E) then
5:    E.flex ← E.attribs − E.spec
6:    return 1
      // specialAnnot is true if E annotated as
      // artifact, message, evolving, or versioned
7: if criticalAnnot(E) then
8:    E.flex ← 0 ; E.spec ← 0
9:    return 0
      // criticalAnnot is true if E annotated as
      // critical performance
10: for all entity P ∈ Parent(E, M) do
11:    attribs ← attribs + P.attribs
12:    flex ← flex + P.flex
13:    spec ← spec + P.spec
14: return ((flex + spec) ÷ attribs)
```

4.1 Overview

Algorithm 1 presents the main pseudo code of ReXSA and works in three phases: (i) it scores each entity and decides if it is mapped to relational data or XML data, (ii) it refines that classification according to hierarchies, and outputs the DDL statements, and (iii) it processes their relationships and output their DDL statements.

Phase 1. This phase classifies each entity as either relational or XML (a *relational entity* is an entity whose attributes are mapped to relational columns, and an *XML entity* is one whose attributes are mapped to an XML column). The classification is based on the result of the function *scoreEntity* (line 2), which uses measures of flexibility (detailed ahead). After receiving its score, an entity is classified as relational or XML by employing a user-specified threshold t on the score (lines 3-8). If the score exceeds the threshold, the entity is classified as XML, otherwise as relational. Such a threshold can be loosely interpreted as the percentage of NULL values the data architect is willing to tolerate if an entity is defined as a relational table.

An internal map structure *schema* keeps the DDL statements for creating each entity. The statements are defined by a special function *defSchema* that receives as input the entity specification from the model (lines 5, 8). If the entity is XML, then the DDL statements create a table with an XML column and also define the XML Schema for that column. The techniques for transforming an information model to XML schemas are well-known and we refer to [4, 10, 8, 16, 7] for more information. Note that if the XML entity is annotated with a schema already, then that one is assigned to the column. The variable *schema* is necessary because tables may be merged or dropped in the next phases.

Phase 2. This phase outputs the DDL statements that create the entities and is divided in two parts. First, lines 9-13 process the hierarchies of the model as explained in section 4.4. In summary, there are different ways of mapping a hierarchy to the DBMS. The tool selects a default mapping and presents the alternatives to the user. Whenever ReXSA presents alternative schemas, it does so by either explaining which scenarios would benefit from each alternative or pointing out why there is an alternative to the default option. This information is fundamental for the user to make a more appropriate decision. Furthermore, when processing each hierarchy, it may be necessary to merge entities in one table or one XML schema. That is why the *output* function returns the schema of the whole hierarchy, rather than of each individual entity. Then, lines 14-16 process the remaining entities that do not belong to any hierarchy.

Phase 3. This phase examines each relationship and translates it to the respective DDL statements. Specifically, Algorithm 1 examines each relationship (lines 17-18) in the model and generates DDL statements for the resulting schema. One important detail (not shown for clarity reasons) is that the tool keeps track of the entities that were merged within the previous phase. For example, consider two entities A and B, and a set of relationships R between A and other entities. Also, A and B belong to one hierarchy and were merged so that the columns of A were added to the table for B. In this case, there is no specific table for A anymore. The tool keeps track that A "became" B and all relationships in R are then directed to B.

4.2 Scoring Entities

A crucial part of ReXSA is to decide whether an entity in the information model is persisted as relational or XML. Practical knowledge and clients' reports tell us that data with high schema variability and sparse data should ideally be persisted in XML [3]. Hence, we use the notion of *flexibility* to describe the variability of an entity's structure.

Measure of Flexibility. An entity is said to be *flexible* if it has many sparse (optional and multivalued) attributes. Another way of looking at flexibility is that if a relational table is used for an entity where the columns contain all the possible attributes the entity has, then flexibility is a rough measure of the number of NULL values in columns and rows of the table. Conversely, the table for a totally inflexible entity has its columns and rows populated by non-NULL values. Hence, relational tables are better for inflexible entities and XML columns are better for flexible entities.

Annotations. Besides flexibility, the decision to persist in relational or XML also depends on whether an entity is

annotated as evolving, business artifact, XML message or performance critical. ReXSA's approach is to assign each entity in the information model a score based these criteria.

Score Function. The pseudo code for the scoring function based on the measure of flexibility and the annotated information is outlined in Algorithm 2. Specifically, each entity keeps 3 variables: *attribs* as the total number of attributes, *flex* as the measure of flexibility (number of optional and multi-value attributes), and *spec* as the number of attributes defined as artifact, XML message and versioned. Lines 1-3 assign those values to the entity through functions *countAttribs*, *countOptAttribs*, and *countSpecialAttribs*. Note that the latter does not consider those attributes already counted as optional.

Lines 4-6 decide if the entity is to be persisted as XML by considering the annotations that define the entity as business artifact, XML message, evolving, versioned. The value of variable *flex* is updated so that $flex + spec = attribs$ (this is to force the formula at the end of the algorithm to result in 1). Likewise, lines 7-9 decide it is persisted as relational because the entity is defined as performance critical, and the values of *flex* and *spec* are updated accordingly. Then, lines 10-13 traverse the hierarchy of the entity (if it exists) in a bottom-up manner and update the entity values by considering the values of its parents. Note that the prototype main algorithm that calls the score function takes care that each hierarchy is evaluated top-down (so that when it processes line 10, all the parents of the current entity are already processed). Line 14 returns the score value according to a formula that sums up the values for *flex* and *spec* and divides the result by *attribs*. An idea for future improvement is to let the user define different weights for the values of *flex* and *spec* to be considered in this formula.

