Roma Tre University
Ph.D. in Computer Science and Engineering
XXIV Cycle

A model oriented approach to heterogeneity

Francesca Bugiotti

Advisor: Prof. Paolo Atzeni

PhD Coordinator: Prof. Stefano Panzieri

A model oriented approach to heterogeneity

A thesis presented by

Francesca Bugiotti

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in Computer Science and Engineering

Roma Tre University

Dept. of Informatics and Automation

April 2012

To my parents

Contents

Abstract

Data heterogeneity is a major issue in any context where software directly deals with data. The most general expectation of any complex system is the so-called seamless integration, where data can be accessed, retrieved and handled with uniform techniques, tools and algorithms.

In this sense, integration can be considered as the opposite of heterogeneity. It is a property of data that gives a measure of the degree of coherent exploitability. Indeed, the more conformed pieces of data are, the more information will be retrieved from them.

Passing from highly heterogeneous to integrated data is the subject of data integration, the discipline that formulates in formal terms all the processes and the technical and algorithmic steps needed to transform data. Heterogeneity is a twofold issue. It has a technical connotation and a theoretical one. Data can be distributed in different data sources, memorized in various formats and encoding conventions, be queried via incompatible programming interfaces. All these kind of problems can be addressed with the design of proper adapting architectures, involving the presence of a number of connectors homogenizing from a technical perspective. On the other hand, the core problem with heterogeneity is that data can be intrinsically different because different data models -collections of structural entities- are adopted to organize them. A relational database instance is intrinsically different from a XML file or from a collection of objects in a NoSQL key-store repository. The divergences are not merely technical, but representational.

The aim of this work is dealing with data integration techniques under a number of perspectives. From the theoretical perspective, we consider the Model Management as the framework to formalize translation problems. A schema, instance of a certain model will be translated to another schema instance of a target model. We recognize the need for a model-independent solution to schema and data translation and, in general, to model management problems. Hence we present MIDST, a tool born from many years of experience on schema and data translation, based on a metalevel approach.

From the performance perspective, we appreciate the value of runtime environments, where translations are not performed out of the system with an import-translate-export process; by contrast, we illustrate, as a novel contribution, MIDST-RT, an evolution of MIDST, where translations are performed at runtime and even generate views of data. Data heterogeneity also poses more dynamic problems, linked with data evolution and maintenance.

As an important example, we adopt the perspective of model management to present a model-independent solution to round-trip engineering problem, that prototypes the typical propagation of changes among related schema. MISM, a model management framework based on MIDST (and MIDST-RT) is then illustrated. The contribution in this field is the use of MIDST as a model management platform.

Nowadays market demand for highly specialized data processors, performing at best in specific cases such as web content retrieval, document search, object serialization, parallel calculation is taken in particular consideration. NoSQL engines promise exceptional performance in non transactional fields and leverage simplified but peculiar data models. Therefore a core goal of data integration is providing techniques and tools to facilitate the interaction with these systems. In the final part of this work, we extend our metalevel approach to encompass NoSQL systems and present several experimental results on them also addressing still unexplored indexing strategies.

In conclusion, in this thesis we provide contributions in the three mentioned fields of data integration: a theoretical model management framework is proposed; performance is addressed through a change of paradigm within the context of metamodel approaches involving the fact that translations are operated directly in the target system; NoSQL systems are touched on and encompassed in the proposed metamodel.

Introduction

Data integration is the data engineering discipline consisting in considering data residing in different sources, environments or technical platforms, having different representations, data model or encoding conventions and combining them in such a way they can be accessed uniformly. In an idiomatic sentence, it is often said that data integration goal is handling heterogeneous data so that they appear as one.

In industry and scientific reality in which multiple representations of the same data coexist even in the same information system, there is a continuous request for reconciliation and conformation. In the practice, this implies the repetition of several high level tasks, transformations, leading to a multiplicity of heterogeneous data to integrated information.

In industrial field, market products show efforts to converge towards an enterprise level master model asserting unambiguously the meaning of a certain pieces of data. Historically, a first level of homogeneity was achieved with the adoption of relational databases [Cod70], where a common (relational interface) offered a structured and unified view of data, constraining them under a well determined model and language. Now, need for homogeneity is leading to several industry practices, known with several similar expressions such as *master data management*, *data federation* and so forth.

However, data integration in general aims at the conformation of all the data schemas present in an information system. It involves both the solution to many technical issues, such as protocol conciliation, data stream management, choice of technical interfaces (drivers) and the conceptual and formal conformation of the involved data models.

Contributions

What is central in this work is the analysis and a concrete contribution to the most up to date techniques and algorithms for the reconciliation of different data models. In the literature, the so-called difference is often referred to as *heterogeneity*. Heterogeneity is a twofold concept: it has a technological aspect, and a representational one.

Technological aspects are faced by a number of research and industry approaches integrating heterogeneous data environments and architectures. This world is often known as *enterprise data integration* and covers a series of integration patterns and techniques. They can be classified in two kinds of general approaches to the problem, *master data management* and *data transformation*.

Master data management philosophy tends to federate sources and representations under a unique and more encompassing one acting as a proxy. Data transformation, borrowing techniques and patterns from classical ETL, aims at creating a shared environment where heterogeneities are conformed into a shared environment.

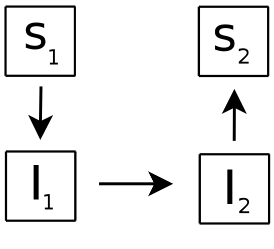
Heterogeneity and model management A major goal of this work, is proposing an approach that aims at being more encompassing than master data management or enterprise data integration. Reasonings are based on a metadata driven approach in order to propose a model-independent solution to data heterogeneity. In fact, the main innovative result is that the proposed techniques and algorithms are independent of the specific involved models, but have a general validity in force of common structural characteristics of different models.

According to our original approach, data models are defined according to a preliminary observation by Hull and King [HK87], as it was considered by Atzeni and Torlone in [AT93], asserting that all the data models share some structural characteristics that can be named as constructs. Therefore here, leveraging those formalizations, we define a data model -or simply model hereinafter- as a collection of constructs.

This work follows the formal framework known as *Model Management*. According to Bernstein [Ber03], model management is a theoretical framework defining an algebra on data models. Model constructs are the operands in this algebra, while schema evolution techniques are the operators, and are synthesized in specific patterns. In particular, we consider the *Modelgen* operator that, in model management, is the one devoted to the generation of a schema of a certain model, given a schema of another model and a set of translation rules. Translation rules specify how constructs of the source model have to be combined and rearranged in order to yield a schema of the target model.

Modelgen operator, as it is described in [Ber03] is model-dependent in the sense that a different implementation is necessary for each pair of involved models. In this discipline, the need for a model-generic approach is widely recognized and often pointed out as a major goal [MRB03].

In Atzeni and Torlone's original approach [AT93] and in several following works [ACB05a, ACB06, ACG07, ACT+08] we proposed a model-independent solution to schema and data translation, relying on the presence of a general and flexible metamodel that can be used to represent and handle data. Transformation and translations rules, that allow the managing and the transformation of data, are defined with respect to this metamodel.



$$\begin{tikzcd} \boxed{S_1} \arrow[d] & \boxed{S_2} \\ \boxed{I_1} \arrow[r] & \boxed{I_2} \arrow[u] \end{tikzcd}$$

Figure 1: The round-trip engineering problem.

Besides the basic problem of translating from a schema of a model into a schema of another model, data heterogeneity presents the data architect with several other high level problems, consisting in the presence of conceptual but repetitive tasks needed to integrate evolutions. As a novel contribution, in this work activities are framed within the model management framework and correlated with a model-independent solution.

As a preliminary example of possible conceptual task coming from integration issues we point out *round-trip engineering*. Consider Figure 1. S_1 is a specification and I_1 its implementation (possibly provided by a design tool).

Then a "manual" modification of the implementation leads to a new version I_2, which is not coherent with S_1. The goal is to find a specification S_2 from which I_2 could be generated. The first is oriented to the coherence at data level, the second is mainly oriented to the maintenance of functionalities of application when data are updated or migrated into another model.

Concerning the round-trip problem there has been the effort to design a general approach to the definition of transformation rules and operators that act at meta-level. On the one hand, in this study we pursue schema evolution and model management operators as formal definitions, on the other hand a major goal is framing them in a global approach tending to a model-independent heterogeneity.

In the data integration scenario, increasing needs for performance and scalability, strongly remarked by web applications first, and cloud services then, introduced the necessity for lightweight and runtime solutions to model management problems.

These non functional demands, led to a definition of an approach that adopts a metalevel representation that allows the definition of the translation problem, where data is not moved from the operational system and translations are performed directly on it [ABBG09a]. What the user obtains at runtime is a set of views (the target schema) conform with the target model. The main difference is that the import process concerns only the schema of the source database and the rules for schema translation are here used as the basis for the generation of views in the operational system. In such a way data is managed only within the operational system itself.

Heterogeneity and NoSQL Moreover, during the last years numerous new systems, not following the RDBMS paradigm (neither in the interface nor in the implementation), have been developed [Sto10, Sto11a]. Their common features are scalability and support to simple operations only (and so, limited support to complex ones), with some flexibility in the structure of data. Most of them also relax consistency requirements. They are often indicated as *NoSQL* systems, because they can be accessed by APIs that offer much simpler operations than those that an be expressed in SQL. Probably, it would be more appropriate to call them *non-relational*, but we will stick to common usage and adopt the term NoSQL.

An interesting but simplified classification has been proposed, based on the modeling constructs available: key-value stores (representatives of which are Redis and Scalaris), document stores (including MongoDB and CouchDB) and extensible record stores (including BigTable, HBase and Cassandra).

There exist more than fifty systems, in the various categories, and each of them can be used by means of a different interface (different model and different API). Indeed, the lack of standard is a great concern for any organization interested in adopting any of these systems: applications are not portable and skills and expertise acquired on a specific system are not reusable with another one.

In this scenario there is the need for a definition of a common interface and a general modeling approach of the use cases scenario and data that are consequently handled in different systems. The original contribution presented here (and also in [ABR12a]) is the definition of a uniform programming and access interface for NoSQL systems. Technical proofs programming simulations were indeed performed ([ABR12b]). The contribution benefits from the definition of a meta-layer encompassing NoSql data models. This meta-layer can be successfully embedded into MIDST dictionary and allows to provide a common view even over relational and non relational systems. The aimed standardization also led to the definition of a general API that can be used to access NoSQL system in a JDBC-like fashion.

The pursuit of model management problems with respect to NoSQL databases also gave the opportunity to give a contribution to improved scalability and performance of tools and methodologies in this field with particular respect to RDF querying.

Obtained results are, to the best of our knowledge, of noticeable interest for the achieved performance.

In particular, a study visit at INRIA Saclay in Paris was the chance for an industry-like experimentation of the cited performance contributions. Amazon S3 was considered as a third-party storage system for RDF documents that had to be queried with SPARQL triples. The study converged in the definition of four indexing strategies for RDF documents and, technically, they were implemented in SimpleDB and presented in [BGKM12].

Overview of the dissertation

The remainder of the work is organized as follows. In Chapter 1 we present our original contributions in the context of the translation tool MIDST. The metamodel approach is presented and a complete description of the dictionary is provided.

In Chapter 2 introduce the formal definition of model management operators and face the major problems in a model-independent fashion. MISM, an evolution of MIDST, is presented as a model management framework.

In Chapter 3 performance issues are dealt with. MIDST and MISM off line approach consisting in an initial import of the data into the supermodel is evolved. MIDST-RT is proposed as a runtime translation tool.

Chapters 5 and 4 discuss non relational databases and encompass them in MIDST supermodel. A common programming interface for NoSQL systems is motivated.

Then, in 6 a particular non relational DBMS, SimpleDB, is pursued and several indexing strategies are presented. In 7 some related works are and in 8 we draw up or conclusions.

CHAPTER 1

MIDST and the metamodel approach

In this Chapter we present our metamodel approach by means of MIDST, our model-inpdependent schema and data translation tool. The methodology behind MIDST is based on the idea of a metamodel defined as a set of constructs that can be used to define models which are instances of the Metamodel [ACB06, ACT$^+$08, AGC09].

Here we present a description of the metamodel adopted in MIDST to implement the MODELGEN operator, the one devoted to generate a schema of a model given a schema of another model.

1.1 The metamodel

MIDST adopts a model-generic representation of schemas based on a combination of constructs. Its founding observation is the similarity of features which arises across different data models. This means that all the existing models can be represented with a rather small set of general purpose constructs [HK87] called *metaconstructs* (or simply *constructs* when no ambiguity arises). Let us briefly illustrate this idea. Consider the concept of entity in the ER model family and that of class in the OO world: they both have a name, a collection of properties and can be in some kind of relationship between one another. To a greater extent, it is easy to generalize this observation to any other construct of the known models and determine a rather small set of general

Metaconstruct	**Relational**	**Object-Relational**	**ER**	**XSD**
Abstract	-	typed table	entity	root element
Lexical	column	column	attribute	simple element
BinaryAggregation-OfAbstracts	-	-	binary relationship	-
AbstractAttribute	-	reference	-	-
Generalization	-	generalization	generalization	-
Aggregation	table	table	-	-
ForeignKey	foreign key	foreign key	-	foreign key
StructOfAttributes	-	structured column	-	complex element

Figure 1.1: Simplified representation of MIDST metamodel

constructs. Therefore models are defined as sets of constructs from a given universe, in which every construct has a specific name (such as "entity" or "object"): for instance a simple version of the ER model may be composed of Abstracts (the entities), Aggregations of Abstracts (the relationships) and Lexicals referring to Abstracts (attributes of entities); instead the relational model could have Aggregations (the tables), Lexicals referring to Aggregations (the columns), and foreign keys specified over finite sets of Lexicals. Thus schemas are collections of actual constructs (schema elements) related to one another. Figure 1.1 lists a subset of MIDST metaconstructs giving the intuition about how model representation is hanlded.

1.2 The supermodel

The set of all the possible constructs in MIDST forms the *supermodel*, a major concept in our framework. It represents the most general model, such that any other model is a specialization of it (since a subset of its constructs). Hence a schema S of a model

M is necessarily a schema of the supermodel as well.

Figure 1.3 describes all the constructs of the *supermodel* in the form of a UML class diagram. The complete description of them follows:

Abstract Any autonomous concept of the scenario.

Aggregation A collection of elements with heterogeneous components. It make no sense without its components.

StructOfAttributes A structured element of an Aggregation, an Abstract, or another StructOfAttributes. It could be not always present (isOptional) and/or admit null values (isNullable). It could be multivalued or not (isSet).

AbstractAttribute A reference towards an Abstract that could admit null values (isNullable). The reference may originate from an Abstract, an Aggregation, or a StructOfAttributes.

Generalization It is a "structural" construct stating that an Abstract is a root of a hierarchy, possibly total (isTotal).

ChildOfGeneralization Another "structural" construct, related to the previous one (it can not be used without Generalization). It is used to specify that an Abstract is leaf of a hierarchy.

Nest It is a "structural" construct used to specify nesting relationship between StructOfAttributes.

BinaryAggregationOfAbstracts Any binary correspondence between(two) Abstracts. It is possible to specify optionality (isOptional1/2) and functionality (isFunctional1/2) of the involved Abstracts as well as their role (role1/2) or whether one of the Abstracts is identified in some way by such a correspondence (isIdentified).

AggregationOfAbstracts Any n-ary correspondence between two or more Abstracts.

ComponentOfAggregationOfAbstracts It states that an Abstract is one of those involved in an AggregationOfAbstracts (and hence can not be used without AggregationOfAbstracts). It is possible to specify optionality (isOptional1/2) and functionality (isFunctional1/2) of the involved Abstract as well as whether the Abstract is identified in some way by such a correspondence (isIdentified).

Lexical Any lexical value useful to specify features of Abstract, Aggregation, StructOfAttributes, AggregationOfAbstracts, or BinaryAggregationOfAbstracts. It is a typed attribute (type) that could admit null values, be optional, and identifier of the object it refers to (the latter is not applicable to Lexical of StructOfAttributes, BinaryAggregationOfAbstracts, and AggregationOfAbstracts).

ForeignKey It is a “structural” construct stating the existence of some kind of referential integrity constraints between Abstract, Aggregation and/or StructOfAttributes, in every possible combination.

ComponentOfForeignKey Another “structural” construct, related to the previous one (it can not be used without ForeignKey). It is used to specify which are the Lexical attributes involved (i.e. referring and referred) in a referential integrity constraint.

XMLAttribute The last “structural” construct, used in the current version of the tool only in ObjectRelational-XML model and qualified by it. It relates a Struct to an Abstract.

MIDST manages the information of interest in a rich dictionary. Let us summarize its main features. It has two layers, both implemented in the relational model: a *basic* level and a *metalevel* (Figure 1.2).

The basic layer of the dictionary has a model-specific part (some tables of which are shown in Figure 1.5 with reference to a simple example in Figure 1.4), where schemas are represented with explicit reference to the various models, and, more important, a model-generic one, where there is a table for each construct in the super-

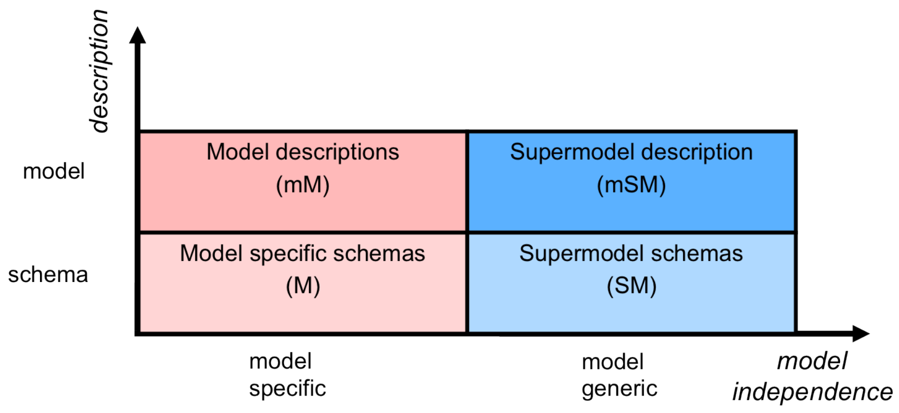


Figure 1.2: The four parts of the dictionary

model: so there is a table for SM_ABSTRACT (the SM_ prefix emphasizes the fact that we are in the supermodel portion of the dictionary) a table for SM_AGGREGATION and so on (with an example in Figure 1.6). These tables have a column for each property of interest for the construct (for example, a Lexical can be part of the identifier of the corresponding Abstract, or not, and this is described by means of a boolean property). References are used to link constructs to one another, for example an attribute of entity in the ER model has to belong to a parent construct, which could be an Abstract (an entity). In both parts, constructs are organized in such a way they guarantee the *acyclicity constraint*, meaning that no cycles of references are allowed between them. This is convenient in situations where a complete navigation through the schemas is necessary and a topological order is helpful.

The two parts of the dictionary play complementary roles in the translation process, which is MIDST's main goal: the model specific part is used to interact with source and target schemas and databases, whereas the supermodel part is used to perform translations, by referring only to constructs, regardless of how they are used in the individual models. This allows for model-independence.