For example, considering the academic model from Figure 2, entity *Person* has a score 0.4 since it has two out of five attributes defined as multi-valued and optional/versioned. Entities *Faculty*, *Student* and *Applicant* have scores that also consider the score of their parent *Person*. Entity *Resume* (independent from the specification of its attributes) has a score 1, because it is defined as business artifact.

4.3 Mapping Relationships

Table 1 summarizes the DDL generation rules for relationships between two entities that do not participate in a hierarchy. Each entity has a set of attributes not shown in the table. A many-to-many relationship can also have a set of attributes. However, one-to-many (or many-to-one) relationships generally do not have attributes, because attributes that are specific to the relationship can be inlined into the entity with the larger cardinality. Each rule is described as follows (considering `[A:XML]-0..1-⟨R⟩-N-[B:REL]` as the textual representation for a relationship R between entities A and B, where A is an XML entity with minimum cardinality of 0 and maximum of 1, and B is a relational entity with cardinality N).

For relationships between two relational entities, the mapping to DDL statements is well-known and covered into any database textbook. For relationships between an XML entity and a relational entity, there are four main cases.

(1) `[A:XML]-0..1-⟨R⟩-1-[B:REL]` (each relational instance of B can be related to zero or one instance of A). In the general case, two tables are created, one for each entity. Table A has an XML column with the content of its attributes. The relationship is then established by a foreign key constraint on table A. If A is not involved in any other relationship, it can be inlined into B as an XML column of B, thus saving one table.

(2) `[A:XML]-N-⟨R⟩-1-[B:REL]` (each relational instance of B can be related to many instances of A). In the general case, two tables are created, one for each entity, and linked through a foreign key constraint into table A. Table A has an XML column for the content of its attributes. If A is not involved in any other relationship, all the instances of A that are related to a particular instance of B can be composed into a single XML document. The resulting document can be inlined into B as an XML column, thus saving one table.

(3) `[A:XML]-1-⟨R⟩-N-[B:REL]` (each XML instance of A can be related to many instances of B). Similar to the first case, two tables are created and a foreign key constraint establishes their relationship.

(4) `[A:XML]-M-⟨R⟩-N-[B:REL]` (many-to-many relationship between A and B). As the counterpart case of pure relational modeling, three tables are created, one for each entity and one for the relationship. The relationship table R contains the relationship's attributes, a foreign key to table A, and a foreign key to table B.

Likewise, for relationships between two XML entities, there are three cases.

(1) `[A:XML]-0..1-⟨R⟩-1-[B:XML]` (each instance of entity B can be related to zero or one instance of A).

(2) `[A:XML]-N-⟨R⟩-1-[B:XML]` (each instance of B can be related to many instances of A).

(3) `[A:XML]-N-⟨R⟩-1-[B:XML]` (many-to-many relationship between A and B).

All three cases are similar to their counterparts when only one entity is XML. The difference is that the XML documents from A and B can be merged in one document (*BandA*, *BandSetOfA*), except when the relationship is many-to-many. Furthermore, since each entity has its schema specified (in phases 1 and 2 of Algorithm 1), if the entities are merged when mapping the relationships, so are their schemas. In this case, there are two main options: (i) XML entity becomes an individual column within another entity, then its schema is assigned to the respective column as well; (ii) XML entity is merged with another XML entity in one column (for both entities), then their schemas are merged into one and assigned to the respective column. Note that merging is not recommended if the XML entities have other relationships (to different entities). In this case, they stay in separate tables with a primary key so that the other incident relationships can refer to the respective entities easily.

4.4 Mapping Entity Hierarchies

We have described how to map simple entities and relationships to DDL statements. In this section, we discuss how to map entities and relationships that involve entity hierarchies to DDL statements. The problem of mapping hierarchies to a pure relational schema has been studied by the Object-Relational DBMS community [20] and addressed by several commercial products (eg. Object-Relational mapping systems such as Oracle's Toplink and Hibernate). The mapping that we address in this section differs from previous work in that the entity hierarchies are not mapped to a pure relational schema, but to a hybrid or mixed relational-XML schema. Mapping to a hybrid schema introduces the

Table 1: Summary of how relationships are mapped to tables. Each relationship R relates two entities A and B. An entity B has attributes $\{b_1, \ldots, b_n\}$ of type $\{type_1, \ldots, type_n\}$. A many-to-many relationship R can also have attributes $\{r_1, \ldots, r_n\}$. For ease of presentation, the primary key of each table is denoted by ID.