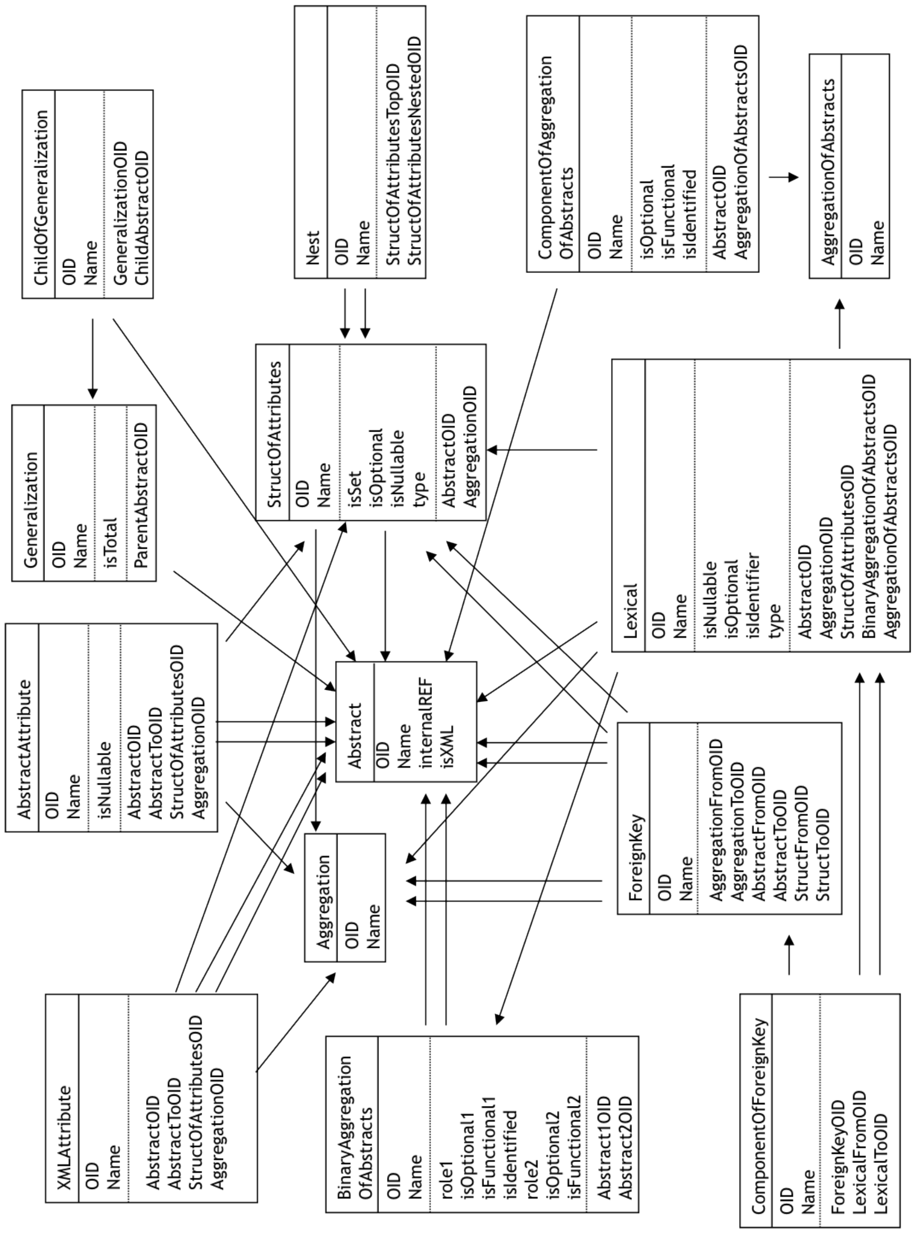


Figure 1.3: The supermodel

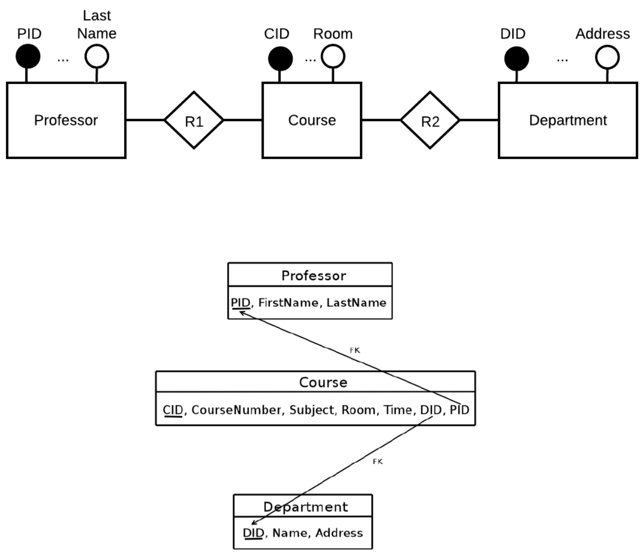


Figure 1.4: A simple example

MIDST dictionary includes a higher layer, a *metalevel*, which gives a characterization of the construct properties and relationships among them [ACB05b, ACT+08]. It involves few tables, each with few rows, which form the core of the dictionary. A significant portion is shown in Figure 1.7. Its main table, named MSM_CONSTRUCT (here, the MSM_ prefix denotes that we are in the "metasupermodel" world, as we are describing the supermodel) stores the name and a unique identifier (OID) for each construct, so this table actually memorizes every allowed construct; indeed, the rows of this table correspond essentially to those in Figure 1.1. Each construct is also char-

ER_ENTITY		
OID	Entity-Name	Schema
e1	Professor	s1
e2	Department	s1
...	...	...

ER_ATTRIBUTEOFENTITY					
OID	Entity	Att-Name	Type	isKey	Schema
a1	e1	PID	int	true	s1
a2	e1	FirstNane	string	false	s1
a3	e2	LastName	string	false	s1
a4	e2	DID	int	false	s1
a5	e2	Name	string	false	s1
...	...	...	...	...	...

ER_BINARYRELATIONSHIP							
OID	Rel-Name	Entity1	IsOpt1	IsFunctional1	Entity2	...	Schema
b1	R1	e1	false	true	e2	...	s1
...	...	...	...	...	...	...	...

REL_TABLE		
OID	Table-Name	Schema
t1	Professor	i1
t2	Department	i1
...	...	...

REL_COLUMN					
OID	Table	Col-Name	Type	isKey	Schema
c1	t1	PID	int	true	i1
c2	t1	FirstName	string	false	i1
...	...	...	...	...	i1
c7	t2	Name	string	false	i1
...	...	...	...	...	...

Figure 1.5: A portion of a model-specific representation of schemas S_1 and I_1 of Figure 1.4

acterized by a set of properties describing the details of interest. There is a table, MSM_PROPERTY, reporting name, type and owner construct for each property. The properties, for example, allow to define whether an entity attribute is identifier or not

SM_ABSTRACT		
OID	Abs-Name	Schema
e1	Professor	s1
e2	Department	s1
...	...	...

SM_AGGREGATION		
OID	Aggr-Name	Schema
t1	Professor	i1
t2	Department	i1
...	...	...

SM_LEXICAL						
OID	Abstract	Aggr	Lex-Name	Type	isId	Schema
a1	e1	-	PID	int	true	s1
a2	e1	-	FirstName	string	false	s1
a3	e2	-	LastName	string	false	s1
...	...	...	...	...	...	...
c1	-	t1	DID	int	true	i1
...	...	...	...	...	...	...
c7	-	t2	Name	string	false	i1
...	...	-	...	...	...	...

SM_BINARYAGGREGATIONOFABSTRACTS							
OID	Aggr-Name	Abstract1	IsOpt1	IsFunctional1	Abstract2	...	Schema
b1	R1	e1	false	true	e2	...	s1
...	...	...	...	...	...	...	...

Figure 1.6: A portion of a model-generic representation of the schemas S_1 and I_1 of Figure 1.4

and to specify the cardinality of relationships. Constructs refer to one another with references, recorded in the table MSM_REFERENCE.

As we have illustrated, the metalevel lays the basis for the definition of constructs which can be then used in defining models and so on the structure of the lower layer of the dictionary: in fact, the model-generic layer (Figure 1.6) has one table for each row

MSM-CONSTRUCT		
OID	Construct-Name	IsLex
mc1	Abstract	false
mc2	Lexical	true
mc3	BinaryAggregationOfAbstracts	false
mc4	AbstractAttribute	false
...	...	...

MSM-PROPERTY			
OID	Prop-Name	Constr	Type
mp1	Abstract-Name	mc1	string
mp2	Att-Name	mc2	string
mp3	IsId	mc2	bool
mp4	IsFunctional1	mc3	bool
mp5	IsFunctional2	mc3	bool
...	...	...	...

MSM-REFERENCE			
OID	Ref-Name	Constr	ConstrTo
mr1	Abstract	mc2	mc1
mr2	Abstract1	mc3	mc1
mr3	Abstract2	mc3	mc1
...	...	...	...

Figure 1.7: The supermodel part of the metalevel portion of the dictionary of MIDST

in MSM_CONSTRUCT (and so we have, as we said, tables named SM_ABSTRACT, SM_AGGREGATION, SM_LEXICAL, and so on), with columns corresponding to the properties and references of the construct, as described in MSM_PROPERTY and MSM_REFERENCE, respectively.

1.3 The translations

To manage heterogeneous data, many applications need to translate data and their descriptions from one model, to another. The metamodel approach was conceived in order to define a common layer among the various datamodels in such a way that model-independent schema translation was possible. According to the model management terminology, this translation is performed by MODELGEN operator: given two models M_1 and M_2 and a schema S_1 of M_1, MODELGEN translates S_1 into a schema S_2 of M_2 that properly represents S_1.

Defining the MODELGEN operator for all the couples of models is a complex operations so the a major benefit of the supermodel is the fact that it can be used as a pivot model so that it is sufficient to have translation of each model to and from the supermodel. This approach is implemented in MIDST.

In the tool translations are composed of three phases: first of all schemas in their model-specific representation are imported into the translation tool. Model specific representation are mapped into a model independent one. With reference to the supermodel illustrated before the process of importing a specific metamodel leads to use only a subset of its constructs. Then in this phase the actual translation takes place.

According to our methodology model translations are described as a decomposition of elementary steps transforming typical structural patterns into others. Therefore a complex translation is divided into many declarative mappings that are executable within the supermodel.

Each translation step in MIDST is specified as a Datalog program, which is a set of Datalog rules. The language was chosen for two reasons: first, it matches in an effective way the structure of our data model and dictionary (which is implemented in relational form); second, its high level of abstraction and the declarative form allow for a clear separation between the translations and the engine that executes them. Moreover, Datalog can be straightly translated into SQL and the original choice was aimed at covering the widest spectrum of application scenarios. However, other syntax or specification formalisms could be adopted as well. More precisely, we use a variant of Datalog with "value invention" [Cab98, HY90], where values for new OIDs are generated.

We use Skolem functors to generate OIDs and literals can be qualified with attributes. For example, the following rule translates an Abstract (an entity in an ER model) into an Aggregation (a simple table):

```
Aggregation (
    OID: SK1(oid),
    Name: name
)
<-
Abstract (
    OID: oid,
    Name: name
);
```

Notice the use of a Skolem functor, SK1 in the example, which, given the OID of an Abstract, produces a corresponding OID for an Aggregation. We use Skolem functions to generate new identifiers for constructs, given a set of input parameters, as well as for referring to them whenever needed, given the same set of parameters. Skolem functions are injective. So, in this case SK1 will generate a different OID, and so a different new Aggregation, for each Abstract in the source schema. For a given target construct many functors can be defined (denoted by numeric suffixes in the examples), each taking different parameters in dependence on the source constructs the target one is generated from. As a consequence, in order to guarantee the uniqueness of the OIDs, the ranges of the Skolem functions are disjoint. Other functors for Aggregation generate a different set of OIDs.

Translations taking place in real scenarios require several Datalog programs to specify the transformation of each construct. We pursue a modular approach and decompose translations into simple steps (each returning a coherent schema of a specific model that is then used by the subsequent step). This is done by means of a library of Datalog programs implementing elementary steps and of an inference engine which can determine the appropriate sequence of steps to be applied.

For example, assume we have as the source an ER model with binary relationships (with attributes) and no generalizations and as the target is the Relational model. To perform the task, we would first translate the source schema by renaming constructs into their corresponding homologous elements (abstracts, binary aggregations, lexicals, generalizations) in the supermodel and then apply the following steps:

1. eliminate many-to-many aggregations, by introducing new abstracts and one-to-many aggregations

2. eliminate attributes of aggregations,by introducing new lexicals to the corresponding abstract Ş

If instead we have a source ER model with generalizations but no attributes on relationships (still binary), then, after the copy in the supermodel, we can apply steps (2) and (3) above, followed by another step that takes care of generalizations:

4. eliminate generalizations (replacing them with references)

5. eliminate abstracts by introducing new aggregations

6. eliminate aggregation of attributes by introducing foreign keys and component of foreign keys

It is important to note that the basic steps are highly reusable. Here is an example of Datalog Rule that translates a BinaryAggregationOfAbstract in ForeignKey.

```
R2    ForeignKey (
           OID: SKi(aggrOID),
           Name: aggName,
           AggregationFrom: #SKj(absOid1),
           AggregationTo: #SKj(absOid2)
       )
      <-
      BinaryAggregationOfAbstract (
           OID: aggrOID,
           Name: aggName,
           Abstract1: absOid1,
           IsOptional1: isOpt1,
           IsFunctional1: ``TRUE'',
           Abstract2: absOid2
       );
```

In the rule, the # symbol denotes a Skolem functor, which is used to generate new identifiers (in the same way as we did in MIDST [ACT+08]). Indeed, the functor is interpreted as an injective function, in such a way that the rule produces a new construct for each different source construct on which it is applicable. The various functions also have disjoint ranges.

CHAPTER 2

A model-independent approach for model management

Model management is a metadata-based approach to database problems aimed at supporting the productivity of developers by providing schema manipulation operators aiding the automation of high level tasks.

Here MIDST is pursued on the perspective of describing a new platform for model management: MISM (Model Independent Schema Management). It offers a set of operators to manipulate schemas, in a manner that is both model-independent (in the sense that operators are generic and apply to schemas of different data models) and model-aware (in the sense that it is possible to say whether a schema is allowed for a data model). This is the first proposal for model management in this direction.

We consider the main operators in model management: merge, diff, and modelgen. These operators play a major role in solving various problems related to schema evolution (such as data integration, data exchange or forward engineering), and we show in detail a solution to a major representative of the class, the round-trip engineering problem.

2.1 Model Management

The need for complex transformations of data arises in many different contexts, because of the presence of multiple representations for the same data or of multiple sources that need to coexist or to be integrated [BM07, Haa07, HAB$^+$05]. A major goal of technology in the database field is to enhance the productivity of software developers, by offering them high-level features that support repetitive tasks. This has been stressed since the introduction of the relational model, with the emphasis on set-oriented operations [Cod70, Cod82], but it was pursued, at least implicitly, in earlier developments of generalized techniques [McG59]. The *model management* proposal [BHJ$^+$00, Ber03] is a recent, significant effort in this direction: its goal is the development of techniques that consider metadata and operations over them. More precisely, a model management system [BM07] should handle schemas and mappings between them by means of operators supporting operations to discover correspondences between schemas (MATCH), performing the most common set-oriented operations (such as union of schemas, MERGE, and difference of schemas, DIFF) and translating them from a data model to another (MODELGEN). These operations should be specified at a high level, on schemas and mappings, and should allow for the (support to the) generation of data-level transformations. Many application areas can benefit from the use of model management techniques, including data integration over heterogeneous databases, data exchange between independent databases, ETL (Extract, Transform, Load) in data warehousing, wrapper generation for the access to relational databases from object-oriented applications, dynamic Web site generation from databases.

Most of the work in model management has considered the need for *model independence*, that is, the fact that the techniques do not refer to individual data models,[1] but are more general. In detail, this requires that a single implementation of the operators should fit (i.e. be applicable) to any schema regardless of the specific data model

[1]There is some disagreement on terminology in the literature: we use the term *data model* here for what is often called just *model* [ACB06, AT96] and in some papers *metamodel* [BM07].

it belongs to. This has usually been done by adopting some “universal data model,” a model that is more general than the various models of interest in a heterogeneous framework. In the literature, such a data model is called *universal metamodel* [BM07] or *supermodel* [ACB06, AT96]. If the operations of interest also include translations from a data model to another (the MODELGEN operator), it is important that the individual data models are represented, in such a way that it becomes possible to describe the fact that a schema belongs to a data model. We will call this property *model-awareness*. The various proposals for MODELGEN [ACB06, AT96, MBM07b, PT05] do include the model independence feature, to a larger or lesser extent. For the other operators, the major efforts in the model management area (as summarized by Bernstein and Melnik [BM07]) do not handle the explicit representation of data models nor generic definitions of the operators.

The goal of the approach described in this chapter is to show a model independent and model aware approach to model management, thus providing concrete details to Bernstein’s original proposal [Ber03] and contributing to support its feasibility.

Motivating examples

In order to have a context for specific examples and a complete solution, we will refer to the “round-trip engineering” problem [Ber03], which can be defined as follows: given two schemas, where the second is somehow obtained from the first (for example, generated in a semiautomatic way, with standard rules partially overridden by human intervention), the problem has the goal of “repairing” the first if the second is modified. This problem is often considered in model management papers [Ber03] as a representative of the “schema evolution” family. These problems arise in all application settings and therefore can be used to demonstrate the effectiveness of model management, in terms of both individual operators and compositions of them.

Let us consider an example derived from an academic scenario (see Figure 2.1):

a university has various schools and one of them has a relational database with a portion containing all the information of interest about its departments, courses, and

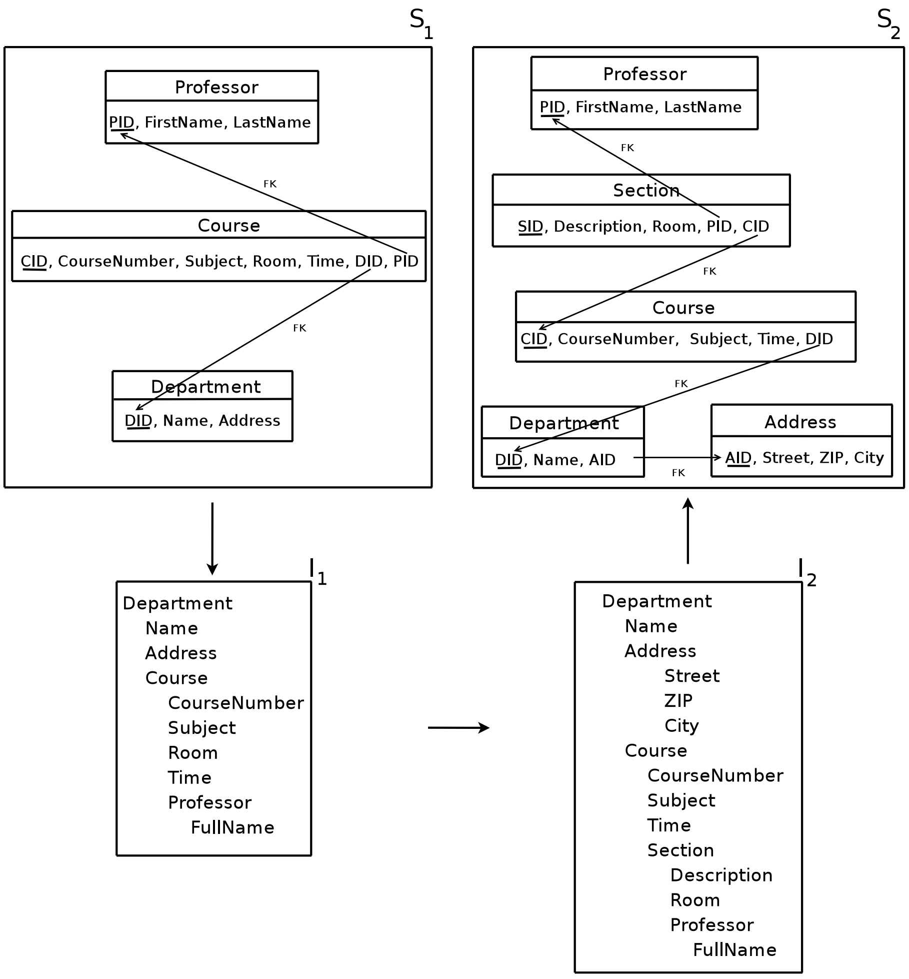


Figure 2.1: The round-trip engineering problem

professors. Its schema is shown in the box labeled S_1 in Figure 2.1. It is composed of three tables, *Professor*, *Course*, and *Department*. Apart from the specific attributes, each relation has a key, denoted by the "ID" suffix and underlined in the figure. As each course is offered by a specific department and given by a professor, there are foreign keys from *Course* to the other two tables, denoted by arrows in the figure.