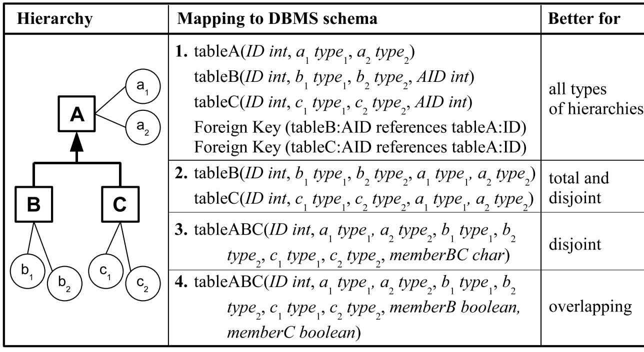
REL-REL Relationship	Schema Well-known
XML-REL Relationship	Schema
A : XML (0..1) — R — (1) B : REL	if A has other relationships, create separate tables: tableA(ID int, A xml, BID int) tableB(ID int, $b_1 type_1, \ldots, b_n type_n$) Foreign Key (tableA:BID references tableB:ID)
	if A has no other relationship, inline the XML entity: tableB($b_1 type_1, \ldots, b_n type_n$, A xml)
A : XML (N) — R — (1) B : REL	tableA(ID int, A xml, BID int) tableB(ID int, $b_1 type_1, \ldots, b_n type_n$) Foreign Key (tableA:BID references tableB:ID)
	if A has no other relationships, it can be composed into one XML document: tableB(ID int, $b_1 type_1, \ldots, b_n type_n$, $SetOfA$ xml)
A : XML (1) — R — (N) B : REL	tableA(ID int, A xml) tableB(ID int, $b_1 type_1, \ldots, b_n type_n$, AID int) Foreign Key (tableB:AID references tableA:ID)
A : XML (M) — R — (N) B : REL	tableA(ID int, A xml) tableB(ID int, $b_1 type_1, \ldots, b_n type_n$, AID int) tableR(AID int, BID int, $r_1 type_1, \ldots, r_n type_n$) Foreign Key (tableR:AID references tableA:ID) Foreign Key (tableR:BID references tableB:ID)
XML-XML Relationship	Schema
A : XML (0..1) — R — (1) B : XML	if A has other relationships, create separate tables: tableA(ID int, doc xml, BID int) tableB(ID int, doc xml) Foreign Key (tableA:BID references tableB:ID)
	if A has no other relationships, it can be merged with B tableB(ID int, B xml, A xml)
	tableB(ID int, $BandA$ xml)
A : XML (N) — R — (1) B : XML	tableA(ID int, A xml, BID int) tableB(ID int, B xml) Foreign Key (tableA:BID references tableB:ID)
	if A has no other relationships, it can be merged tableB(ID int, $b_1 type_1, \ldots, b_n type_n$, $SetOfA$ xml)
	tableB(ID int, $BandSetOfA$ xml)
A : XML (M) — R — (N) B : XML	tableA(ID int, A xml) tableB(ID int, B xml) tableR(AID int, BID int, $r_1 type_1, \ldots, r_n type_n$) Foreign Key (tableR:AID references tableA:ID) Foreign Key (tableR:BID references tableB:ID)

extra flexibility of having parts of the hierarchy as relational schema and others as XML schema. This section takes the first steps on how to do such mapping.

We assume each hierarchy may be further specified with *disjointness* and *completeness* constraints (following nomenclature from [9]). The first constraint specifies that the specialization is either *disjoint* (a parent instance belongs to at most one of the child entities) or *overlapping* (it may belong to more than one child entity). The second constraint specifies that the specialization is either *total* (a parent instance must belong to some child entity) or *partial* (it may belong to the parent only). ReXSA considers that a specialization is total disjoint by default. Considering these features, the translation of hierarchies to DDL statements is divided into two parts: (i) mapping entities to tables, and (ii) mapping the relationships between entities.

4.4.1 *Mapping Entities from a Hierarchy*

Considering only the relational case, there are four basic ways for mapping a hierarchy to a database schema, as illustrated in Figure 3. Each way of mapping may be more suitable for one type of hierarchy. Additionally, each mapping has its advantages and disadvantages. Method 1 maps each entity to an individual table. It preserves each entity individually at the cost of processing multiple joins when querying for attributes from different entities. Method 2 creates tables only for the leaves of the hierarchy, then mapping the attributes of the parent entities to each of its children. It solves the join problem at the cost of replicating information of the parent and processing an *outer union* for recovering all instances of the parent. Method 3 maps the whole hierarchy to one table and adds a control attribute in order to identify to which child entity the instance belongs. It requires no join or outer union to retrieve any of the entities but leads to many NULL values in the subclasses attributes. Method 4 is similar to the previous one, but it has one control attribute for each child (usually a boolean value). It also does not require join or outer union and will probably have less NULL values when the specialization is overlapping. ReXSA chooses by default method 1 for mapping



Hierarchy	Mapping to DBMS schema	Better for
	1. tableA(*ID int*, a_1 *type*$_1$, a_2 *type*$_2$) tableB(*ID int*, b_1 *type*$_1$, b_2 *type*$_2$, *AID int*) tableC(*ID int*, c_1 *type*$_1$, c_2 *type*$_2$, *AID int*) Foreign Key (tableB:AID references tableA:ID) Foreign Key (tableC:AID references tableA:ID)	all types of hierarchies
	2. tableB(*ID int*, b_1 *type*$_1$, b_2 *type*$_2$, a_1 *type*$_1$, a_2 *type*$_2$) tableC(*ID int*, c_1 *type*$_1$, c_2 *type*$_2$, a_1 *type*$_1$, a_2 *type*$_2$)	total and disjoint
	3. tableABC(*ID int*, a_1 *type*$_1$, a_2 *type*$_2$, b_1 *type*$_1$, b_2 *type*$_2$, c_1 *type*$_1$, c_2 *type*$_2$, *memberBC char*)	disjoint
	4. tableABC(*ID int*, a_1 *type*$_1$, a_2 *type*$_2$, b_1 *type*$_1$, b_2 *type*$_2$, c_1 *type*$_1$, c_2 *type*$_2$, *memberB boolean, memberC boolean*)	overlapping

Figure 3: Example of generic entity hierarchy and four possible mappings to a DB schema.

relational hierarchies, and presents the other methods as alternatives according to the hierarchy constraints. Finally, the impact of those options on the mapping of relationships will be discussed in section 4.4.2.

For hierarchies composed by XML entities, ReXSA decides how to map the entities to XML columns and then it defines an XML schema for each of those columns. Similar to the relational case, there are three options: (i) mapping each entity to its own table; (ii) mapping the leaves to tables; and (iii) mapping the whole hierarchy to one table. The first and the second options define the tables and foreign keys as the relational counterparts. The difference is that each table has one XML column associated to the respective XML schema. The third option creates a unique table with one XML column associated to one schema.

ReXSA uses the approach in [1] for defining the XML schema for the second and third options[2]. For example, for the hierarchy in Figure 3, ReXSA defines complex types A, B, and C where B and C are derived by extension from A. Depending on the hierarchy constraints, A is abstract (total) or not (partial), and the identifier is unique among the types (disjoint) or not (overlapping). The combinations of constraints entail matching those features. For example, a total disjoint hierarchy is mapped to XML schema by having the parent entity as an abstract type and unique identifiers among all complex types defined for that hierarchy.

This approach works well because the XML model is made for data with hierarchical structure. Nevertheless, merging the whole hierarchy within one document may not be the user's intention. In fact, considering the other properties of the hierarchy and the relationships within it, it may be a better option to define one table for each entity or one table for each child. Hence, ReXSA deals with these three options by having the third one (unique XML schema) as default option, and presenting the other two as alternatives.

4.4.2 Mapping Relationships from a Hierarchy

While mapping entities follows deterministic methodologies, mapping the relationships of the entities within a hierarchy is more challenging. Furthermore, although ReXSA

[2]However, the types of hierarchy constraints follow a different nomenclature in [1]: *mutual exclusion* is partial disjoint, *partition* is total disjoint, *union* is total overlapping, and without constraint is partial overlapping.

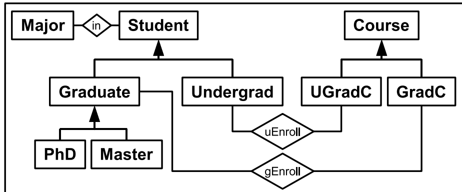


Figure 4: Example of an entity hierarchy for students and courses in a university.

uses approaches based on [9, 1] for mapping the entities of a hierarchy to the schema, to the best of our knowledge, there is no standard procedure for mapping hierarchies with relationships to database schemas. Given an entity hierarchy composed of relational entities, XML entities or both, the problem is how to properly map the constraints imposed by the relationships defined for those entities.

One simple solution would be to apply the mappings from Table 1 to the entities on the hierarchy. However, due to the inheritance constraints established by a hierarchy, the mappings from Table 1 may not be directly applied. Therefore, in this section we discuss the challenges imposed by three main scenarios and their variations. We also introduce the main reasoning for the solutions employed by ReXSA. Then, we summarize the resulting mapping methodology in Table 2 (which is employed by Algorithm 1).

The scenarios are discussed considering the EER diagram in Figure 4, a partial view of an academic system. There are two entity hierarchies, one for students, and one for courses, where attributes are omitted for clarity purposes. Graduate students enroll only on graduate courses and undergrad students on undergrad courses, and each student is in a major.

Other factors are considered for mapping each scenario. Specifically, the tool must be able to map hierarchies composed of relational entities, XML entities and sometimes both. It also needs to evaluate the hierarchy constraints of disjointness and completeness. Clearly, the main challenge is to solve each of the possible combinations of scenario, type of entity within the hierarchy and type of hierarchy constraint. Hence for each hierarchy within the model, the tool proceeds as follows: first, it identifies which levels of the hierarchy have relationships (i.e. identifies one of the three scenarios); second, it checks whether the entities within the hierarchy are all relational, all XML or mixed; last, it verifies the type of hierarchy (total/partial, disjoint/overlapping) and proposes candidate schemas accordingly.

Scenario 1: Parent Relationships. The first scenario presents the parent entity with relationships and the children with none. This scenario is illustrated by the hierarchy defined by entities *Graduate*, *PhD* and *Master* in Figure 4 (let's disregard entity *Student* for a moment). The entity *Graduate* has a relationship *gEnroll* with entity *GradC*, and its children have no relationship. In this case, students on both *PhD* and *Master* may also enroll in graduate courses, since the classes inherit such relationship. Effectively, the entire hierarchy subtree rooted at *Graduate* is considered as a single entity with respect to the relationship *gEnroll*.

As already mentioned, different solutions may be proposed according to the hierarchy participants and constraints. First, consider that all entities in the hierarchy are rela-

tional. In this case, mapping the relationships is similar to mapping the entities, where the four mappings presented in Figure 3 may be applied regarding the relationship as a regular attribute. Specifically, defining individual tables for each entity (method 1) has the advantage that the relationship relates to the parent table and no other mapping is necessary. On the other hand, defining tables only for the children (method 2) has the disadvantage that the relationship needs to relate to each of the children (since there is no table for the parent). Merging the hierarchy in one table (methods 3 and 4) requires that the relationship be related to the one table itself, and no other mapping is necessary.

Mapping the relationships when the hierarchy is composed of XML entities also depends on the way the hierarchy is mapped. Defining the hierarchy as one table *Graduate* (the default option for XML hierarchy) or three individual tables (one per entity) implies that the relationship *gEnroll* relates directly to *Graduate*. Then, no additional constraint is necessary. Defining tables only for the children results in the same problem as the relational case.

In both cases (relational and XML hierarchies), having tables only for the children when there is a relationship with the parent is not the best idea. Therefore, when presenting such mapping as an alternative, this warning is presented as well. Finally, the default option is to keep the tables for *Graduate* separated from those with which it has relationships. However, when the whole hierarchy is mapped to one XML column, it can also be merged with the related entity. This merging can only happen on the following situations: (i) the parent has only one relationship, then it can be merged as a column on the related entity; or (ii) the parent has other relationships, but the related entity has only this one, then it can be merged to the parent table.

Scenario 2: Children Relationships. The second scenario presents the opposite case, where the child entities have relationships and the parent has none. An illustration of this scenario is the hierarchy defined by entities *Course*, *UGradC*, and *GradC*, where each child entity has one relationship. Having a relational hierarchy, the relationships are mapped similar to the previous scenario. Defining individual tables for each entity is easier and no other mapping is necessary. Defining tables only for the children may be even better, since the parent has no other relationship. Merging the hierarchy as one table is more complicated. In this case, it is necessary to specify additional check constraints in order to verify the correctness of the relationship.