Assume now that this portion of the database is used (together with other goals) as the source to send data on courses to a central office in the university, which gathers data from all schools. This office requires data in an XML format, which is the one sketched in the box labeled I_1 in Figure 2.1. There is indeed a close correspondence between S_1 and I_1 (possibly because they were designed together). In fact, I_1 can be obtained by means of a nesting operation based on departments, each with the associated set of courses and with the instructor for each course. Clearly, this is one natural way to transform the relational data in S_1 into XML, but not the only one, as there would be other solutions that involve course or professor as the root. In this sense, we can say that this is not the result of an automated translation, but of a customization, that is, a choice among a few standard alternatives. Let us also observe that in S_1 we have attributes *FirstName* and *LastName* for *Professor*, whereas in I_1 we have the element *FullName*. There could be various reasons for this, but the only aspect relevant here is that, again, the transformation has been customized, with the concatenation of the two attributes in S_1 into a single element in I_1.

Then, assume that the exchange format is modified, with a new version, I_2, also shown in Figure 2.1. There are a few differences between I_2 and I_1. First, we have that *Address* is a simple element in I_1, while it is a complex element in I_2, composed of *Street*, *Zip*, and *City*. The second, and most important, difference is the presence of a complex element *Section* nested in *Course* and containing *Professor*. A course can be composed of various sections. Each section has a single professor, and therefore *Professor*, which in I_1 was directly contained in *Course*, is part of *Section*. Each section of a course takes place in a different room so the element *Room* is now in *Section*.

Now, the goal is to obtain a schema in the relational model (for example the one shown in the box labeled S_2 in Figure 2.1) that properly corresponds to S_1 as modified by the changes in I_2. It should be clear that S_2 cannot be obtained by applying to I_2 a standard, automatic translation from XML to the relational model (an application of the MODELGEN operator), because we could not keep track of the customizations we mentioned above. The idea for a solution to this problem was proposed by Bernstein [Ber03], in terms of a script of model management operators, using DIFF to compute differences, MODELGEN to translate and MERGE to integrate. Intuitively, we have to detect the actual differences between the original and the modified target schemas I_1 and I_2 respectively. Then we have to translate these differences back to the specification model (in our case the relational one) and finally integrate the translated differences with the original specification S_1 obtaining a revised specification S_2. The requirement is that we should obtain I_2, if we apply to S_2 the sequence of translations and customizations used to obtain I_1 from S_1. With reference to our example, applying the sequence of operators as described in the algorithm, we produce indeed the relational schema illustrated in the box labeled S_2 in Figure 2.1. Schema S_2 includes new tables *Section* and *Address* corresponding to the new complex elements in I_2. *Department* has a foreign key to *Address* and *Section* to *Course*. Also, attribute *Room* is in *Section* and not anymore in *Course*.

In the existing literature, the proposals for the various operators are not general and accurate enough, as they refer to a rather limited set of models and do not have features that support the description of models, and so the plan proposed by Bernstein has not yet been implemented in a general way.

With the twofold goal of using a different model and of presenting a simpler example, let us consider another scenario. Let us assume we have a high level specification tool that translates ER schemas into relational tables by generating appropriate SQL DDL, allowing some customization. Again, if changes are made to the SQL implementation, then we want them to be propagated back to the ER specification.

This is illustrated in Figure 2.1, where S_1 represents a specification in the ER model and I_1 represents its relational implementation. The customization in the trans-

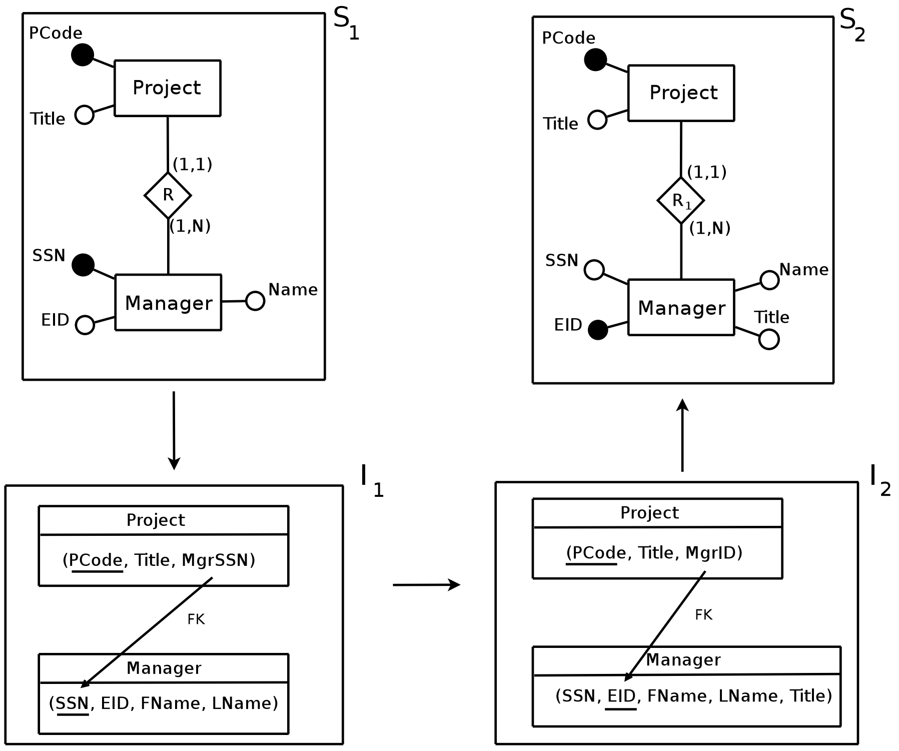


Figure 2.2: A simple scenario for the round-trip engineering problem

lation produces two columns *FName* and *LName* in I_1 for the single attribute *Name* in S_1. Then, if I_1 is modified to a new version I_2, the latter is not coherent with S_1. The main difference between I_2 and I_1 is in the key for the *Manager* table and, as a consequence, in the foreign key structure that refers to it. Also, *Manager* has a new attribute, *Title*. The goal is to find a specification S_2 from which I_2 could be generated, in the same semiautomatic way as I_1 was obtained from S_1.

Indeed, what we want to obtain is an ER schema S_2, which differs from the original one in the attributes of the entity *Manager*: the identifier is *EID* instead of *SSN* and there is the new attribute *Title*.

In the remainder of this chapter we will follow this second example, which will allows us to explain completely the approach, without taking too much space.

The approach

The solution that is proposed includes a definition and implementation of the major model management operators (DIFF, MERGE, and MODELGEN). It is based on our experience in the MIDST platform [ACB06, ACG07, ACT$^+$08], where a model-independent approach for schema and data translation was introduced (with a generic implementation of the MODELGEN operator). MIDST adopts a metalevel approach in which the artifacts of interest are handled in a repository that represents data models, schemas, and databases in an integrated way, both model-independent and model-aware. This is a fundamental starting point, as stated before, in order to be able to define a model management system. This repository is implemented as a multilevel dictionary. Data models are defined in terms of the constructs they involve. A schema of a specific data model is allowed to use only the constructs that are available for that model. In this framework, the *supermodel* is the model that includes the whole range of constructs, so that every schema in every model is also a schema in the supermodel. Then, all translations are performed within the supermodel, in order to scale with respect to the size of the space of models [ACT$^+$08]. In this chapter, we show how the dictionary and the supermodel provide grounds for the model-independent definition of the other operators of interest, namely MERGE and DIFF.

MISM is based on MIDST but extends it in a significant way. We start from MIDST's representation for data models, schemas, and databases and define model management operators by means of Datalog programs with respect to such representation. Specifically, we leverage on the features of MIDST's dictionary for the uniform representation of models as well as the infrastructure for the definition and the application of schema manipulation operators. MISM offers all the major operators, including MERGE, DIFF, and a basic version of MATCH, all implemented in a model-generic way.

The structure of the dictionary also allows for the automatic generation of Datalog programs implementing the new operators, with respect to the given supermodel, in such a way that, if the supermodel were extended, the operators would be automatically extended as well.

Contribution

This is the first proposal for a model-independent platform for model management. Specifically, this the approach offers three main contributions:

- The model-independent definition and implementation of important model management operators. In fact, we define them by means of programs with predicates acting on the constructs of the supermodel.
- The automatic generation of the programs implementing the operators only using the supermodel as input. These programs are valid for any schema defined in terms of model-generic constructs.
- A complete solution to the round-trip engineering problem as a representative of the problems that can be solved with this approach. It is based on a script defined in terms of a convenient combination of our operators and allows a walk through of our implementation.

2.2 Operators

Model management refers to a wide range of problems, which share the need for high level solutions. Therefore many operators have been proposed, depending on the family of problems of interest. Here we concentrate on schema evolution, where proposals [Ber03, BHP00] require MATCH, DIFF and MERGE and, if an explicit representation of models is needed, also MODELGEN. In such proposals, the MATCH operator is used to discover mappings between the elements of the involved schemas.

In fact, mappings play a major role, as they provide the operators with essential information about the relationships between the involved schemas. For example, an operator that computes the difference between two schemas needs to know the correspondences between constructs in order to subtract them correctly. Likewise, an operator that combines schemas must know those correspondences in order to avoid the generation of duplicates.

Here, exploiting our construct-based representation of data models, we can propose definitions of the main operators (DIFF, MERGE, and MODELGEN) that compare constructs on the basis of their names and structures. In fact, we assume that if two constructs have different names or different structures, they should be considered as different.

In this way, as we clarify in the next subsection, our approach considers MATCH as complementary.

We already have an implementation for MODELGEN in our MIDST proposal (and hence in MISM as well), and so we have to concentrate on DIFF and MERGE.

In the rest of this section we will present specifications for these operators that refer to MIDST dictionary, preceded by the discussion of a preliminary notion, equivalence of schema elements. Then, in Section 2.3 we will show how to generate Datalog implementations for them.

Equivalence of schema elements

The basic idea behind the DIFF and MERGE operators is the set-theoretical one. In fact, we can consider each schema as composed of a set of *schema elements* (the actual constructs it involves), and then consider DIFF as a set-theoretic difference (the elements that are in the first schema and not in the second) and MERGE as a union (the elements that are at least in one of the two schemas). In general, we might be interested in comparing schemas that represent the concepts of interest by means of different elements. In such a case, a preliminary step would require the identification or specification of the correspondences between them. This is usually done by means of an application of the MATCH operator, which, in general, can produce correspondences of various types (i.e. one-to-one, one-to-many, or even many-to-many) and may require a human intervention in order to disambiguate or to better specify. Besides, in MIDST context, let us recall that each schema element is represented with respect to a specific model-generic construct (i.e. an element refers to an Abstract, another one refers to an Aggregation and so on): in this sense we say that an element is an *instance of* a construct. Consequently, we distinguish between *construct-preserving* correspondences and *non construct-preserving* ones. The first type maps elements, instances of a certain construct, only to elements that are instances of the same model-generic construct; viceversa, correspondences not satisfying this property belong to the second type. For example in the XML schemas of Figure 2.1 the correspondence between the simple element *Address* and the complex one (again called *Address*), composed of *Street*, *Zip*, and *City*, is not construct-preserving. In fact the address is represented by a simple element in the first schema (i.e. a Lexical in MIDST), while in the second one it requires a complex element (i.e. a StructOfAttributes in MIDST) with its components (i.e. some Lexicals in MIDST). Clearly, non construct-preserving correspondences denote different ways to organize the data of interest and therefore the involved constructs of the two schemas have to be considered as different. On the other hand, constructs that have different names but the same structure while handling the same data, have to be considered as equivalent. These are one-to-one correspondences, which can be discovered manually or by means of a matching system (among the many existing ones [RB01]).

The arguments above lead to a notion of renaming of a schema: given a correspondence c, the *renaming* of a schema S with respect to c is a schema where the names of the elements in S are modified according to c. Then, we have a basic idea of equivalence conveyed by the following recursive statement:

two schema elements are equivalent with respect to a renaming if: (i) they are instances of the same model-generic construct; (ii) their names are equal, after the renaming; (iii) their features (names and properties) are equal; and (iv) they refer to *equivalent elements.*

For the sake of simplicity, we can assume that the renaming is always applied to one of the schemas, in order to guarantee that corresponding constructs with the same type also have the same name. In some sense this would correspond to a *unique name assumption*. Then, equivalence would be simpler, as name equality would be required:

two schema elements are equivalent if their types, names and features are equal and they refer to equivalent elements.

It is important to observe that the definition is recursive, as equivalence of pairs of elements requires the equivalence of the elements they refer to. This is well defined, because the structure of references in our supermodel is acyclic, and therefore recursion is bounded. Let us consider few cases from our running example, namely schemas I_1 and I_2 in Figure 2.2. We have a column *Title* for a table *Project* in both schemas, and the two are equivalent, as they have the same name, the same properties (they are both non-key), and refer to equivalent elements (the tables named *Project*). Instead, the column *Title* of *Project* in I_1 is not equivalent to *Title* of *Manager* in I_2, because *Project* and *Manager* are not equivalent. Also, the two columns named *SSN* are not equivalent, because the one in I_1 is key and that in I_2 is not.

Definitions of the operators

We are now ready to give our definitions and show some examples. According to what we said in the previous section, we assume that suitable renamings have been applied in such a way that a unique name assumption holds. We start with a preliminary notion, to be revised shortly.

Given two schemas S and S', the difference DIFF(S, S') *is a schema S'' that contains all the schema elements of S that do not appear in S'.*

This first intuitive idea must be refined, otherwise some inconsistencies could arise. In fact, it may be the case that a schema element appears in the result of a difference while an element it refers to does not. This causes incoherent schemas with "orphan" elements. With respect to the schemas in our running example, this happens for the column *MgrID* in the difference DIFF(I_2, I_1), which belongs to the result, while the table *Project* does not. Instead we want to have *coherent* schemas, where references are not dangling.

In order to solve this difficulty, we modify our notion of a schema, by introducing *stub elements* (similar to the *support objects* of [Ber03]). Specifically, we extend the notion of *schema element*, by allowing two kinds: *proper elements* (or simply *elements*), those we have seen so far, and *stub elements*, which are essentially fictitious elements, introduced to guarantee that required references exist. We say that a schema is *proper* if all its elements are proper.

According to this technique, the result of DIFF(I_2, I_1) contains the stub version of *Project* in order to avoid the missing reference of *MgrID*.

The definition of the difference should therefore be modified in order to take into account stub elements both in the source schemas and in the result one.

Given S and S', DIFF(S, S') *is a schema S'' that contains: (i) all the schema elements of S that do not appear in S'; (ii) stub versions for elements of S that appear also in S' (and so should not be in the difference) but are referred to by other elements in* DIFF(S, S').

The notion is recursive, but well defined because of the acyclicity of our references.

In the literature [Ber03], the DIFF operator is often used in model management scripts to detect which schema elements have been added to or removed from a schema. Our definition addresses this target. Given an "old" schema S and a "new" one S', the "added" elements (also called the *positive* difference) can be obtained as DIFF(S', S) whereas the "removed" ones (the *negative* difference) are given by DIFF(S, S').

With respect to the running example in Figure 2.2, the negative difference, DIFF(I_1, I_2), contains the columns *MgrSSN* of *Project* and *SSN* (key) and *EID* (non-key) of *Manager*. Column *MgrSSN* belongs to the difference since it belongs to I_1 and there is no attribute with the same name in I_2. Instead, *EID* and *SSN* belong to DIFF(I_1, I_2) because the attributes with the same respective names in I_2 have properties that differ from those in I_1: *EID* is key in I_1 and not key in I_2, whereas the converse holds for *SSN*. The negative difference does not contain the two tables as proper elements, because they appear in both schemas, but it needs them as stub elements because the various columns have to refer to them. The negative difference also includes the foreign key in I_1 since it does not appear in I_2 (the foreign key in I_2 involves different columns).

Similarly, the positive difference includes the columns *MgrID* of *Project* and *SSN* (non-key), *EID* (key) and *Title* of *Manager*, both tables as stub elements, and the foreign key in I_2.

An important observation is that the definition we have given here is model-independent, because it refers to constructs as they are defined in our supermodel. At the same time, it is *model-aware*, because it is always possible to tell whether a schema belongs to a model, on the basis of the types of the involved schema elements. As a consequence, it is possible to introduce a notion of closure: we say that a model management operator O is *closed with respect to a model* M if, whenever O is applied to schemas in M, then the result is a schema in M as well. Given the various definitions, it follows that the difference is a closed operator, because it produces only constructs that appear in its input arguments.

Let us now turn our attention to the second operator of interest, MERGE. We start again with a preliminary definition.

Given S and S', their merge Merge(S, S') *is a schema S'' that contains the schema elements that appear in at least one of S or S', modulo equivalence.*[2]

[2]Technically, both here and in the difference, we should note that schema elements have their identity. Therefore, in all cases we have new elements in the results; so, here, we copy in the result schema the elements of the two input schemas, and "modulo equivalence" means that we collapse the pairs of elements of the two schemas that are equivalent (only pairs, with one element from each schema, as there are no equivalent elements within a single schema).

It is clear that merge is essentially a set-theoretic union between two schemas, with the avoidance of duplicates managed by means of the notion of equivalence of schema elements.

Since our schemas might involve stub elements, as we saw above, let us consider their impact on this operator. Clearly, the operator cannot introduce new stub elements, as it only copies elements. However, stubs can appear in the input schemas, and the delicate case is when equivalent elements appear in schemas, proper in one and stub in the other.[3]

Given S and S', their merge MERGE(S, S') *is a schema S'' that contains the schema elements that appear in at least one of S or S', modulo equivalence. An element in S'' is proper if it appears as proper in at least one of S and S' and stub otherwise. As an example, consider the following schemas, each composed of a single table. S:* Project(PCode, Title) *and S':* Project(PCode, MgrSSN). *Their merge will be another schema S'' containing the table* Project(PCode, Title, MgrSSN). *Notice that the table* Project *and the column* PCode *appear both in S and in S' and, since they are recognized as equivalent, there are no duplicates in S''. The column* Title *appears only in S while* MgrSSN *only in S'; therefore one copy of each is present in the result schema S''. We will see a complete example of* MERGE *in Section 2.4, while discussing the details of our running example.*

For this operator, arguments for model independence and model closure can be made in the same way as we did for DIFF: specifically, only schema elements deriving from schemas S and S' will appear in the result and, consequently, if they belong to a given model, then S'' will belong to that model as well.

For the sake of homogeneity in notation, let us define also the operator that performs translations between models:

Given a schema S of a source model M and a target model M', the translation

[3]Equivalence of elements neglects the difference between stub and proper elements, as it is not relevant in this context.

MODELGEN(S, M') *is a schema* S' *of* M' *that corresponds to* S.