For example, according to either method 3 or 4 from Figure 3, the entities *Course*, *UGradC*, and *GradC* are then merged into one table *CourseUG*. The information model specifies relationship *gEnroll* in which instance of the entity *Graduate* relates only to instances of *GradC* (now merged into *CourseUG*). In order to preserve the information model specification, defining the relationship *gEnroll* to entity *CourseUG* requires an extra check constraint assuring the related instance in *CourseUG* has the control attribute defining the instance as member of *GradC*.

In this case, Figure 5 illustrates two alternatives. Even though *gEnroll* and *uEnroll* are N-to-M relationships, this figure has the mappings of 1-to-N relationships, for the sake of discussion. In the first case (within the shaded area), the student tables refer to *CourseUG*. The referential integrities specified by the foreign keys on those tables simply ensure that *courseU* and *courseG* refer to rows on table

Undergrad

gID	col1	col2	courseU
101	x	x	1
102	x	x	1
...	...	...	

CourseUG

cID	c1	c2	u1	u2	g1	g2	control
1	x	x	x	x			**U**
2	x	x			x	x	**G**

Graduate

gID	col1	col2	courseG
201	x	x	2
...	...	...	

CourseUG

cID	c1	c2	u1	u2	g1	g2	control	underg	grad
1	x	x	x	x			**U**	101	
2	x	x			x	x	**G**		202
...	...	...	...	...	...	...	...	...	...

Undergrad

uID	col1	col2
101	x	x
102	x	x
...	...	...

Graduate

gID	col1	col2
201	x	x
202	x	x
...	...	...

Figure 5: Tables for merged courses and the respective student tables. The x indicates that there is some value for that column. In the shaded area, the student tables refer to the course table. At the bottom, the course table refers to the students.

CourseUG. The question is how to specify that the row referred by *courseU* (within *CourseUG*) has the *control* value equal to 'U'. Specifically on DB2, a check constraint can only refer to values on the same table. Hence, the only solution in DB2 is to specify the condition within triggers and stored procedures, which is what ReXSA does in this case.

The same problem arises when the foreign keys go on reverse, from *CourseUG* to the student tables as illustrated at the bottom of Figure 5. In this case, *CourseUG* has two foreign keys, one for *Undergrad* and other for *Graduate*. The procedure needs to specify that if *control* is 'U' then the foreign key for *Graduate* is null. Otherwise, if *control* is 'G' then the foreign key for *Undergrad* is null.

Now, consider a hierarchy composed of XML entities. As the relational case, mapping each entity or each child to individual tables presents no problem for the relationship constraints. On the other hand, mapping the hierarchy to one document within a table presents the same concerns as the relational case. The solution is to define triggers and stored procedures for the XML tables as well.

Scenario 3: Parent and Children Relationships. The third scenario combines the previous ones such that both parent and child entities have different relationships. An illustration of this scenario is the hierarchy composed of entities *Student*, *Graduate* and *Undergrad*, and it is necessary to preserve the constraints entailed by all relationships at the same time. Therefore, the first mapping with individual tables is probably the easiest one, since the relationships will relate the respective entity tables. Defining tables only for the children may incur in update anomalies due to the redundancy of the parent information stored into the children tables. Collapsing the hierarchy in one table is equally complicated because additional checking is necessary to verify the correctness of the relationship. All these problems happen to both relational and XML hierarchies. Therefore, in this case, the default option adopted by ReXSA is to keep each entity in an individual table in order to preserve the relationships. However, the other options and the required triggers and stored procedures may also be presented to the user, with their respective pros and cons.

Table 2: Summary of how relationships within hierarchies are mapped to tables.

Relational Hierarchy	XML Hierarchy
PARENT RELATIONSHIP **1.** tableA(*ID int*, a_1 $type_1$, ***DID** int*) tableB(*ID int*, b_1 $type_1$, *AID int*) tableC(*ID int*, c_1 $type_1$, *AID int*) Foreign Key (tableB:AID references tableA:ID) Foreign Key (tableC:AID references tableA:ID) **Foreign Key (tableA:DID references tableD:ID)** -- or D references A; or R1 has separate table	**1.** individual tables, all entity attributes grouped into one XML column, same foreign keys.
2. tableB(*ID int*, b_1 $type_1$, a_1 $type_1$, ***DID** int*) tableC(*ID int*, c_1 $type_1$, a_1 $type_1$, *DID int*) **Foreign Key (tableB:DID references tableD:ID)** **Foreign Key (tableC:DID references tableD:ID)** -- or D references B, C; or R1 has separate table	**2.** children tables, parent and entity attributes grouped into one XML column, same foreign keys.
3. tableABC(*ID int*, a_1 $type_1$, b_1 $type_1$, c_1 $type_1$, *memberBC char,* ***DID** int*) **Foreign Key (tableABC:DID references tableD:ID)** -- or D references ABC; or R1 has separate table	**3.** one table, one schema for hierarchy, same FK. -- XML as a column of D if entity A no other relationsip -- D as column of ABC or inlined within schema, if D has no other relationship
CHILD RELATIONSHIP **1.** tableA(*ID int*, a_1 $type_1$) tableB(*ID int*, b_1 $type_1$, *AID int*) tableC(*ID int*, c_1 $type_1$, *AID int,* ***EID** int*) Foreign Key (tableB:AID references tableA:ID) Foreign Key (tableC:AID references tableA:ID) **Foreign Key (tableC:EID references tableE:ID)** -- or E references C; or R2 has separate table	**1.** individual tables, all entity attributes grouped into one XML column, same foreign keys.
2. tableB(*ID int*, b_1 $type_1$, a_1 $type_1$) tableC(*ID int*, c_1 $type_1$, a_1 $type_1$, ***EID** int*) **Foreign Key (tableC:EID references tableE:ID)** -- or E references C; or R2 has separate table	**2.** children tables, parent and entity attributes grouped into one XML column, same foreign keys.
3. tableABC(*ID int*, a_1 $type_1$, b_1 $type_1$, c_1 $type_1$, *memberBC char,* ***EID** int*) **Foreign Key (tableABC:EID references tableE:ID)** **+ triggers/stored procedures** -- or E references ABC; or R2 has separate table	**3.** one table, one schema for hierarchy, same FK, + triggers/procedures. -- XML as a column of E: never: loose information. -- E as column of ABC with extra constraint. -- E inlined into ABC (in C) if no other relationship

4.4.3 Discussion

These variability in how entity hierarchies are mapped to database schema is partly a modeling problem and partly a mapping problem. A user may prefer to *model* the hierarchy flattened to the children or as individual entities, which is a modeling decision. When a user model the *whole hierarchy* with individual entities, then mapping it to individual tables or flattening it to children tables is a mapping decision. The schema advisor must support both ways of designing the tables, which results in more candidate schemas. Hence, when a hierarchy is defined (i.e. the user chose to model the hierarchy composed of individual entities), the default approach for mapping it to the schema is to keep individual tables. Moreover, ReXSA allows the user to set up the default as defining children tables or unique tables as well.

Finally, table 2 has a summary of the mapping possibilities. It shows only the scenarios for parent relationships and children relationships, because the third scenario (both have relationships) is a merge of those two. The table has the options separated by relational and XML hierarchies, and by the three possible entity mappings (the fourth method is the same as the third one with more control attributes).

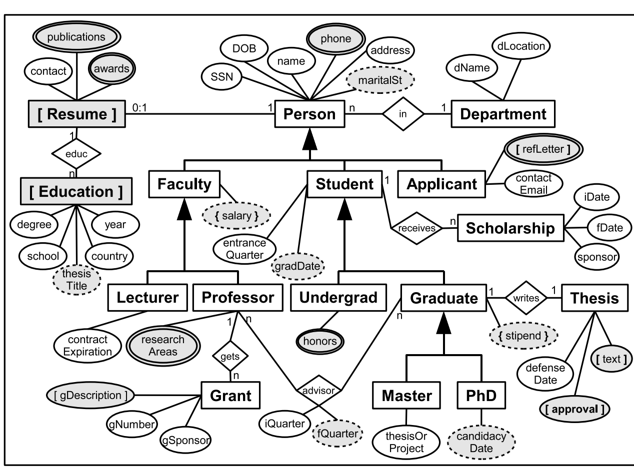


Figure 6: Academic scenario with personal information on the members of a department. The notation is the same as previously defined with special elements in gray: business artifacts within brackets, versioned attributes within curly brackets, optional and multi-value attributes.

5. APPLYING REXSA TO USE CASES

In this section, we apply ReXSA on two use cases to analyze how the prototype works. The prototype input is a conceptual model encoded as a text file. The prototype analyzes the input by specifying a score to each entity according to Algorithm 2. Then it writes the output as a set of DDL statements according to Algorithm 1. Our use cases are composed by two scenarios: a custom created academic model with different features supported by the tool, and a real-case scenario of a clinical document repository.

The alternative statements are shown to the user interactively, so that he/she can choose what better applies to the application. For presentation purposes, we opted for illustrating all alternatives as comments within a DDL file (even though this is *not* how the user receives the output file).

Custom Model. The first scenario is a traditional academic model with focus on the personal information of the members within department. Figure 6 illustrates an EER diagram with the following entities *Department*, *Person* (the main entity) with its hierarchy formed by *Faculty*, *Lecturer*, *Professor*, *Student*, *Undergrad*, *Graduate*, *Master*, *PhD*, *Applicant*, and the secondary entities *Resume*, *Education*, *Grant*, *Scholarship*, *Thesis*. Each entity has a set of attributes and relationships. The element names are self-explanatory, with special ones marked in gray. Note that we do not intent to model the whole scenario, but rather the minimal information necessary to illustrate some of the main points of our approach.

Table 3 presents the score for each entity and its components (*flex* and *attribs*) separated by the values of entity and the values inherited within the entity hierarchy. For simplicity, the academic design does not specify any entity as performance critical. The values for the final score vary from 0 to 1. Since entities *Resume* and *Education* are annotated as business artifact with the same name (not illustrated), the *flex* value is the same as *attrib*, composing a score 1.