We have discussed at length MODELGEN elsewhere [ACB06, ACT$^+$08]. Here we just mention that this notation refers to a generic version of it that works for all source and target models (the source model is not needed in the notation as it can be inferred from the source schema), thus avoiding different operators for different pairs of models. Indeed, our MIDST implementation [ACG07, ACT$^+$08] of MODELGEN includes a feature that can select the appropriate translation for any given pair of source and target models.

2.3 Model-independent operators in MISM

In this section we show how the definitions of the operators can be made concrete, in a model-independent way, in our tool, leveraging on the structure of its dictionary. The implementation has been carried out in Datalog, and here we concentrate on its main principles, namely the high-level declarative specification, and the possibility of automatic generation of the rules, on the basis of the metalevel description of models.

The Datalog specification of each operator is composed of two parts:[4]

1. equivalence test;

2. procedure application.

The first part tests the equivalence to provide the second part with necessary preliminary information on the elements of the input schemas.

We first illustrate how the equivalence test can be expressed in Datalog, and then proceed with the discussion for the specific aspects of DIFF and MERGE. At the end of the section, we discuss how all these Datalog programs can be automatically generated out of the dictionary.

[4]For the sake of readability we describe them in a procedural way, even if the specification is clearly declarative.

Equivalence test

The first phase involves the implementation of a test for equivalence of constructs, according to the definition we gave in Section 2.2. Given the definition, all we need is a rule for each model-generic construct that compares the schema elements that are instances of such a construct. It refers to two schemas, denoted by the "schema variables" SOURCE_1 and SOURCE_2, respectively. DEST refers to the target schema where the results will be stored. If a rule is applied with reference to only one source (target) schema the keywords SOURCE (DEST) can be omitted. It generates an intensional predicate (a view, in database terms) that indicates the pairs of OIDs of equivalent constructs. As an example, let us see the Datalog rule that compares Aggregations.

```
EQUIV_Aggregation [DEST] (
    OID1: oid1,
    OID2: oid2
)
<-
Aggregation [SOURCE_1] (
    OID: oid1,
    Name: name
),
Aggregation [SOURCE_2] (
    OID: oid2,
    Name: name
);
```

Aggregation has no references (and also no properties) and so the comparison is based only on name equality (verified with the variable *name*). If the names of the two Aggregations are equal, then they are equivalent, and so their OIDs are included in the view for equivalent Aggregations. In the running example, tables *Project* and *Manager* of the two schemas are detected as equivalent since they have the same names, respectively.

The situation becomes slightly more complex when constructs involve references. This is the case for Lexicals of Aggregation (in the running example, the various columns of *Project* and *Manager*).

```
EQUIV_Lexical [DEST] (
    OID1: oid1,
    OID2: oid2
)
<-
Lexical [SOURCE_1] (
    OID: oid1,
    Name: name,
    isIdentifier: isId,
    isNullable: isNull,
    type: t,
    aggregationOID: oid3
),
Lexical [SOURCE_2] (
    OID: oid2,
    Name: name,
    isIdentifier: isId,
    isNullable: isNull,
    type: t,
    aggregationOID: oid4
),
EQUIV_Aggregation (
    OID1: oid3,
    OID2: oid4
    );
```

The first and the second body predicates compare names and homologous properties of a pair of Lexicals, one belonging to I_1 (SOURCE_1) and the other to I_2 (SOURCE_2). Comparisons are made by means of repeated variables (such as *name*, *isId*, *isNull*, *t*). Moreover, as Lexicals involve references to Aggregations (as no column exists without a table, in the example), we need to compare the elements they refer to. The last predicate in the body performs this task by verifying that the Aggregations (tables) referred to by the Lexicals (columns) of I_1 and I_2 are equivalent (i.e. the corresponding pair of OIDs is in the equivalence view for Aggregation).

If the constructs under examination belonged to a deeper level, there would be a predicate to test the equivalence of ancestors for each step of the hierarchical chain. Each predicate would query the appropriate equivalence view to complete the test. Termination is guaranteed by the acyclicity of the supermodel.

Let us observe that the Datalog program generated in this way is model-aware since it takes into account the type of constructs when performing comparisons. In fact, as it is clear in the examples, Datalog rules are defined with specific respect to the type of the constructs to be compared: a Lexical is compared only with another Lexical and so for an Abstract or other constructs.

The program is model generic as well, since the set of rules contains a rule for each construct in the supermodel. Then a given pair of schemas will really make use of a subset of the rules, the ones referring to the constructs they actually involve according to their model.

The DIFF operator

The DIFF operator is implemented by a Datalog program with the following steps:

1. equivalence test (comparison between the input schemas);
2. selective copy.

The first step is the equivalence test we have described in Section 2.3.

As for the second step, there is a Datalog rule for each construct of the supermodel, hence taking into account each kind of schema element: the rule verifies whether the OID of an element of the first schema belongs to a tuple in the equivalence view. If this happens, this means that there is an equivalent construct in the second schema, implying that the difference must not contain it, otherwise the copy takes place. For example, the rule for Aggregations results as follows:

```
Aggregation [DEST] (
    OID: #AggregationOID_0(oid),
    Name: name
)
<-
Aggregation [SOURCE_1] (
    OID: oid,
    Name: name
),
!EQUIV_Aggregation (
    OID1: oid
);
```

The rule copies into the result schema all the Aggregations of SOURCE_1 that are not equivalent to any Aggregation of SOURCE_2. The condition of non-equivalence is tested by the negated predicate (negation is denoted by "!") over the equivalence view; in fact, if the OID of an Aggregation of the first source schema is present in the view, then it has a corresponding Aggregation in the second source schema, and so it must not belong to the difference.

With reference to the running example, let us compute DIFF(I_1, I_2). The rule above represents the computation of the difference with respect to tables. Since in Figure 2.2 both *Project* and *Manager* in I_1 have an equivalent table in I_2, then the difference does not contain any Aggregation.

Consider now the rule for Lexicals (columns):

```
Lexical [DEST] (
    OID: #LexicalOID_0(oid),
    Name: name,
    isIdentifier: isId,
    isNullable: isNull,
    type: t,
    aggregationOID: #AggregationOID_0(oid1)
)
<-
Lexical [SOURCE_1] (
    OID: oid,
    Name: name,
    isIdentifier: isId,
    isNullable: isNull,
    type: t,
        aggregationOID: oid1
    ),
    Aggregation [SOURCE_1] (
        OID: oid1
    ),
    !EQUIV_Lexical (
        OID1: oid
        );
```

It copies into the result schema all the Lexicals of SOURCE_1 that are not equivalent to any Lexical of SOURCE_2.

In the example of Figure 2.2, the Lexical *MgrSSN* has been removed from *Project*. Also, *SSN* of *Manager* is key in I_1 but not in I_2 and the converse for *EID*. Consequently all of the mentioned Lexicals will belong to the difference DIFF (I_1, I_2).

For the sake of simplicity, we have omitted from the above rules the features that handle stub elements. However the actual implementation of the difference requires them in order to address the consistency issues we have discussed in the previous section. The strategy we adopt is the following: when a non-first level element (that is, one with references) is copied, the procedure copies its referred elements too if they are not copied for another reason. Then, unless they are proper parts of the result, the procedure marks the referred elements as stub. The following rule exemplifies this with respect to Aggregations.

```
Aggregation [DEST] (
    OID: #AggregationOID_0(oid),
    Name: name,
    isStub: true
)
<-
Aggregation [SOURCE_1] (
    OID: oid,
    Name: name
),
EQUIV_Aggregation (
    OID1: oid,
    isStub: false
    );
```

If a Lexical (referring to an Aggregation) belongs to the difference, then the referred Aggregation must be copied into the difference as stub (if it has not been copied directly). The rule above copies from the first schema every Aggregation that would not belong to the difference since it has an equivalent (non stub) element in the second schema (which is verified by the predicate over the view, which also contains information on whether the equivalence involves stub elements) and marks it as stub. As for the input, we must subtract schemas with stub elements properly. Thus the selective copy in step 2 must be adapted: it should copy (into the result schema) a non-stub element in the first schema only if the second schema does not contain a non-stub equivalent element. This last condition is tested by a predicate over an equivalence view like the one in the above Datalog rule.

The techniques described refer to the rules for the specification of the difference of schemas. Indeed, as our dictionary includes also a data level which lists all data items that instantiate a given construct, it is interesting to see how the operator could be specified in such a way that the result is a schema, as we saw above, together with the associated data. While working at MODELGEN, we tackled the same issue, and we developed a technique that generated data level Datalog programs out of schema level ones [ACB06]. In such a context, correctness was a delicate issue, as each translation has its own specific features, and the tool administrator has the responsibility of verifying the correctness. Here we are interested in a general program, that implements difference, and therefore we cannot rely upon the approval of a human. However, things are indeed easier, as the difference needs to include all instances of the constructs that appear in the result schema: for example, if the result of DIFF includes table *Manager*, then we need all its instances in the result database, but this is just a copy, as *Manager* is a table in the source schema as well. So, data level rules for DIFF could be produced as rules that copy all instances of constructs, with the condition that the construct appears in the result schema, which is easy to express, as it is indeed the condition in the body of the schema rule. Therefore, while we omit the details for the sake of space, we can safely claim that we can generate correct rules that operate on data from those that operate on schemas.

The MERGE operator

The approach we follow for MERGE is based on the same ideas as the one for DIFF. We code it in terms of Datalog rules defined over the constructs of MIDST supermodel. Rules copy elements of one type to elements of the same type and we guarantee the needed model closure.

The MERGE operator, as defined in Section 2.2, is represented by a Datalog program with the following tasks:

1. equivalence test (comparison between the input schemas);
2. selective copy from the first argument;
3. selective copy from the second argument.

The first step involves the computation of an equivalence view containing the correspondences between the elements of the input schemas.

Assume we are computing $S'' = \text{MERGE}(S, S')$. In step 2 the procedure copies into the destination schema S'' all the elements in S, except those that are stub in S and non-stub in S'. In step 3 the procedure copies all the elements of S' that are not present in S and those that are non-stub in S' and stub in S.

The combination of these two steps implies that in S'' there will not be duplicates of any element. If an element is present both in S and S', in S as a stub and in S' as a non-stub, it will be present in S'' as a non-stub. A stub element will appear in the result as stub as well, if an element is present only in S or S' as a stub or both in S and S' as stub.

In such an implementation of the MERGE, a thorough handling of references is important and we achieve this by means of Skolem functions, which are injective as we said in the previous section.

In fact, it may happen for an element of the result schema to have a stub parent in the first source schema and a non-stub parent coming from the second source schema: let E be an element of S which is copied into the result schema. E has a stub parent P in S and there is another element P' which is the equivalent non-stub element of P in S'. P will not be copied from S, but there will be its equivalent P' coming from S'. As a consequence, the reference of E to P must use an OID that is derived from the OID of P' in S' and not from the OID of P in S. As we have seen for the difference, this logic can be implemented in Datalog on the basis of a predicate over the equivalence views.

By following arguments similar to those for DIFF, we can claim that, from the schema level Datalog programs for MERGE, we can generate programs that implement the operator on data, thus performing the merge of the actual databases (in the internal representation in our dictionary). The reason is that the operator is again a sort of selective copy.

Automatic generation of Datalog programs for the operators

The implementations of both the phases of the operators are based on comparisons and copies of schema elements considered in terms of constructs of the supermodel. We have seen in Chapter 1 that MIDST handles the descriptions of these constructs in a dictionary, defining their names, features and references to one another. An automatic generation of the Datalog programs we have presented is possible and indeed represents a key point of the approach we propose here. Concretely, we propose a new module of MISM, *OpGen*, that automatically generates the rules according to the supermodel constructs. OpGen reads the information in the dictionary about constructs, their references, and their properties, and uses it to produce appropriate Datalog rules in the right order, according to the structure of constructs. As we said in the respective sections, for each operator we can generate data level rules that perform the selective copy of the instances of the involved constructs.

Automatically generated operators are not only model-independent but also supermodel-independent. In fact, in case of extensions to and modifications of the supermodel, all we need is to use OpGen to generate an updated version of the operators.

It is worth noting that our model-generic operators are scalable, since their internal complexity does not depend on the size of the input schemas nor on the number of modifications. In fact, they are generated by OpGen once and work for every possible set of input schemas defined in terms of constructs of MIDST supermodel. Moreover, although more efficient implementations of them could be designed, their application is entirely devoted to the database system which addresses, as a consequence, all the optimization issues.

2.4 A model-independent solution to the round-trip engineering problem

In the previous sections we described the most common model management operators. We have shown that since they are defined over the constructs of MIDST supermodel, they are model-independent; moreover we have shown that it is possible to exploit their model awareness in order to satisfy the model closure property. This implies that solutions to model management problems, given in terms of these operators, are model-independent.

Here we show how our approach can be used to provide a model-independent solution to the round-trip engineering problem, illustrated in the introduction as one of the most representative ones in the model management area.

The general procedure

Consider Figure 2.3: S_1 is the *specification schema* and I_1 the *implementation* schema obtained from S_1 with the application of the transformation (a translation and, possibly, some customizations) *map*$_1$. Let I_2 be a modified version of I_1. The goal is to determine a specification S_2 from which I_2 could be derived.

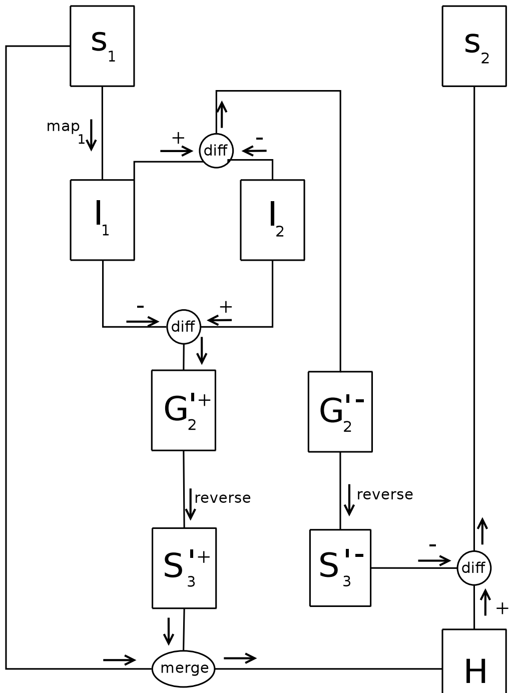


Figure 2.3: A procedure for the round-trip engineering problem

Operationally, we assume that I_1 has been generated from the specification schema by the MODELGEN operator, possibly followed by a customization step; viceversa, we make no specific assumption on how I_2 has been obtained: it could be some transformation (specified by means of a Datalog program or in some other way), or a manual modification or evolution of I_1, or it could even come from an external input.

Then the procedure is as follows.

1. $G_2'^{-} = \text{DIFF}(I_1, I_2)$

   Here we use the DIFF operator to detect which elements of the implementation schema I_1 do not appear in the revised version I_2: these are the elements belonging to I_1 but not to I_2 (i.e. the removed elements).

2. $G_2'^{+} = \text{DIFF}(I_2, I_1)$

   This difference (with parameters swapped with respect to the previous one) allows to compute which elements have been added in the revision which led from I_1 to I_2. In fact, these elements are all the ones present in I_2 but not in I_1.

3. $S_3'^{-}$ is obtained by applying to $G_2'^{-}$ the reverse of the mapping *map*$_1$. The details then depend on the way *map*$_1$ is defined. In the common case where it is an automatic translation from the specification model to the implementation one (an application of MODELGEN), possibly followed by a customization, we have that reverse can be done with MODELGEN as well, with a translation from the implementation model to the specification one. This ignores the possible customizations, under the assumption that changes in I_1 (yielding I_2) do not involve customized elements. In fact, if this is the case, $G_2'^{-}$ will not include the customized elements, since they are removed by the difference step. It should be noted that in general the existence of the inverse of a given translation is not guaranteed. We will discuss this issue later in this section.

4. Similarly for the other difference: $S_3'^{+}$ is obtained by applying to $G_2'^{+}$ the reverse of the mapping *map*$_1$.

5. $H = \text{MERGE}(S_1, S_3'^{+})$

   H is the union of the original specification S_1 with the reversed difference $S_3'^{+}$ containing the added elements. Therefore, H contains all the original elements plus the added ones.

6. $S_2 = \text{DIFF}(H, S_3'^{-})$

   The last operation of the procedure subtracts $S_3'^{-}$ from the temporary result H, because the elements in $S_3'^{-}$ are those that correspond to the elements removed in the implementation.

It is clear that this procedure does not require information about the models of the source schemas, since the operators act at MISM metalevel, dealing with constructs directly, however the model awareness of MISM guarantees the model closure. In fact, in the same way as we do for translations in our previous tool MIDST (see Section 1.1), we apply our operators in the supermodel framework, and the procedure is preceded and followed by copy steps, the first from the specific source model to the supermodel and the second from the supermodel to the specific model, which essentially rename constructs. An example should get the meaning across: suppose the specification data model is ER, while the implementation belongs to the relational model. Before applying the DIFF between I_1 and I_2, we rename all the elements in terms of constructs of the supermodel. After this step, there is no need to take into account the model-specific constructs anymore and the procedure can continue with respect to model-generic constructs only. Then, since the operators are defined in such a way that the difference between two schemas of a model belongs to that model, then we are guaranteed that the two differences in the procedure belong to the relational model as well. Finally, we apply MERGE and DIFF on ER schemas. These operators work independently of the model. However, we are sure that the results will also belong to the ER model because, as we have illustrated, the operators do not add any new element.

Moreover, it is important to observe that, if S_1, I_1 and I_2 are proper (and coherent,[5] as we always assume) schemas, then the result S_2 of the script is a proper schema as well. Consider the last two steps of the procedure ((i) $H = \text{MERGE}(S_1, S_3'^{+})$, (ii) $S_2 = \text{DIFF}(H, S_3'^{-})$): S_1 is assumed to be proper (the script starts from a specification without stubs). $S_3'^{+}$ contains added elements which may refer to stub parents. However, as I_1 and I_2 are coherent, we have that non-stub equivalents for these stub parents are already present in S_1. Therefore H is proper. $S_3'^{-}$ contains the removed constructs. Then, as I_2 is coherent, in $S_3'^{-}$ we cannot come across the removal of

[5]As we said in Section 2.2, a schema is coherent if all its constructs have no dangling references to other constructs.

parent elements when their descendants are preserved. Therefore S_2 is proper.

In the above procedure, we have referred to applications of MODELGEN from the specification model to the implementation one and viceversa, as if they were one the inverse of the other. This need not be always the case, because models have different expressive power. However, from the practical point of view, we have reasonable solutions, as follows. A preliminary observation is that our translations can be seen as schema mappings where the correspondences are represented by Skolem functions. In general, schema mappings are not always invertible according to the strict definition, but in the literature there are proposals for relaxed constraints guaranteeing the existence of a kind of inverse mapping. According to Fagin et al. [FKPT07] a Local As View (LAV) schema mapping, having a set of Tuple Generating Dependencies (TGDs) where their left-hand sides are singleton, always admits a *quasi-inverse* corresponding mapping. Let us consider a mapping m and a source schema S; applying m to S we obtain another schema T. A quasi-inverse mapping does not permit to reobtain S (with its original data) from T, however, it allows to obtain a schema S^* such that applying m to it we have T again (with all its data). In our approach the only translation rules dealing with the actual data are the ones involving Lexicals. All these rules are LAV TGDs and therefore the whole translation is a LAV schema mapping and so each translation admits at least a quasi-inverse one that is part of the MISM repository. In general, a translation can lead to loss of information (i.e. when we translate a model into a less expressive one); in such cases it is not possible to define an inverse translation, but only a quasi-inverse one. It is worth noting that this loss of information has already been accepted by the user of the system when performing the first translation (from the specification to the implementation). Moreover, this is the only loss of information of the whole process. In fact after the first translation, it is possible to apply the quasi-inverse translation and the direct one repeatedly always obtaining the same schemas (with the same data). The inverse (quasi-inverse) translation does not cause loss of information even if it turns a model into a more expressive one. In fact, the input schema of the inverse translation has been obtained from a schema of a less expressive model; therefore it contains only structures that can be represented in such a model.