Table 3: Final score and component values for academic model (ordered by final score)

		Own values		Inherited values	
Entity	**Score**	**Flex**	**Attribs**	**Flex**	**Attribs**
Department	0	0	2		
Scholarship	0	0	3		
Grant	0.33	1	3		
Person	0.33	2	6		
Lecturer	0.37	0	1	3	7
Student	0.37	1	2	2	6
Applicant	0.37	1	2	2	6
Master	0.40	0	1	4	9
Faculty	0.42	1	1	2	6
Undergrad	0.44	1	1	3	8
Graduate	0.44	1	1	3	8
Professor	0.50	1	1	3	7
PhD	0.50	1	1	4	9
Thesis	0.67	2	3		
Resume*	1	3	3		
Education*	1	5	5		

*Resume and Education have flex=attribs because the entities are defined as business artifacts (i.e. 100% flexibility)

```
CREATE TABLE person
    (name varchar (100),
    ssn int NOT NULL PRIMARY KEY ,
    DOB date,
    address varchar (256),
    phone XML,
    maritalSt varchar (10));
CREATE TABLE department
    (dName varchar (100),
    dLocation varchar (256),
    department_ID bigint
        NOT NULL PRIMARY KEY);
CREATE TABLE student
    (entranceQuarter varchar (10),
    gradDate date,
    student_ID bigint
        NOT NULL PRIMARY KEY);
CREATE TABLE thesis
    ( thesis_ID bigint
        NOT NULL PRIMARY KEY,
    thesisX XML);
CREATE TABLE resume
    ( resume_ID bigint
        NOT NULL PRIMARY KEY,
    resumeX XML);
CREATE TABLE education
    ( education_ID bigint
        NOT NULL PRIMARY KEY,
    educationX XML);
```

```
-- add foreign keys for inheritance
ALTER TABLE student
   ADD COLUMN inh_person bigint NOT NULL
       ADD CONSTRAINT FK_inh_person8
          FOREIGN KEY (inh_person)
          REFERENCES person;
-- in = person-dept=1:1 1:N
ALTER TABLE dept
   ADD COLUMN in bigint NOT NULL
   ADD CONSTRAINT FK_in3
      FOREIGN KEY (in)
      REFERENCES person;
-- writes = graduate-thesis=1:1 1:1
ALTER TABLE thesis
   ADD COLUMN writes bigint  NOT NULL
   ADD CONSTRAINT FK_writes5
      FOREIGN KEY (writes)
      REFERENCES graduate;
-- This is an 1 to 1 relationship
-- an alternative is: opposite foreign key:
-- ALTER TABLE graduate
--    ADD COLUMN writes bigint  NOT NULL
--    ADD CONSTRAINT FK_writes5
--       FOREIGN KEY (writes)
--       REFERENCES thesis;
-- This is an 1 to 1 relationship to XML entity
-- an alternative is: XML as column:
-- DROP TABLE thesis;
-- ALTER TABLE graduate
--    ADD COLUMN thesis XML;
-- educ = resume-education=1:1 1:N
-- This is a relationship within the same
-- business artifact:  XML schema of Resume
-- updated for  including Education schema
 DROP TABLE education;
```

Figure 7: DDL statements for academic example with threshold = 60%.

According to Algorithm 1, the threshold defines an entity as relational or XML. Assuming a high threshold of 60%, then the business artifact entities *Resume* and *Education* are defined as XML, and so is the entity *Thesis*. Figure 7 presents some of the DDL statements generated by the tool.

Specifically, the *CREATE TABLE* statements for *person*, *department*, and *student* are as usual, but with optional and multi-valued attributes defined as XML data type (multi-valued are always mapped to XML column; if the user prefers an individual table for it, then he/she has to model the attribute as an entity). Alternatively, the tool also offers the option of specifying multi-valued attributes as independent tables (not illustrated for space limitation). Note that the tool assigns primary keys to all those entities that do not have one specified in the conceptual model.

The entities defined as XML (*thesis*, *resume*, *education*) have the same columns: one for the primary key, and one for the XML data type. The name of the XML column is the name of the entity with an "X" at the end. When an XML column is specified for an entity, the XML schema file is also created. When an attribute is specified as business artifact in the information model, no schema is defined for it (since there is no detailed information for defining one). As an option, the user may also annotate the respective schema information to be defined for that business artifact. The annotation must include the parameters for the *REGISTER XMLSCHEMA* statement. In this case, besides the DDL statements shown, there are also the statements that register each XML schema to its respective column, as for example:

```
REGISTER XMLSCHEMA 'http://univ.place/resume.xsd'
FROM 'file:///c:/ReXSA/schemas/resume.xsd'
AS academia.resumeSchema
```

The inheritance in this example is mapped so that each entity has an individual table with the inheritance links mapped to foreign keys. Then, the relationships are mapped to DDL as *ALTER TABLE* statements. The relationship *in* between *person* and *department* is simple. Note that the tool makes sure all constraint names are unique by adding a serial number at the end of each constraint name.

The relationship *writes* is an 1-to-1 relationship (between *graduate* and *thesis*) and hence, there are two options for the direction of the foreign key. One of them is chosen by default and the second one is presented as alternative to the user (illustrated in the figure as comments). Moreover, since this is a relationship with an XML entity, the alternative of having the XML entity as a column within the relational table is also presented to the user (if the XML entity has no other relationship, following Table 1).

The relationship *educ* is an 1-to-N relationship between two XML entities that belong to the same business artifact. In this case, the individual table for *Education* is deleted and its schema is merged to the schema of *Resume*.

Real Case Scenario. This use case considers HL7 (Health Level Seven) [12], an organization that specifies a messaging standard for health care applications to exchange key sets of clinical and administrative data. Specifically, HL7 has defined an XML architecture for exchanging clinical documents based on XML DTDs and whose semantics are defined using the HL7 RIM (Reference Information Model) and HL7 registered coded vocabularies. Finally, the RIM is expressed using UML and can be extended by HL7 international affiliates in order to meet local needs.

Figure 8 illustrates a partial view of HL7 RIM. The main classes are: *Act* (actions that are executed and must be documented), *Participation* (context of an act: who, for whom, where...), *Entity* (physical things and beings in health care), *Role* (entity's role in an act), *ActRelationship* (bind one act to another), and *RoleLink* (relationship between individual roles). One central piece is the class *Entity* that includes living subjects, organizations, material, pharmaceutical substances, places, etc (not shown for lack of space).