Application of the round-trip solving procedure

Now we present the details of the application of the round-trip solving procedure to the case already shown in Figure 2.2. The specification domain is the ER model, while the implementations are relational schemas. It is a common scenario in which high level specifications are conceptually designed with an ER schema. The implementation, which in this situation belongs to the relational model, is then derived from the ER through the application of a translation rule.

The various steps are shown in Figure 2.4. Schema S_1 is composed of two entities, *Project* and *Manager*, and has a relationship R between them. *PCode* and *Title* are *Project* attributes (*PCode* is key), while *SSN*, *Name* and *EID* are *Manager* attributes (*SSN* is key).

Map$_1$ is implemented in two parts: a first part of the transformation is represented by ER-to-relational translation rule. A second part of it consists of the customization step which splits *Name* into *FName* and *LName*.

The transformation from the old to the new implementation modifies the table *Project* by changing the name of its column *MgrSSN* (to *MgrID*); it also modifies the *Manager* by adding the column *Title* and changing its key (from *SSN* to *EID*). The foreign key that in I_1 connects the column *SSN* with the table *Manager*, does not exist anymore, it is replaced by a new foreign key from the column *MgrID* of *Project* to the table *Manager*.

The first step of the solving procedure is the double application of the DIFF rules to I_1 and I_2 which yields $G_2'^{-}$ (negative difference) and $G_2'^{+}$ (positive difference), as we have already seen with examples for the operator in Section 2.2.

Then each semi-difference is reversed with the application of the MODELGEN operator, with the ER model as a target. In the case under examination, the reverse translation is simple, while in general it might be much more complex. Notice that in

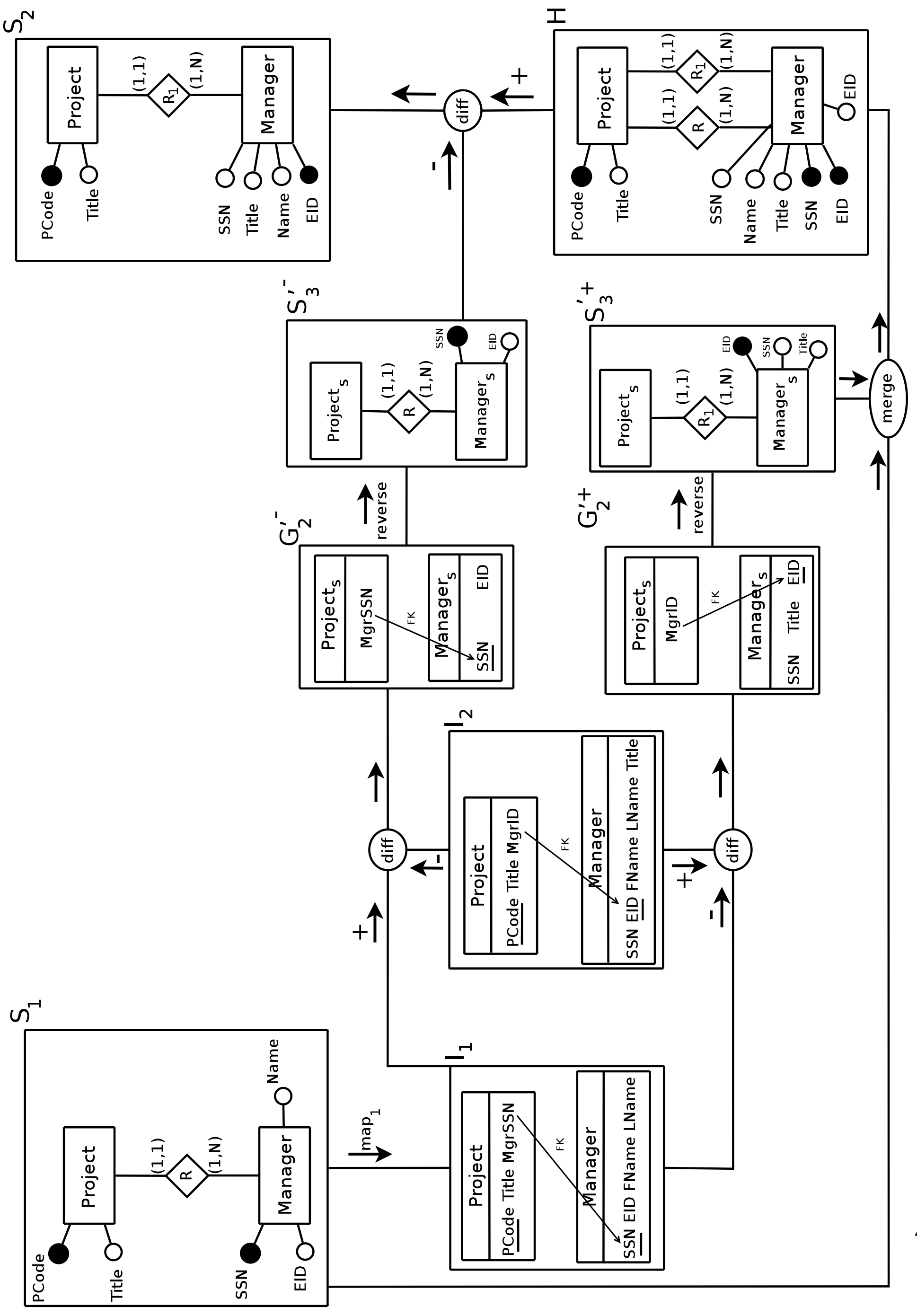


Figure 2.4: An example of application of the round-trip solving procedure

the application of the reverse rule, the stubness property of elements is preserved, then for example the entity *Project* in $S_3'^+$ is stub as well as in $G_2'^+$. Notice that the foreign key of $G_2'^-$ is reversed into the relationship R (that is the same as in S_1,[6] while the foreign key of $G_2'^+$ is reversed into the relationship R_1 (that is different from the one in S_1).

Now we have three different versions of the specification: the original one, S_1, together with $S_3'^-$, including all the elements that have to be removed, and $S_3'^+$, containing all the added elements.

The set-oriented merge of schemas S_1 and $S_3'^+$ leads to an updated specification, H, containing all the initial elements plus the added ones. Then in H we have *Project* with *PCode* (coming from S_1) and *Title* (from S_1) (the table *Project* is not stub anymore since it comes from S_1); moreover, there is the table *Manager* (non-stub for the same reason as *Project*) with the attributes *Name* (coming from S_1), *SSN* (from S_3^+), *SSN* (key) (from S_1), *EID* (from S_1), *EID* (key) (from S_3^+) and *Title* (from S_3^+). H also contains two relationships, R (coming from S_1) and R_1 (from $S_3'^+$).

Finally, we need to subtract from H all the non-stub elements in $S_3'^-$. Therefore, *SSN* (key) and *EID* are not present in the obtained result S_2. The relationship R of H is also present in $S_3'^-$, so the only relationship between *Project* and *Manager* in S_2 will be R_1.

2.5 Conclusions

In this chapter we moved from the classical Bernstein's definition of model management and illustrated MISM, a model management system based on MIDST. As a concrete example of a major model management problem we considered round-trip engineering; indeed, many other problems such as data integration, forward engineering and so on, can be considered a specialization of it. MISM, also presented

[6]We can get back the "original" name because each construct has a name property; hence also the foreign key has a name property (not shown in figure) in our construct-based representation; in detail, we instantiated the name of the foreign key during the translation from S_1 to I_1 and we did the same during this step.

in [ABBG08, ABBG09b], is the technical opportunity to describe a completely model-independent but model-aware approach to model management that, to the best of our knowledge, is the first proposal in this direction.

CHAPTER 3

A run-time approach to model and schema translation

MIDST, the platform we described in the previous chapters, was conceived to perform translations in an off-line fashion. In such an approach, the source database (both schema and data) is imported into a repository, where it is stored in a universal model. Then, the translation were applied within the tool as a composition of elementary transformation steps, specified as Datalog programs. Finally, the result (again both schema and data) was exported into the operational system.

In this chapter we illustrate a new, lightweight approach where the database is not imported. MIDST-RT needs only to know the schema of the source database and the model of the target one, and generates views on the operational system that expose the underlying data according to the corresponding schema in the target model. Views are generated in an almost automatic way, on the basis of the Datalog rules for schema translation.

The proposed solution can be applied to different scenarios, which include data and application migration, data interchange, and object-to-relational mapping between applications and databases.

3.1 The scenario

The problem of translating schemas between data models is acquiring progressive significance in heterogeneous environments. This is motivated by the fact that applications are usually designed to deal with information represented according to a specific data model, while the evolution of systems (in databases as well as in other technology domains, such as the Web) led to the adoption of many representation paradigms. For example, many database systems are nowadays object-relational (OR) and so it is reasonable to exploit their full potentialities by adopting such a model while most applications are designed to interact with a relational database. Also, object-relational extensions are often non-standard, and conversions are needed. Moreover, there is currently a significant adoption of XML repositories that manage native XML data. This fact has increased the heterogeneity of representations.

In general, the presence of several coexisting models introduces the need for translation techniques and tools. In fact, *Model Management* (Bernstein [Ber03]), a high-level approach to meta data management that offers high-level operators to deal with schemas and mappings, includes an operator (called *ModelGen*) for translating schemas from a model to another.

MIDST adopts a metalevel approach towards translations by performing them in the context of a universal model (called the *supermodel*[1]), which allows for the management of schemas in many different data models. The tool has been experimented with many models, including relational, object-oriented (OO), object-relational, entity-relationship (ER), XML-based, each in many different variants. Translations are organized according to the following pattern. First, the source database is imported into the tool and described in its dictionary in terms of the supermodel. Then, the translation is performed by means of a sequence of elementary steps (rules), each dealing with a specific aspect to be eliminated or transformed. Finally, the obtained database

[1]The use of a universal model has been adopted, in different forms, by the various projects mentioned above [ACT+08, AT96, Hai06, MP99, MBM07b, PT05, SM08].

is exported into the target operational system.[2] This approach provides a general solution to the problem of schema translation, with *model-genericity* (as the approach works in the same way for many models) and *model-awareness* (in the sense that the tool knows models, and can use such a knowledge to produce target schemas and databases that conform to specific target models). However, as pointed out by Bernstein and Melnik [BM07], this approach is rather inefficient. In fact, the necessity to import and export a whole database in order to perform translations is out of step with the current need for interoperability in heterogeneous data environments.

Here we illustrate an evolution of MIDST, leading to a new platform, MIDST-RT [ABB+12, ABBG09a]: it is based on a runtime approach to the translation problem, where data is not moved from the operational system and translations are performed directly on it. What the user obtains at runtime is a set of views (defining the target schema) conforming to the target model. The approach is model-generic and model-aware, as it was the case with MIDST, because we leverage on MIDST dictionary for the description of models and schemas and also on its key idea of having translations based on the supermodel, obtained as composition of elementary ones. However, the management of the involved data is completely different. In fact, the import process concerns only the schema of the source database. The rules for schema translation are used here as the basis for the generation of views in the operational system. In such a way, data is managed only within the operational system itself. In fact, our main contribution is the definition of an algorithm that generates executable data level statements (view definitions) out of schema translation rules.

A major difference between an off-line and a runtime approach to translation is the following. For an off-line approach, as translations are performed within the translation tool (MIDST in our case), the language for expressing translations can be chosen once, for all models. A significant difficulty is in the import/export components, which have to mediate between the operational systems and the tool repository, in terms of

[2]We use the term *operational system* to refer to the system that is actually used by applications to handle their data.

both schemas and data. In fact, in the development of the original, off-line version of MIDST, a lot of effort was devoted to import/export modules, whereas all translations were developed in Datalog. In a runtime approach, instead, the difficulties with import/export are minor, because only schemas have to be moved, but the translation language depends on the actual operational systems. In fact, if there is significant heterogeneity, then stacks of languages may be needed (involving for example, SQL, SQL/XML, XQuery, and combinations of them). Also, different dialects of the various languages may exist, and our techniques need to cope with them.

In order to handle the heterogeneity of the involved languages, we propose an approach that, after a preliminary abstract representation, first generates views organized according to the target model, but independent of the specific languages, and then actually concretizes them into executable statements on the basis of the specific language supported by the operational system.

In this Chapter we describe a general solution to the language independent step, whereas for the final one we concentrate on SQL, with respect to a set of models that include many variations of the object-relational model and of the relational one, and their extension with XML.

3.2 Application cases

The main result of our work is the ability to define views over the operational system, in order to execute a light transformation that needs only to import the source schema in our dictionary. The meaning of “view” depends on the operational system: for an RDBMS or an ORDBMS, a “view” is a stored query leading to a virtual table that shows data in a different way; for an object-oriented language, a “view” is a set of objects that reference each other; for the Web, a “view” can be an XML document that shows data extracted from a relational database.

In this section we briefly describe some representative problems in this context and explain how MIDST-RT can support them.

Data and application migration

“Data-migration” is a process of data movement between different storage systems (and different technologies) caused by changes in the technology or in the organization of data. In order to obtain an effective migration, it is important to keep in mind that applications have to be migrated as well. As argued by Brodie and Stonebraker [BS95], migration needs to be incremental, and some legacy functions should coexist with the newly developed ones. MIDST-RT can support this, by offering two different interfaces to the same database. Let us consider the following practical problem in order to see how to solve it using MIDST-RT:

- let A be an ORDBMS used by some applications of an enterprise;
- the enterprise decides to change its commercial partner, so it will use B, another DBMS (with a different version of the object-relational model or with the relational one);
- given the actual differences between systems A and B, the original schema is not compatible with B and so current applications do not work with it. New applications need to be developed and tested, without interrupting operation.

Figure 3.1 explains how we can support this problem:

1. MIDST-RT can generate a set of views over system A that show a schema compatible with the specification of system B. In such a way, the enterprise can gradually update its applications: the modified components will use the new schema shown by the set of generated views, while the other ones will use the original schema;
2. then, after all applications have been modified, the data can be actually migrated, by executing the same queries that define the views, this time materializing their results.

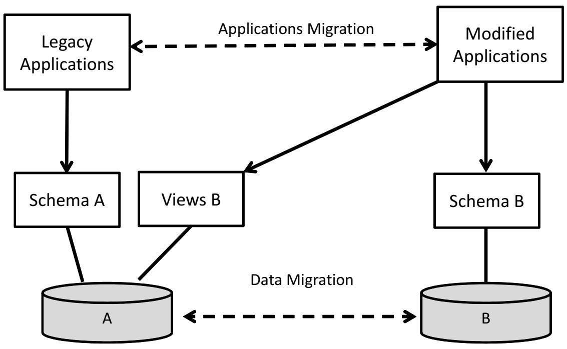


Figure 3.1: The data migration scenario

It is important to observe that this approach would support the intermediate phase where the old data exist together with the new schema, while off-line or data-exchange approaches [ACT$^+$08, HHH$^+$05, Hai06, SM08] would support only the last step.

XML for data interchange

XML is widely used in the process of data movement between applications or DBMSs, especially via a network. Thus, a user can benefit from the usage of XML formats for different reasons: she can migrate a database using the network, she can allow the communication between incompatible systems, she can use such an XML file as an input for an application, and so on. MIDST-RT is able to create an XML view over a relational or object-relational database. We talk about an "XML view" because we perform a runtime translation: first, we do not import data into MIDST-RT, but only the schema of the source database; then, we produce executable statements, so the XML file is always up to date even when the source database is frequently updated. As an example, we can consider the following simple scenario. A user has a relational database and she wants to send data to a Java application through the network. She needs to produce an XML document that contains such a data, and then she needs a framework for the marshalling/unmarshalling of the document. MIDST-RT helps the creation of the XML document in a flexible way with three interesting features which differentiate this approach from the data exchange one [MHH00]: the dynamic generation, the handling of various source models and the possibility of customizing translations. Such a document will be processed by the destination application, possibly by taking advantage of the document schema (XSD).

In Subsection 3.7 we will show a concrete example for this scenario by using MIDST-RT.

O/R mapping

The need for mapping object-oriented applications and relational databases arises in many contexts [MAB07, MBM07b], and various technologies have been developed to support it. The problem can be seen in two major forms: (i) given a relational schema (or an object-relational one) it is convenient to produce object-oriented wrappers, that is, software artifacts that ensure an object-oriented access to a relational database; (ii) given a set of classes that define objects in an object-oriented language it is often useful to obtain the schema of a relational database for the persistent storage of the corresponding data.

As far as the first scenario concerns, MIDST-RT can contribute with the generation of wrappers, which can be seen as a form of views, with benefits in the flexibility both in the source model (many variations of the relational and object-relational ones) and in the target model (different languages and programming conventions), as well as in the mapping (which can be customized, for example for performance issues). We will see in Subsection 3.7 a concrete example for this scenario.

In the second scenario we want to automatically generate a database from a set of classes. Even with respect to existing technologies that support this problem (such as JDO [Jor03], Hibernate [BK06], or ADO.NET Entity Framework [MAB07]), or in combination with them, MIDST-RT can provide specific benefits:

- flexibility in the database management, thanks to the knowledge of different representations of the object-relational model;

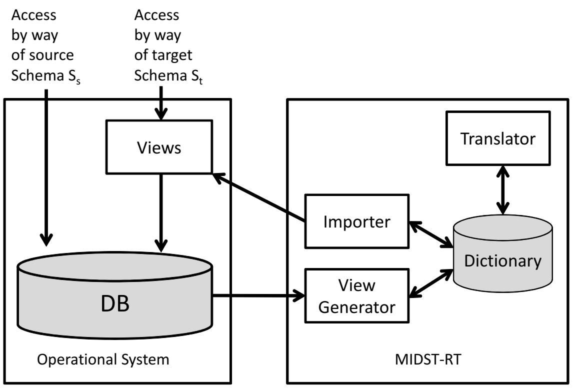


Figure 3.2: The runtime translation procedure

- flexibility in the definition of the target schema, thanks to the possibility of transforming the source schema before the creation of the mapping.

At the moment, we have not implemented this scenario in MIDST-RT, but we are working on it, since it is an important application for our tool. This can be realized by introducing an object-oriented importer that translates a set of Java or C# classes into MIDST-RT internal representation.

3.3 Outline

The starting point for this work is MIDST [ACT+08], a platform for model-independent schema and data translation based on a meta-level approach over a wide range of data models as described in Chapter 1.

Here is described a new approach and an enhanced version of the platform (called MIDST-RT) which enables the creation of executable statements generating views in the operational system. Let us illustrate the runtime translation procedure, by following the main steps it involves, with the help of Figure 3.2:

1. given a schema S_s (of a *source* model M_s) in the operational system, the user (or the application program) specifies a *target* model M_t;

2. schema S_s (but not the actual data) is imported into MIDST-RT, and specifically in its dictionary, where it is described in supermodel terms;

3. MIDST-RT selects the appropriate translation **T** for the pair of models (M_s, M_t), as a sequence of basic translations available in its library;

4. the schema-level translation **T** is applied to S_s, still within the tool, to obtain the target schema S_t (according to the target model M_t);

5. on the basis of the schema-level translation rules in **T**, MIDST-RT generates views in the specific language available in the operational system;

6. MIDST-RT exports and executes the produced statements over the operational system, in order to create a set of views that perform the translation.

Let us observe that steps 1-4 appear also in the previous version of MIDST, whereas 5 is completely new, in all its phases, and clearly significant. Step 6 is a revised form of the export step of MIDST: it exports and executes the view creation statements, rather than exporting the data.

As a running example, let us consider an environment where some applications interact with an object-relational database. Then, assume we want to write an application that uses the same data but interacting with a relational data model. We can consider a version of the OR model that has the following features:[3] tables, typed tables (i.e. tables with identifiers), references and foreign keys between typed tables and generalizations over typed tables.

In this scenario, our tool generates relational views over the object-relational schema, which can be directly used by the new application program.

[3]This is just a possible version of the OR model, and our tool can handle many others.

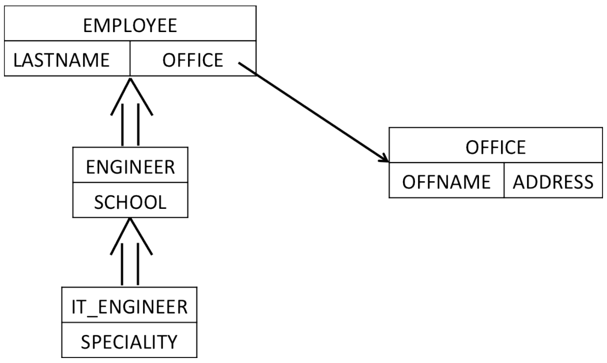


Figure 3.3: An object-relational schema

A concrete case for this example involves the OR schema sketched in Figure 3.3. The boxes are typed tables: employee (EMP) is a generalization for engineer (ENG), which is in turn a generalization for IT-engineer (IT_ENG); office (OFFICE) is referenced by employee.

The goal of the runtime application of MIDST is to obtain a relational database for this, such as the one that involves the following tables with the foreign keys suggested by the names of the attributes (details omitted for the sake of space):[4]

OFFICE (<u>OFFICE_OID</u>, offName, city)
EMP (<u>EMP_OID</u>, lastName, OFFICE_OID_fk)
ENG (<u>ENG_OID</u>, school, EMP_OID_fk)
IT_ENG (<u>IT_ENG_OID</u>, specialty, ENG_OID_fk)

Given the schema in Figure 3.3, our tool first imports it into its dictionary. Then, given the specification of the target model (the relational one), the tool automatically selects an appropriate schema-level translation, which is a sequence of basic translations, each specified by means of a Datalog program. The user can customize the

[4]As it is well known, there are various ways to map generalizations to tables, and this is one of them.

proposed sequence, in order to execute a personalized translation. The customization may include the addition, removal or reorder of the basic translations of the sequence chosen by the tool.

In the example, the schema-level translation performs the following tasks: it first eliminates the multiple levels of generalizations (in the example, the one between ENG and EMP and the one between IT_ENG and ENG) and then transforms the typed tables (all tables in the source) into value-based tables. In MIDST this would be done in four steps, with a first Datalog program for the elimination of generalizations and a fourth one for the transformation of typed tables into value-based ones with two auxiliary intermediate steps for the introduction of keys and the replacement of references with foreign keys, respectively. The tool we propose here generates a set of view statements for each of these Datalog programs.

The following is a sketch of one of the view definitions generated in the first step:

```
CREATE VIEW ENG_A ... AS
    SELECT ... SCHOOL, ... EMP_OID
    FROM ENG ;
```

We use the name ENG_A to distinguish the new version from the original one.[5] View ENG_A extends ENG with a supplementary attribute, EMP_OID. It implements a strategy for the elimination of generalizations, where both the parent and child typed tables are kept, with a reference from the child to the parent.

In Section 3.5 and 3.6, we will see in detail how we produce views of this kind, by showing the principles, the complete description of the algorithm, and the specific details that are needed for the SQL statements.

3.4 Translations

As described in Chapter 1 MIDST translation procedure is based on the generic constructs defined in the *supermodel*. Let us illustrate the main constructs and the main

[5]In the technical sections, we use this convention, with the suffix, when needed for the sake of readability and conciseness. In the tool, names are repeated and distinguished by using different schema names.

Metaconstruct	**Relational**	**OR**
Abstract	-	typed table
Lexical	column	column
BinaryAggregationOfAbstracts	-	-
AbstractAttribute	-	reference
Generalization	-	generalization
Aggregation	table	table
ForeignKey	foreign key	foreign key
StructOfAttributes	-	structured column

Figure 3.4: Simplified representation of MIDST metamodel

translation rules used in the running example, the object-relational schema of Figure 3.3. Figure 3.4 reports a list of MIDST for Relational and ObjectRelational models. Each of the typed tables (EMP, ENG, IT_ENG and OFFICE) is seen as an Abstract in the supermodel. Then, each column of the typed tables (Specialty for IT_ENG, School for ENG, LastName for EMP, OffName and City for OFFICE) is a Lexical and is related to the corresponding Abstract. Similarly, reference fields (Office in this case) are modeled as AbstractAttributes (of EMP in the example). Finally, Generalizations appear in the supermodel: ENG is a child of EMP and IT_ENG is a child of ENG.

With reference to our running example, let us take into account the translation from the version of the OR model we are considering towards a classical relational model. In MIDST [ACT$^+$08] this could be done as a process in four steps:

A elimination of generalizations;

B generation of identifiers for typed tables;

C elimination of reference columns with the introduction of value-based columns and foreign keys;

D transformation of typed tables into tables.

In each translation step we copy, with a simple "copy rule", all the constructs that are not modified. For example, in order to copy an Abstract, we have the *copy-abstract* rule (R_1):

```
R1     Abstract (
           OID: SK2(oid),
           Name: name
       )
   <-
       Abstract (
           OID: oid,
           Name: name
       );
```

The tool has a copy rule for each construct automatically generated out of the definition of the supermodel.

When actual transformations are needed, rules are more complex. Let us illustrate the main points.

As for Step A, there are various ways to eliminate generalizations. Let us refer to the one that keeps both the parent and the child typed tables and connects them with a reference. This requires that we copy all typed tables with their columns and then add a new column for each child typed table with a reference to the respective parent typed table. In terms of MIDST constructs, this means that, for each Generalization between two Abstracts, an AbstractAttribute (a reference column) referring to the parent Abstract must be added to the child Abstract. The Datalog rule implementing this last step is the following (in the following denoted as R_4, or *elim-gen*):

```
R4     AbstractAttribute (
           OID: SK3(genOID, childOID),
           Name: name,
           IsNullable: "false",
           AbstractOID: SK2(childOID),
           AbstractToOID: SK2(parentOID)
       )
   <-
       Generalization (
           OID: genOID,
           ParentAbstractOID: parentOID,
           ChildAbstractOID: childOID
           ),
          Abstract (
              OID: parentOID,
              Name: name
           ),
          Abstract (
              OID: childOID
          );
```

Let us observe that the Skolem functor SK3 has two arguments, because we need to create a new AbstractAttribute for each Generalization and for each of its child Abstracts: indeed a Generalization may have various children and an Abstract may be a child in various Generalizations.

In order to obtain a coherent schema we also need to copy all the constructs in the schema, other than generalizations. This is done by the *copy-abstract* rule (R_1) we have seen above, together with similar ones for the other constructs, *copy-lexical* (R_2) and *copy-abstractAttribute* (R_3) reported below.

```
R2        Lexical (
              OID: SK7(lexOID),
              Name: name,
              IsNullable: isN,
              IsIdentifier: isI
          )
     <-
          Lexical (
              OID: lexOID,
              Name: name,
              IsIdentifier: isI,
              IsNullable: isN,
              AbstractOID: absOID
          ),
          Abstract(
              OID:absOID
          );


R3        AbstractAttribute(
              OID: SK8(oid),
              Name: name,
              IsNullable: isN,
            AbstractToOID: SK2(absToOID),
            AbstractOID: SK2(absOID)
        )
    <-
        AbstractAttribute (
            OID: oid,
            Name: name,
            IsNullable: isN,
            AbstractToOID: absToOID,
            AbstractOID: absOID
        ),
        Abstract(
            OID:absOID
        ),
        Abstract(
            OID:absToOID
        );
```

Step B is needed because it is not guaranteed that typed tables (in the OR model) have key attributes, whereas, in order to transform references into value-based correspondences (subsequent Step C), keys are a precondition. The following Datalog rule (R_5), where the "!" character denotes a negation, implements this strategy: for each Abstract without any identifier, it generates a new key Lexical for it.

```
R5      Lexical ( OID: SK4(absOID),
              Name: name + "_OID",
              IsNullable: "false",
              IsIdentifier: "true",
              Type: "integer",
              AbstractOID: SK2(absOID)
        )
    <-
        Abstract (
              OID: absOID,
              Name: name
        ),
        ! Lexical (
              IsIdentifier: "true",
              AbstractOID: absOID
        );
```

As in the previous step, we need copy rules for all the constructs in the model (the same as above, R_1, R_2, R_3).

Step C replaces reference columns with value-based ones and connects them to the target table with a referential integrity constraint. The following rule (R_6) specifies this: for each AbstractAttribute (reference), it replicates the key Lexicals of the referred typed table into the referring one.

```
R6      Lexical (
              OID: SK5(oid,lexOID),
              Name: lexName,
              IsIdentifier: "false",
              Type: type,
              AbstractOID: SK2(absOID)
        )
    <-
        AbstractAttribute (
              OID: oid,
              AbstractOID: absOID,
              AbstractToOID: absToOID),
        Lexical (
              OID: lexOID,
              Name: lexName,
              AbstractOID: absToOID,
              IsIdentifier: "true",
              Type: type
        ),
        Abstract(
              OID:absOID
        ),
         Abstract(
              OID:absToOID
        );
```

Here we just need the application of two copy rules (R_1 and R_2).

Finally, in Step D, typed tables are eliminated and this is simply performed by means of two Datalog rules. The first translates Abstracts into Aggregations (R_7), the second transforms Lexicals referring to Abstracts into Lexicals referring to Aggregations (R_8). We omit R_7 and R_8 for sake of space, as they would not add much to the discussion.

With respect to the running example of Figure 3.3, we have the following:

- Step A eliminates the hierarchies, hence connects ENG to EMP with a reference and IT_ENG to ENG with another reference;
- Step B creates an identifier for each of the typed tables: EMP_OID for EMP, ENG_OID for ENG, and so on;
- in Step C, references are translated into value-based correspondences: a new Lexical EMP_OID_fk is added to ENG, with foreign key constraint towards the identifier EMP_OID of EMP; similarly OFFICE_OID_fk is added to EMP and ENG_OID_fk to IT_ENG, each with the appropriate foreign key;
- finally, Step D performs the actual translation of EMP, ENG, IT_ENG and OFFICE into tables.

The final result is indeed the relational schema we have already seen in Section 3.3:

OFFICE (<u>OFFICE_OID</u>, offName, city)
EMP (<u>EMP_OID</u>, lastName, OFFICE_OID_fk)
ENG (<u>ENG_OID</u>, school, EMP_OID_fk)
IT_ENG (<u>IT_ENG_OID</u>, specialty, ENG_OID_fk)

3.5 Generating views

As we said, the core goal of the runtime translation procedure is to generate executable statements defining views. This is obtained by means of an analysis of the Datalog programs used to translate schemas as discussed in Section 3.4.

In this section we discuss the major ideas of how views are constructed: which views, which components for them, where values come from and how they have to be correlated if needed (in the relational case, in SQL terms: which views, and, for each of them, which columns, which sources in the `FROM` clause and which join conditions). Then, in the next section, we will discuss the details in terms of a complete algorithm.

The general approach

The first issue to be considered is how to find which views are needed in a translation step, on the basis of the Datalog program that implements it. A key idea in this respect is a classification of MIDST metaconstructs (those in Figure 3.4) according to the role they play. There are three categories: *container*, *content*, and *support* constructs.[6] Containers are the constructs that correspond to sets of structured objects in the operational system (i.e. Aggregations and Abstracts corresponding to tables and typed tables, respectively). Content constructs represent elements of more complex constructs, such as columns, attributes, or references: usually a field of a record (i.e. Lexical and AbstractAttribute) in the operational system. Support constructs do not refer to structures where data are logically stored in the system (for example relations), but are used to model relationships and constraints between them in a model-independent way. Examples are Generalizations (used to model hierarchies) and ForeignKeys (used to specify referential integrity constraints). This sharp distinction is not sufficient in practice, since there are some constructs that can be content and container at the same time: we call them *dual* constructs. For example, we model nested structures using the metaconstruct StructOfAttributes: a StructOfAttributes is a content for the construct in which it is contained (an Abstract or another StructOfAttributes) and a container for the constructs it aggregates.

In turn, Datalog translation rules can be classified according to the construct their head predicate refers to. Therefore, we have *container-* (for example, rules R_1 and R_7 in Section 3.4), *content-* (all other rules in Section 3.4) , *support-* and *dual-generating* rules.

[6]This classification shares some similarity with that proposed by McBrien and Poulovassilis [MP99], which has however a different goal.

The introduction of this classification is motivated by the observation that in all models we have constructs that have an independent existence (and are used to organize data or to represent real-world concepts), other constructs that exist only as components of independent constructs (and maintain component information), constructs that play both these roles, and finally constructs that describe properties of constructs of the previous two categories. These are the four categories we have just illustrated: container, content, dual, and support, respectively.

Exploiting the above observations, the procedure defines a view for each container construct, with fields that derive from the corresponding content (and dual) constructs. Instead, as support constructs do not store data, they are not used to generate view elements (while they are kept in the schemas). More precisely, given a Datalog schema rule $H \leftarrow B$, if H refers to a container construct, we will generate one view for each instantiation of the body of the rule. If H refers to a content or a dual, then we define a field of a certain view.

We will present the translation procedure with its technical details in Section 3.6. In the rest of this section, we first illustrate the procedure with reference to the running example, and then discuss two major issues in the procedure, namely: (i) the provenance of data (that is, where to derive the values from or how to generate them) for the single field and (ii) the appropriate combination of the source constructs, which, from a relational point of view, corresponds to a join.

Let us consider the running example again. Step A includes rules R_1, R_2, R_3, R_4. The only container-generating rule is R_1, which copies all the typed tables, hence we generate a view for each typed table of the operational system: EMP_A, ENG_A, IT_ENG_A and OFFICE_A.[7]

The other rules are content-generating. Rules R_2 and R_3 copy Lexicals (simple fields) and AbstractAttributes (references), respectively. From rule R_2, the procedure infers the owner view, name, and type for each field. For AbstractAttributes the procedure works likewise (rule R_3) with the addition that it has to handle the values encoding

[7]As we said, we use the suffix here to distinguish the versions of tables and views in the various steps.

the references between constructs in an object-oriented fashion.

The main rule of Step A is R_4, which eliminates generalizations by maintaining the parent and the child and connecting them with a reference. Here the problem of data provenance for fields is evident: while in rules R_2 and R_3 the values are copied from the source fields, in rule R_4 an appropriate value that links the child table with the parent one has to be generated. We will discuss this issue later in this Section.

Let us now extend the same reasoning to the non-copying rules of the other steps.

In Step B we generate a key attribute for each typed table using rule R_5. It is a content-generating rule since it generates a key Lexical for every Abstract without an identifier. Hence we add another field to the views that correspond to those Abstracts.

Once Step B has guaranteed the presence of a key, in Step C we translate references into value-based (foreign-key) correspondences.[8] Rule R_6 addresses the need to copy the identifier values of the referred construct into the referring one in order to allow for the definition of value-based correspondences. It implies the addition of a new field to the view that corresponds to the referring Abstract.

Step D is simpler, since the only transformation involves turning typed tables into tables once they do not have any generalizations nor references and the presence of identifiers is guaranteed. The issue is then limited to the internal representation of views handled by the operational system. In fact, many systems have both *views* and *typed views*, and so we have to transform the former into the latter, or vice versa, according to the target model.

This procedure does not depend on the specific constructs nor on the operational system or language. It is not related to constructs because we only rely on the concepts of container and content to generate statements. Other constructs may be added to MIDST supermodel without affecting the procedure: it would be sufficient to classify them according to the role they play (container, content, support, or dual). Moreover, it is not related to the operational system constructs or languages since the statements are

[8]Notice that we refer to foreign-key values, as we use them, but not to foreign-key constraints because they are not usually meaningful in views.

designed as system-generic structures. A specification step, exploiting the information coming from the operational system, will then be needed to generate system-specific statements. Furthermore, this approach is flexible because (as we will see shortly) it allows *annotations* on Datalog programs whenever conditions get more complex and in order to handle specific cases.

The provenance of field values

In this subsection we consider the problem of the data provenance of the individual fields. We discuss how the procedure finds for every value either a source field to derive the value from or a generation technique for it. Our procedure, for a given rule, collects information about the provenance of values by analyzing the Skolem functor used in the head of the rule.

If the Skolem functor has only one parameter and this parameter is the OID of another content field, then the value comes from the instance of the construct having that OID. In the example, this is what happens whenever a Lexical is copied using rule R_2 (with the functor *SK7(lexOID)* that copies the content construct Lexical from a unique source content). Similarly, if the Skolem functor has more than one parameter and only one of them refers to a field, then a source construct can be determined as well. For example, let us refer to a translation step in our repository, which implements the elimination of hierarchies by removing parent Abstracts and moving their attributes to child Abstracts. Such a step includes the following rule, shown here in simplified form:

```
R18     Lexical(
              OID: SK15(lexOID, childOID),
              ...
              AbstractOID: SK2(childOID)
        )
    <-
        Lexical(
              OID:lexOID,
              ...
              AbstractOID:parentOID
),
Generalization( OID: genOID,
      ParentAbstractOID: parentOID,
      ChildAbstractOID: childOID
),
Abstract (
      OID: childOID
),
Abstract (
      OID: parentOID
);
```

In the rule, the functor *SK15* has two parameters: lexOID, referring to a content (the attribute), and childOID, referring to a container (the child Abstract). Clearly, the value is derived from the attribute, in many cases just copied. Concretely, this could have been used to copy the attribute School into IT_ENG.

Instead, if more than two or none of the functor parameters refer to a content construct, the result value has to be retrieved in some other way. This is exactly what happens in steps A and B with rules R_4 (functor *SK3(genOID,childOID)*) and R_5 (*SK4(absOID)*) respectively. This case can be handled with the use of annotations, which specify where values come from. Here we present an informal description of this approach to give an intuition of the adopted strategy while technical details will be shown in Subsection 3.6. In rule R_4, functor *SK3* generates the OID for a reference field (AbstractAttribute) from the OID of a Generalization (a support construct) and from the OIDs of an Abstracts (container construct). Here an annotation is used to specify that the reference from the child table to the parent can be implemented by means of the tuple ID (TID)[9] used as value for the field. A reason for this choice is the fact that every instance of a child typed table is an instance of the parent table too. Then for each tuple of the child container there is a corresponding tuple in the parent one with a restricted set of attributes, but with the same TID. Therefore, the reference

[9]In OR systems, every typed table usually has a supplementary field, which we call TID, a system-managed identifier which can be used to base reference mechanisms on.

can be made by means of some manipulation of this TID. In detail, the rules of Step A in the running example lead to the following system-generic pseudo-SQL statements:

```
CREATE VIEW ENG_A ... AS
    SELECT ... SCHOOL,
         REF(ENG_OID) AS EMP_OID
    FROM ENG ;

CREATE VIEW IT_ENG_A ... AS
    SELECT ... SPECIALTY,
         REF(IT_ENG_OID) AS ENG_OID
    FROM IT_ENG ;
```

ENG participates in a Generalization with EMP, so the rule copies its attributes and adds the values for the field referencing the parent EMP by casting the tuple TID. A similar thing happens for IT_ENG, which participates in a Generalization with ENG.