In order to make this example more interesting, the entities were *not* annotated as evolving, versioned, nor artifact. Only those attributes that are optional, multivalued and artifact were annotated as so. Hence, Table 4 has the score

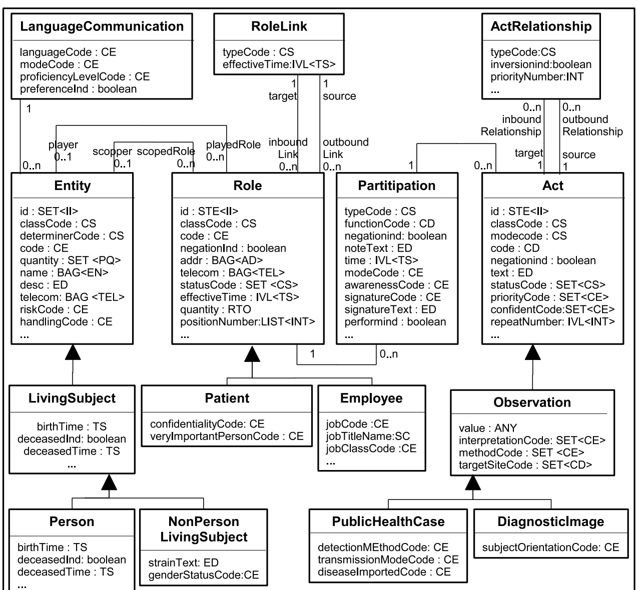


Figure 8: Partial view of HL7 RIM

according to the attributes information only. All entities have score greater than 0.83 (except *RoleLink*), which happens because most of the attributes are either optional or multivalued, and sometimes both. The DDL statements for HL7 create one table with an XML schema because, in fact, all entities are part of the same artifact (not shown due to space constraints).

Discussion. We have applied ReXSA to the artificially constructed academic usecase and to the HL7 reference information model and analyzed the resulting schemas. For the academic usecase, the schema recommended by ReXSA is qualitatively comparable to what a data architect would have produced. For HL7, ReXSA recommends that all its data elements be persisted using XML. This recommendation is actually quite good, because human experts have also made the same choice [12].

6. CONCLUSION

XML support has become a standard feature in all major commercial DBMS. However, practitioners are still unsure of how to design relational-XML schemas. One of the key problems is which portion of the data to persist as relational, which portion as XML. Other challenges are how to incorporate XML data to relational tables and how to maintain relationship constraints when defining the schema for hierarchies. This paper reports on ReXSA, a tool that is being prototyped for IBM DB2 9. Given an annotated information model, ReXSA generates DDL statements for candidate schemas. We described how to annotate an information model and outlined ReXSA's algorithms and mapping procedures. We applied ReXSA on two usecases and discussed the results qualitatively. Our work on ReXSA has uncovered many avenues for future work. We plan to explore how to incorporate data instances and data statistics in the design and mapping process, how to measure the quality of a mapping that would be meaningful to the user, and also how to integrate with other DB2 and Rational tools.

Table 4: Final score and score component values for HL7 model

		Own values		Inherited	
Entity	**Score**	**Flex**	**Att.**	**Flex**	**Att.**
LanguageComun.	1	4	4		
Entity	0.83	10	12		
LivingSubject	0.89	7	7	10	12
Person	0.92	8	8	17	19
NonPersonLiv.	0.90	2	2	17	19
RoleLink	0.50	1	2		
Role	0.90	10	11		
Patient	0.92	2	2	10	11
Employee	0.94	8	8	10	11
Participation	0.92	12	13		
ActRelationhip	0.93	14	15		
Act	0.90	19	21		
Observation	0.92	4	4	19	21
PublicHealthC.	0.92	3	3	23	25
DiagnosticImage	0.92	1	1	23	25

7. ACKNOWLEDGEMENTS

The authors gratefully acknowledge Sharon C. Adler and Anders Berglund for their insightful discussions on the XML design process.

8. REFERENCES

[1] R. Al-Kamha, D. W. Embley, and S. W. Liddle. Representing Generalization/Specialization in XML Schema. In *Proc. of EMISA*, 2005.

[2] M. F. Alin Deutsch and D. Suciu. Storing semistructured data with stored. In *Proc. of SIGMOD Conference*, 1999.

[3] K. S. Beyer et.al. System RX: One Part Relational, One Part XML. In *Proc. of SIGMOD Conference*, 2005.

[4] L. Bird, A. Goodchild, and T. Halpin. Object Role Modelling and XML-Schema. In *Proc. of ER*, 2000.

[5] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *Proc. of ICDE*, 2002.

[6] S.-Y. Chien et.al. Supporting Complex Queries on Multiversion XML Documents. *ACM Trans. Inter. Tech.*, 6(1):53–84, 2006.

[7] C. Combi and B. Oliboni. Conceptual modeling of XML data. In *Proc. of SAC*, 2006.

[8] R. Conrad, D. Scheffner, and J. C. Freytag. XML Conceptual Modeling Using UML. In *Proc. of ER*, 2000.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 5th edition, 2006.

[10] R. Elmasri et.al. Conceptual Modeling for Customized XML Schemas. In *Proc. of ER*, 2002.

[11] D. W. Embley and W. Y. Mok. Developing XML Documents with Guaranteed "Good" Properties. In *Proc. of ER*, 2001.

[12] HL7 Reference Information Model. `http://www.hl7.org/library/data-model/RIM/C30204/rim.htm`.

[13] IBM Alphaworks. Model Transformation Framework. `http://www.alphaworks.ibm.com/tech/mtf`.

[14] IBM Rational Software. Generating XSD schemas from UML models. `http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/topic/com.ibm.xtools.transformations.doc/topics/txsdover.html`.

[15] IBM Rational Software. `http://www.ibm.com/software/rational`.

[16] H. Liu, Y. Lu, and Q. Yang. XML Conceptual Modeling with XUML. In *Proc. of ICSE*, 2006.

[17] M. Nicola and B. V. der Linden. Native XML Support in DB2 Universal Database. In *Proc. of VLDB*, 2005.

[18] M. Rys, D. Chamberlin, and D. Florescu. XML and Relational Database Management Systems: the Inside Story. In *Proc. of SIGMOD Conference*, 2005.

[19] Sparx system's uml to xml schema transformation. `http://www.sparxsystems.com/resources/mda/xsd_transformation.html`.

[20] M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, 1996.