Similarly, in rule R_5, the functor generates the OID for a Lexical from the OID of an Abstract therefore it conveys the fact that the value of the field corresponding to that Lexical derives from a container. Our strategy involves the transformation of the TID into a value for this field. This solution would guarantee the presence of a unique identifier.

Combining source constructs

On the basis of the discussion in the previous subsection, it turns out that, for each field in a view, we have either a provenance or a generation. Provenance can refer to different source constructs, in which case it is needed to correlate them. In database terms, a correlation intuitively corresponds to a join. However, in practice, this is not always necessary. If two fields can be accessed from the same container, then the join can be avoided. For instance, considering an object-relational schema, if a construct C has a reference to a construct D, then we can use that reference to derive the values c of C and d of D, without any join.

In our paradigm we associate join conditions to Datalog rules and Skolem functors whenever necessary. In fact we handle typed functors, in the sense that they generate OIDs for specific constructs given the OIDs of a fixed set of constructs.

Let us see an example with an application of this technique. Consider another way of eliminating generalizations: moving the child attributes into the parent and deleting the child; obviously the parent will preserve its original attributes as well. In a multilevel case, this means that only the "top ancestor" is maintained, and attributes of all the "descendants" are moved to it. This requires, as a preliminary step (handled by a recursive rule), to detect the top ancestor for each child. These pairs are maintained in an auxiliary table and the Lexicals are copied from a child to the corresponding ancestor by means of a rule that uses a Skolem functor *SK6(ancestorOID, childOID, lexOID)*. Conversely, Lexicals from the source ancestor would be copied to the target one by means of the simpler functor *SK7(lexOID)*. Functor *SK6* relates two Abstracts (containers) and generates a new OID for the Lexical whose OID is *lexOID*. Instead, *SK7* generates OIDs for Lexicals given the OID of another Lexical.

The adopted combination of content-generating functors {*SK6*, *SK7*} encodes the sourcing of data as follows: as we will clarify in Section 3.6, it is a left join on TIDs between the ancestor and the child; in such a way, all the instances of the ancestor that are also instances of the child, appear in the result view as a single tuple. Moreover, the left join guarantees the inclusion of the tuples that represent instances of the ancestor that do not belong to the child.

In the running example, we have a two level generalization and so Lexicals have to be copied (if they exist) from two different child tables, thus leading to two left joins:

```
CREATE VIEW EMP_A
   (...,LASTNAME, SCHOOL, SPECIALTY) AS
  SELECT ...EMP.LASTNAME,
    ENG.SCHOOL, IT_ENG.SPECIALTY
  FROM (EMP LEFT JOIN ENG ON
     (CAST (EMP.OID AS INTEGER) =
     CAST (ENG.OID AS INTEGER)))
   LEFT JOIN IT_ENG ON
    (CAST (EMP.OID AS INTEGER) =
    CAST (IT_ENG.OID AS INTEGER)) ;
```

Notice that, in this statement, the pattern bases joins on the sharing of TIDs that takes place between parent and child instances. Moreover, consider that it is not always necessary to perform a join operation. In fact, there are some ORDBMSs (like DB2) that allow to perform our translation by accessing only the top level table of the hierarchy. For example, our query in DB2 will be characterized by the use of the `OUTER` keyword in the `FROM` clause, which exposes all the columns of the parameter table and of its subtables:

```
 CREATE VIEW EMP_A
    (...,LASTNAME, SCHOOL, SPECIALTY) AS
   SELECT ...EMP.LASTNAME,
    EMP.SCHOOL, EMP.SPECIALTY
   FROM OUTER(EMP) ;
```

As mentioned before, there might be cases in which fields of different containers can be accessed by just referring to a single container by means of references. This is what happens in Step C where the values for the fields in the referring typed table can be derived from the key fields in the referred one (rule R_6).

The following statement is among the ones generated for Step C:

```
   CREATE VIEW EMP_C ... AS
      SELECT ... LASTNAME,
             OFFICE->OFFICE_OID AS OFFICE_OID
      FROM EMP_B ;
```

Indeed, EMP has references towards OFFICE (which does not appear in the statement) via the field Office and OFFICE_OID is the identifier for OFFICE added by rule R_5. Then, we need to copy OFFICE_OID values into a field of EMP according to the semantics of the rule. It is clear that there are two sources: EMP and OFFICE. However OFFICE_OID can be accessed via Office, therefore the join between the two containers is not needed.

In this way, joins are avoided when possible, by exploiting *dereferencing* (as in the example) when such a feature is supported by the operational system. Otherwise, when they are necessary, their treatment is globally encapsulated in Skolem functors that relate constructs in a strongly-typed fashion. In general, we can provide a different combination of Skolem functors for each needed join condition. The concept is that we exploit functor expressivity and strong typing to understand how to combine the containers of the different fields.

3.6 The view-generation algorithm

We now illustrate our algorithm for generating views at runtime from Datalog rules encoding schema-level translations. The procedure is shown in Figure 3.5 and includes tasks from the previous version of MIDST (Tasks 0 and 1) as well as new ones (all the others). Let us comment on them.

The algorithm takes as input the name of the source schema and the indication of the desired target model and of the target DBMS. The "import" subprocedure (Task 0 in the figure) is a function already in the previous version of MIDST, adopted here in order to build an internal representation of the source schema. It maps each construct of the source schema in terms of supermodel constructs. Then (Task 1) we use the target model parameter to invoke another existing MIDST function: findAutomaticTranslation. It produces a translation (for translating the source schema into the target model), which is composed of a sequence of elementary steps. Each step is, in turn, a set of Datalog rules. The rest of the procedure generates the views, on the basis of the Datalog rules in the translation steps. This is done in various tasks, with a process that finds general features first and then specializes them to the actual target context. Specifically, Task 2 produces language-independent views, and this is done in two subtasks: we first produce “view-generators” (Subtask 2.a), which depend only on the model at hand, and then instantiates them to (language-independent) views, which refer to the schema elements of interest (Subtask 2.b). Then, Task 3

```
      Procedure $translateAsView(source\_schema, target\_model, targetDBMS)$

      Input: a source schema, a target model and a target DBMS
      Output: SQL statements that perform the runtime translation
 0      schema := importFromTargetSystem(schema_name);
 1      translation := findAutomaticTranslation();
        for each translation_step in translation do
 2.a        view_generators := computeViewGenerators(translation_step);
            for each view_generator in view_generators do
 2.b            l_independent_views.add(
                    instantiateViewGenerator(view_generator, source_schema));
            for each l_independent_view in l_independent_views do
 3              pseudoSQL_statements.add(
                    computePseudoSQLstatement(l_independent_view));
            for each pseudoSQL_statement in pseudoSQL_statements do
 4              executable_statements.add(
                    computeExecutableStatement(pseudoSQL_statement, targetDBMS));
 5      return executable_statements
```

Figure 3.5: The view-generation algorithm

transforms these views into statements in pseudo-SQL.[10] Finally, Task 4 compiles executable statements in the specific language (e.g. SQL, SQL/XML, XQuery) of the target operational system.

We describe the technical details of the procedure in the next subsections, as follows: the generation of language-independent views (Task 2) in Subsection 3.6, their conversion to pseudo-SQL views (Task 3) in Subsection 3.6, and finally the compilation of the executable view-creation statements (Task 4) in Subsection 3.6.

Language-independent views

As we said, language-independent views are built in two subtasks. The first of them, which produces "view-generators" (Subtask 2.a), is performed by means of the algo-

[10]As we will clarify later, this is essentially a simplified version of SQL, which has the goal of generalizing in a declarative syntax the various languages of the commercial DBMS's.

```
    Procedure $computeViewGenerators(translation\_step)$

    Input: a set of Datalog rules of a translation step
    Output: the view-generators corresponding to the translation step
1.    containerRules := findContainerRules(translation_step);
2.    for each containerRule in containerRules do;
3.       contentRules := findContentRules(containerRule, translation_step);
4.       view_generator := createViewGenerator(containerRule, contentRules);
5.       view_generators.add(view_generator);
6.    return view_generators;
```

Figure 3.6: The algorithm for finding view-generators

rithm shown in Figure 3.6. Its input is an elementary translation step *T*, which is a set of Datalog rules. As we said in Section 3.5, our goal is to produce a view for each container construct in the head of rules in *T* with components (columns in relational terms) for each of its content constructs. The classification of constructs is part of our supermodel, and so it is immediate for our procedure to discover which schema elements have to become views and which components thereof. In fact, line 1 in the algorithm finds container-generating rules by means of a simple inspection. Then, the loop in lines 2-6 builds a view-generator for each container rule. The most delicate step is to associate components with views, that is, to establish, for each component, which is the view it belongs to. This is done by finding the content-generating rules associated with the container rule at hand (line 3) and then building a view generator for the container rule and the associated content rules (line 4).

Let us introduce a bit of notation. Given a translation *T*, we denote the set of content-generating and dual-generating rules in it as $Contents(T)$ and the set of container-generating rules as $Containers(T)$. Given *T* and a container-generating rule R in *T*, we denote as $content(R, T)$ the set of rules in $Contents(T)$ generating content (and dual) constructs for R.

So, line 3 in the algorithm computes the association between container and content rules: given a container rule in a translation step, it finds the corresponding content rules. This is is determined by analyzing the Skolem functors in the rules in *T*. In our context, each Skolem functor *SK* is associated with a given construct, the one which it generates OIDs for. Each functor always appears with the same arity and arguments, each one having a fixed type. The associated function is injective and function ranges are pairwise disjoint. For example, consider functor *SK5* of Section 3.4, used in rule R_6 (which eliminates the references). As it can be seen from the rule, and especially its head, *SK5* takes as input the OID of an AbstractAttribute and the OID of a Lexical and generates a unique OID for another Lexical:

$$SK5 : AbstractAttribute \times Lexical \rightarrow Lexical$$

The relationship between content and container constructs is determined by the OIDs. Container constructs have one main OID whose uniqueness is guaranteed by a *primary* Skolem functor (the one that generates the OID in the head). On the other hand, content constructs have more than one OID: one of them identifies the content itself while the others relate it to other constructs such as the container. This second category of OIDs is generated by a family of *secondary* Skolem functors. Our procedure includes in *content*(R, T) the content rules in *T* that involve, as secondary functor, the primary functor of the container rule R.

For example, the head of the rule R_1 (which copies Abstracts) has the form:

```
Abstract ( OID: SK2(oid),
           Name: name )
```

and it is apparent that it is only characterized by its OID, the one that identifies it. Conversely, a content construct has at least two functors (one for each characterizing OID). This is the case for example for Lexical as mentioned in the head of rule R_2 (repeated here for the sake of convenience):

```
R2      Lexical (
              OID: SK7(lexOID),
              Name: name,
              IsNullable: isN,
                    IsIdentifier: isI,
                    AbstractOID: SK2(absOID)
              )
        <-
              Lexical (
                    OID: lexOID,
                    Name: name,
                    IsIdentifier: isI,
                    IsNullable: isN,
                    AbstractOID: absOID
              ),
              Abstract(
                    OID:absOID
              );
```

Here, *SK7* is the primary functor, used to generate unique OIDs for instances of Lexical from OIDs of other Lexicals; *SK2* is a secondary one, used to connect each instance of Lexical (content) to the appropriate Abstract (container) by retrieving the OID of the target Abstract (abstractOID) from the one of the source (absOID).

Therefore, in our running example, if *T* is the translation of Step A, we have that $\mathit{Containers}(T) = \{\mathrm{R}_1\}$ and $\mathit{Contents}(T) = \{\mathrm{R}_2, \mathrm{R}_3, \mathrm{R}_4\}$ and $\mathit{content}(\mathrm{R}_1, T) = \{\mathrm{R}_2, \mathrm{R}_3, \mathrm{R}_4\}$. In fact, each of the rules R_2, R_3, R_4 has *SK2* (the primary functor of R_1) as a secondary functor.

This completes the discussion of line 3 of the algorithm in Figure 3.6. The rest of the algorithm is pretty easy. Line 4 is based on a definition, as follows. For each $R \in \mathit{Containers}(T)$ (that is, for each container generating rule) we define a *view-generator* as a pair $\mathit{VG} = (R, \mathit{content}(R, T))$, composed of the rule itself and of a set of rules, those that define contents for its container. Essentially, a view-generator tells which rules define containers (and so will lead to views in the target schema) and which are the rules that define the respective contents (and so will lead to fields of the corresponding views). Finally, line 5 just prepares the result to be returned by the algorithm.

In the example, our algorithm will determine, for Step A, the following view-generator: $VG_1 = (R_1, \{R_2, R_3, R_4\})$. Intuitively, this view-generator says that in the target schema we have container constructs as generated by rule R_1 (and so, Abstracts), each with content constructs generated by R_2, R_3, and R_4 (Lexicals and AbstractAttributes).

Let us now move to the actual construction of language-independent views, Subtask 2.b in the main algorithm in Figure 3.5. This does not require procedural details, and is based on some definitions.

Given a Datalog rule R, we define an *instantiated body* IB as a specific assignment of values for the constructs appearing in the body of R. It means that, for each construct in the body, we have values for name, properties, references, and OID that satisfy the predicates in the body of the rule itself with respect to the considered schema. For example, given the body of rule R_2 (copy-lexical), an instantiated version of it is the following one:

```
Lexical (
      OID: 100,
      Name: "lastName",
      IsIdentifier: "false",
      IsNullable: "false",
      AbstractOID: 3
),
Abstract(
      OID: 3
);
```

In the running example, it expresses the fact that we are copying the Lexical "lastName" (with OID 100) from the Abstract "EMP" (with OID 3). We remark that in general the conditions expressed in the bodies of Datalog rules (which are evaluated within MIDST-RT supermodel) may refer to container, content, and dual constructs as well as to support ones.

We define an *instantiated head* IH for a given instantiated body IB, as a construct whose name, properties, references, and OID are instantiated as a consequence of the instantiation of variables in *IB*. Again with reference to R_2, we have the following instantiated head:

```
Lexical ( OID: SK7(100),
          Name: "lastName",
          IsNullable: "false",
          IsIdentifier: "false",
          AbstractOID: SK2(3)
)
```

This head defines a new Lexical for a given Abstract (with OID obtained applying the functor *SK2* to the argument 3) that is a copy of the original Lexical of the Abstract with OID 3.

Finally, an *instantiated Datalog rule IR* is a pair (IH, IB) where *IH* is an instantiated head for the instantiated body *IB* of R.

Then, Subtask 2.b in the algorithm in Figure 3.5 computes a set of *language-independent views* for a view-generator *VG*, where each of them is defined as $V = (IR, \{c_1, c_2, \ldots\ c_n\})$, and is composed of an instantiation *IR* of rule R and of the set of all the possible instantiations of rules in *content*(R, T) that are coherent with *IR*.

In the example, the language-independent views for VG_1 are:[11]

$$\begin{aligned} V_1 = \ & (EMP \rightarrow_{\text{copy-abstract}} EMP\ , \\ & \{\ EMP(lastName) \rightarrow_{\text{copy-lexical}} EMP(lastName), \\ & \ EMP(office) \rightarrow_{\text{copy-abstractAttribute}} EMP(office)\}) \\ V_2 = \ & (OFFICE \rightarrow_{\text{copy-abstract}} OFFICE\ , \\ & \{\ OFFICE(offName) \rightarrow_{\text{copy-lexical}} OFFICE(offName), \\ & \ OFFICE(city) \rightarrow_{\text{copy-lexical}} OFFICE(city)\}) \\ V_3 = \ & (ENG \rightarrow_{\text{copy-abstract}} ENG\ , \\ & \{\ ENG(school) \rightarrow_{\text{copy-lexical}} ENG(school), \\ & \ Gen(EMP, ENG) \rightarrow_{\text{elim-gen}} ENG(EMP)\}) \end{aligned}$$

[11]The descriptive names of the rules are inserted for the sake of readability of the example. Notice that we have omitted the suffix _A as no ambiguity arises.

$$\begin{aligned} V_4 = \ & (\mathit{IT\_ENG} \rightarrow_{\text{copy-abstract}} \mathit{IT\_ENG}\,, \\ & \{\ \mathit{IT\_ENG}(\mathit{specialty}) \rightarrow_{\text{copy-lexical}} \mathit{IT\_ENG}(\mathit{specialty}), \\ & \ \mathit{Gen}(\mathit{ENG}, \mathit{IT\_ENG}) \rightarrow_{\text{elim-gen}} \mathit{IT\_ENG}(\mathit{ENG})\}) \end{aligned}$$

In plain words, this means that we will have to produce four views, each with the associated components. For example, V_1 says that there will be a view *EMP*, with columns *lastName* and *office*.

It is worth noting that in our tool language-independent views contain additional information besides the one shown above. In particular, a language-independent view is a map of actual values assigned to the variables of the rules (content- and container-generating) that belong to the view-generator. As a concrete example, the language-independent view V_4 is represented in our tool as:

```
CONTAINER: [oid=75; name = IT_ENG;
        internal_oid = IT_ENG_OID ]
CONTENTS: {
        [oid=332; name = SPECIALTY;
        absOID=75; isN = false; isId = false],
        [oid=6; parentOID=74; childOID=75;
        genOID=6; parentName = ENG ]
    };
```

where "CONTAINER" represents the instantiation of the container-generating rule that copies the Abstracts (in the example the typed table IT_ENG), while "CONTENTS" represent the useful instantiations of the content-generating rules that copy Lexicals and remove Generalizations.

Pseudo-SQL view creation statements

Let us now devote our attention to Task 3 of the procedure in Figure 3.5. It performs the translation of a language independent view into a pseudo-SQL view-creation statement and it follows the algorithm shown in Figure 3.7. Its input is a language-independent view, $V = (\mathit{IR}, \{c_1, c_2, \dots c_n\})$ instantiation of a view-generator

```
     Procedure $computePseudoSQLstatement(l\_independent\_view)$

     Input: a language-independent view
     Output: the pseudo-SQL view creation statement
1.     name := l_independent_view.getContainerName();
2.     columns := l_independent_view.getContentNames();
3.     sourceContainers := instantiated_container_rule.getSource();
4.     for each instantiated_content_rule in l_independent_view do
5.        targetList.add(instantiated_content_rule.calculateSource());
6.     fromClause.add(sourceContainers);
7.     for each instantiated_content_rule in l_independent_view do
8.        if instantiated_content_rule.getSourceContainer $\notin$ sourceContainers
          then
9.           fromClause.add(instantiated_container_rule.getJoinCondition());
10.    fromClause.simplify();
11.    pseudo-SQL_statement :=
        create pseudo-SQL_statement(name, columns, targetList, fromClause);
12.    return pseudo-SQL_statement;
```

Figure 3.7: The creation of pseudo-SQL view statement

VG $= (R, \mathit{content}(R, T))$, for a container rule R. The resulting pseudo-SQL statement has the following structure:

```
CREATE VIEW $name(col_1, col_2, \ldots, col_n)$ AS
    SELECT $a_1(s_{j_1}.col_1), a_2(s_{j_2}.col_2), \ldots, a_n(s_{j_n}.col_n)$
    FROM $sources$;
```

Line 1 of the algorithm in Figure 3.7 obtains *name* from V (the variable named *l_independent_view*) by retrieving the name of the head construct of the instantiation IR of the container-generating rule R. This is the name of the actual view to be created.

Next, line 2 derives the names for the columns of the view, col_1, col_2, ..., col_n, by getting the names of the constructs generated by the heads of the instantiated rules $\{c_1, c_2, \ldots\ c_n\}$ in V, and so each of them is a content (or dual) construct.

In line 3, the algorithm identifies, on the basis of the primary Skolem functor of the container rule R, the *main source containers* for the view: essentially, these are the containers (usually just one[12]) in the source schema that are transformed in to the view being constructed here. In the example, we will have that EMP_A is the source container for EMP_B and so on.

Then the algorithm proceeds by producing the details for the `SELECT` statement in the view:

(a) the identification of the source container (let us call it *source*$(s_{j_i}.col_i)$) for the provenance of each content element col_i and of the respective actual value for it (indicated with the functional symbols a_i); this is done in lines 4-5, discussed in detail in Subsection 3.6

(b) the actual construction of *sources* in the `FROM` clause, with a refinement and the merge of the various elements *source*$(s_{j_i}.col_i)$, with the possible use of suitable join conditions (lines 6-10, illustrated in Subsection 3.6).

At the end the procedure creates and returns the pseudo-SQL statement putting together the elements produced in the previous steps (lines 11-12).

Let us consider the aspects, (a) and (b) above, in turn.

Finding value provenance

Let us now discuss how the algorithm identifies, in lines 4-5, the sources of each content col_i. Specifically, this involves the decision on whether the value can be copied (if so, from where) or has to be generated (if so, how). This is done on the basis for the information given by the Skolem functors of the rules that generate col_i and the annotations possibly specified on them. Let us provide some detail. Given a content-generating rule R', its secondary functor links the generated content to its source container (the one the functor is applied to). The parameters of the functor are instantiated as a consequence of the instantiation of the body of R'. The primary functor conveys information about the provenance of data (that is, the content to derive the value from)

[12]In the sequel, in order to simplify the discussion, we will assume that there is only one main source container for each view. The more general case is intricate but straightforward.

for the content under examination. In general, the joint instantiation of both primary and secondary functors indicates where to retrieve the values from. Specifically, if the primary functor can link the head content to a source construct, then the secondary functor allows to determine the corresponding container construct. It may happen that it is not possible to associate the primary functor to a source content (and thus to a source container) uniquely. In such cases the strategy we follow relies on the possibility of using of *annotations*, fragments of pseudo-SQL code that can be associated with Datalog rules, and more precisely to functors in them. Specifically it is possible to associate the primary functor with a generation technique for the value. This is essential for the functors that have two or more content arguments (or no content arguments at all). For example in Rule R_4 we have the primary functor *SK3* that has no content argument. As we will shortly see, an annotation is needed here.

Then, our algorithm proceeds as follows.

(a.1) Default case: there is no annotation on the primary functor; this is possible when (i) the functor has exactly one parameter, a content, or (ii) it has more parameters, with at least a content one and at most a container one. In case (i) the column of the view comes from the container in the source indicated by the secondary functor. In case (ii) the column of the view comes from the container mentioned in the functor. In both cases, the algorithm finds a target list element for the SQL statement composed of the names (in the source schema) of the container and of the content element. The algorithm computes also the provenance for such an element (to be used in the subsequent steps to build the `FROM` clause): it is the container mentioned above; if it does not coincide with the main source container for the view, the provenance is defined as a join of the two containers (and possibly others) on the basis of repeated OIDs in the body of the rule.

(a.2) Annotation case: if the primary functor is annotated with a query fragment a, then a is applied in order to calculate the value. Notice that the query can be written referring to all the literals in the instantiated content-generating rule.

Usually, these queries are simple and involve a small number of parameters. The provenance is handled as in case (a.1), on the basis of the containers involved in the rule and in the annotation.

As an example of case (a.1), consider again the rule R_3 of Step A (which we partially show here again for convenience), which copies the AbstractAttributes:

```
R3     AbstractAttribute(
              OID: SK8(oid),
              Name: name,
              IsNullable: isN,
              AbstractToOID: SK2(absToOID),
              AbstractOID: SK2(absOID)
       )
       <- ...
```

This rule is not annotated and its functor *SK8* takes in input the OID of the AbstractAttribute. In this case, in the target list of the view we will have an element s.*col*, where s is the name of the Abstract and *col* the name of the AbstractAttribute. The provenance of such an element will be the Abstract s. In the actual example, we will have, in the construction of the view EMP_A, an element in the target list of the form (EMP.Office) and its provenance would be EMP.

On the other hand, as an example of case (a.2), consider the rule R_4 of Step A which replaces the generalizations between two typed tables by adding a specific reference field (AbstractAttribute) in the child table:

```
R4     AbstractAttribute (
              OID: SK3(genOID, childOID),
              Name: name,
              IsNullable: "false",
              AbstractOID: SK2(childOID),
              AbstractToOID: SK2(parentOID)
       )
   <-
       Generalization (
              OID: genOID,
              ParentAbstractOID: parentOID,
              ChildAbstractOID: childOID
```

```
         ),
         Abstract (
               OID: parentOID,
               Name: name
         ),
        Abstract (
               OID: childOID
         );
```

Here, *SK3*, the primary functor, takes in input the OID of the Generalization and the OID of an Abstract. In this case, the functor is annotated with:

```
SELECT INTERNAL_ID FROM ABSTRACT(parentOID)
```

This annotation specifies that the value of the reference (indeed AbstractAttributes represent references) must coincide with the OID of the Abstract that is the parent of the generalization. In this case, in the target list, we will have the parent Abstract (in the sense that, as allowed by most OR systems, we will use the system managed TID as a value). Such an Abstract will also be the provenance for the value. However, as the main container for the view to be generated is the child Abstract, the actual provenance is the join of the two Abstracts. In the actual example, in the elimination of the Generalization between ENG and EMP, we would have the element EMP in the target list for view ENG_A and its provenance would be the join between ENG and EMP.

A similar strategy should be followed to cope with rule R_5 of Step B. As we have seen, such a rule generates a key field for every typed table without an identifier: thus the problem of generating a unique value at data level arises. In the head of the rule, the primary functor *SK4* takes an Abstract as input parameter, and so there are no natural sources for the values. A possible annotation could be the following one:

```
SELECT INTERNAL_ID FROM ABSTRACT(absOID)
```

This implies the adoption of the values of internal tuple identifiers (INTERNAL_ID) as elements for the key of the typed table as explained at the end of Subsection 3.5.

Building the FROM clause

Let us now discuss how point (b) above is performed, that is, how sources for the various elements are constructed and combined.

The `FROM` clause is initialized (line 6 in Figure 3.7) with the source containers for the language-independent view at hand. Then, the instantiated content rules in the view are examined one at the time (lines 7-9) and if the source container is not a main source container, then the join condition (computed in line 5, as we said above) involving both containers (and additional ones of needed) is added to the `FROM` clause.

Let us show a result of the application of this step, both to illustrate it and to motivate the next one. In the first example in Subsection 3.5, the algorithm would generate three elements for the source clause, namely the main source container EMP, and the two left joins between EMP and ENG, and between EMP, ENG and IT_ENG.

Finally (line 10), the algorithm examines the elements in the `FROM` clause that has been initially generated, and performs simplifications by merging the various join conditions, on the basis of common containers and of subsumed expressions. In the example, the simple element EMP and the left join between EMP and ENG would be removed because they are subsumed by the double left join over EMP, ENG, and IT_ENG.

At the end (lines 11-12), the procedure creates the pseudo-SQL statement combining the information retrieved on the previous steps and returns the statement.

Executable view-creation statements

After a system-generic SQL statement has been generated for a Datalog translation, it is customized according to the specific language and structures of the operational database system in order to be finally applied.

With respect to a complex translation involving more than one phase, each system-generic SQL statement encoding an elementary step is translated in terms of a system-specific and executable one.

The following SQL statements exemplify the elimination of hierarchies (rule R_4) which takes place in step A with reference to IBM DB2. This DBMS adopts the concept of *typed view*, which is a view whose type has to be defined explicitly. This motivates the presence of the two initial statements defining the types EMP_A_t and ENG_A_t in the result schema. The statements below implement the strategy consisting in using the internal OID to make the child refer to its parent. It is apparent that a lot of DB2 technical details are introduced in this last phase (for example, the use of type constructors, the various cast functions and explicit scope modifiers).

```
CREATE TYPE EMP_A_t AS (lastName varchar(50))
NOT FINAL INSTANTIABLE
MODE DB2SQL WITH FUNCTION ACCESS REF USING
                                       INTEGER;

CREATE TYPE ENG_A_t AS (
     toEMP REF(EMP_A_t),
     school varchar(50))
...;

CREATE VIEW EMP_A of EMP_A_t MODE DB2SQL
         (REF is EMPOID USER GENERATED) AS
     SELECT EMP_A_t(INTEGER(EMPOID)), lastName
     FROM EMP;

CREATE VIEW ENG_A of ENG_A_t MODE DB2SQL
         (REF is ENGOID USER GENERATED,
          toEMP WITH OPTIONS SCOPE EMP_A) AS
     SELECT ENG_A_t(INTEGER(ENGOID)),
       EMP_A_t(INTEGER(EMPOID)), school
     FROM ENG;
```

The produced statements are finally sorted in order to take care of the dependencies between views, so that a view that refers to another one is created later.

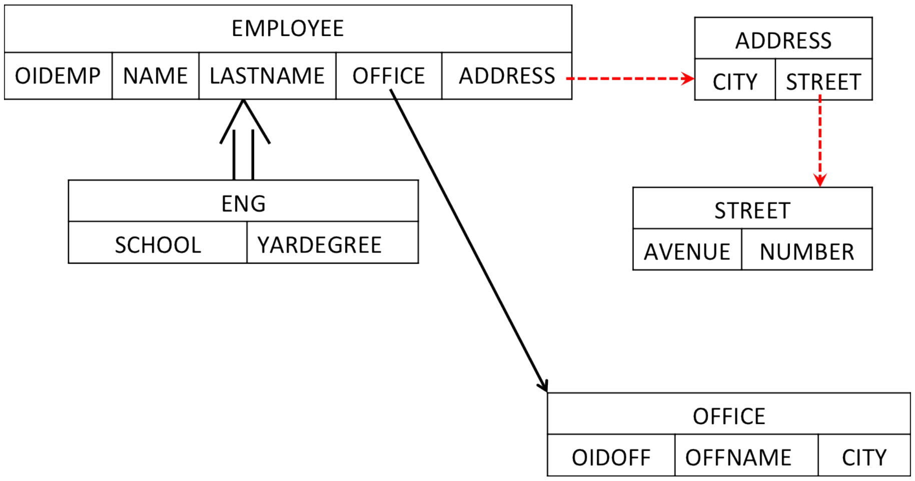


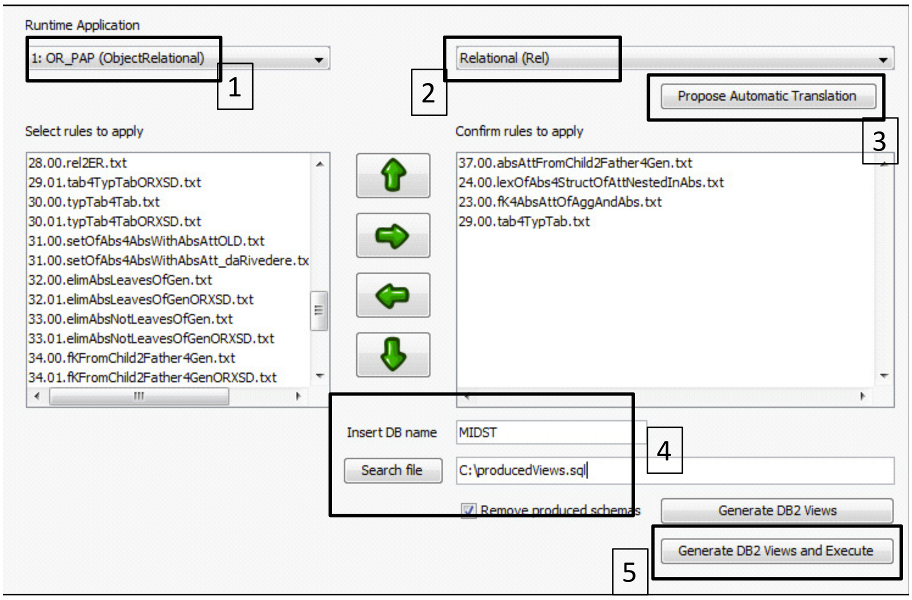
Figure 3.8: An object-relational schema: OR_DEMO

3.7 Views for the example scenarios

In this section we consider three scenarios of executable statements, in order to better understand the concept of "view" with respect of the motivating examples proposed at the beginning of this chapter.

Relational views

Let us consider the object-relational schema OR_DEMO shown in Figure 3.8, where we have three typed tables and two structured types. We have a generalization (ENG is a subtable of EMP) and a reference (from EMP to OFFICE). Structured types are used to build a two-level complex object (the value of Address comes from the Address type whose values involve the Street type). We want a translation that produces a set of relational views with reference to IBM DB2 [CRV00]. MIDST-RT completely supports this activity, with a component whose interface is shown in Figure 3.9. The user would perform the following sequence of steps, which are highlighted in the



1: OR_PAP (ObjectRelational)

Relational (Rel)

Select rules to apply

28.00.rel2ER.txt
29.01.tab4TypTabORXSD.txt
30.00.typTab4Tab.txt
30.01.typTab4TabORXSD.txt
31.00.setOfAbs4AbsWithAbsAttOLD.txt
31.00.setOfAbs4AbsWithAbsAtt_daRivedere.tx
32.00.elimAbsLeavesOfGen.txt
32.01.elimAbsLeavesOfGenORXSD.txt
33.00.elimAbsNotLeavesOfGen.txt
33.01.elimAbsNotLeavesOfGenORXSD.txt
34.00.fKFromChild2Father4Gen.txt
34.01.fKFromChild2Father4GenORXSD.txt

Confirm rules to apply

37.00.absAttFromChild2Father4Gen.txt
24.00.lexOfAbs4StructOfAttNestedInAbs.txt
23.00.fK4AbsAttOfAggAndAbs.txt
29.00.tab4TypTab.txt

Insert DB name: MIDST

C:\producedViews.sql

☑ Remove produced schemas

Figure 3.9: A screenshot of MIDST-RT

figure:

0. Import of the source schema from the operational system into the tool dictionary (this step is not shown in the figure).
1. Selection of the source schema (the one imported in step 0).
2. Selection of the target schema (relational).
3. Automatic selection of the programs to apply. The user can modify the set of selected programs in order to customize some steps of the translation.[13]
4. Insertion of useful information for the generation of the statements, such as the DB name and the path in which the statements will be produced.

[13]For example, the system proposes to remove generalizations merging the children into the parent, while the user wants to keep both the children and the parent.

5. Generation of the statements in a text file or direct execution over DB2.

Let us comment on the produced statements.[14] For brevity and without loss of generality, we describe only the first step of the translation (that is, the removal of generalizations). DB2 handles object views with the concept of typed view, which is a view whose type has to be defined explicitly. This motivates the presence of the "create type" statements defining the types OFFICE_t, EMP_t and ENG_t in the result schema. The statements below implement the strategy consisting in using the internal OID to make the child refer to its parent.

```
-- ****************************************************
-- STEP 1: removing generalizations
-- ****************************************************
CREATE TYPE OR_DEMO_1.OFFICE_t AS(
   CITY varchar(50),
   OFFNAME varchar(50))
MODE DB2SQL REF USING INTEGER;

CREATE VIEW OR_DEMO_1.OFFICE of
         OR_DEMO_1.OFFICE_t MODE DB2SQL
   (REF is OIDOFFICE USER GENERATED) AS
   SELECT
      OR_DEMO_1.OFFICE_t(
          CAST(OR_DEMO.OFFICE.OIDOFF AS INTEGER)),
      OR_DEMO.OFFICE.CITY,
      OR_DEMO.OFFICE.OFFNAME
   FROM OR_DEMO.OFFICE;

CREATE TYPE OR_DEMO_1.EMP_t AS(
   LASTNAME varchar(50),
   FIRSTNAME varchar(50),
   ADDRESS OR_DEMO.ADDRESS_t,
   OFFICE REF(OR_DEMO_1.OFFICE_t))
MODE DB2SQL REF USING INTEGER;

CREATE VIEW OR_DEMO_1.EMP of
         OR_DEMO_1.EMP_t MODE DB2SQL
   (REF is OIDEMP USER GENERATED,
    OFFICE WITH OPTIONS SCOPE OR_DEMO_1.OFFICE) AS
   SELECT
```

[14]Notice that, as we anticipated in Section 3.3, in the tool the names are distinguished by means of schema names, and so there is no need to use suffixes.

```
      OR_DEMO_1.EMP_t(
       CAST(OR_DEMO.EMP.OIDEMP AS INTEGER)),
      OR_DEMO.EMP.LASTNAME,
      OR_DEMO.EMP.FIRSTNAME,
      OR_DEMO.EMP.ADDRESS,
      OR_DEMO_1.OFFICE_t(
       CAST(OR_DEMO.EMP.OFF AS INTEGER))
   FROM OR_DEMO.EMP;

CREATE TYPE OR_DEMO_1.ENG_t AS(
   SCHOOL varchar(50),
   YEARDEGREE integer,
   to_EMP REF(OR_DEMO_1.EMP_t))
MODE DB2SQL REF USING INTEGER;

CREATE VIEW OR_DEMO_1.ENG of
         OR_DEMO_1.ENG_t MODE DB2SQL
   (REF is OIDENG USER GENERATED,
    to_EMP WITH OPTIONS SCOPE
       OR_DEMO_1.EMP) AS
   SELECT
      OR_DEMO_1.ENG_t(
       CAST(OR_DEMO.ENG.OIDEMP AS INTEGER)),
      OR_DEMO.ENG.SCHOOL,
      OR_DEMO.ENG.YEARDEGREE,
      OR_DEMO_1.EMP_t(
       CAST(OR_DEMO.ENG.OIDEMP AS INTEGER))
   FROM OR_DEMO.ENG;
```

The subsequent steps of the translation process will refer to the previous ones. This means that, after the removal of generalizations, we will have a set of views that represents a new schema without generalizations. We call this schema OR_DEMO_1. The next step is the elimination of nested types: we define a new set of views over the views previously defined. Thus, we will have a schema OR_DEMO_2 composed of a set of views defined over OR_DEMO_1. Then we must eliminate all the references (we introduce foreign-keys) and we must transform typed tables into simple tables. At the end, we have four new schemas (because the translation is composed of four steps), but only the last one, OR_DEMO_4, represents our target schema, a relational

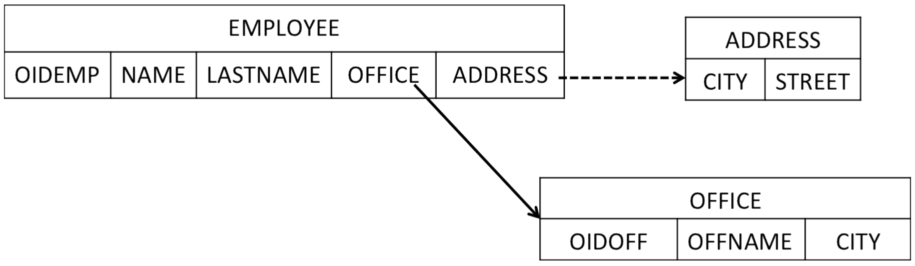


Figure 3.10: An object-relational schema

one.

XML views

Consider the object-relational schema shown in Figure 3.10 and suppose we need an XML document that contains all its data in a structured way.

We can do this with MIDST-RT by choosing XSD as the target model. In this way, the tool produces a statement that, executed over DB2, will create an XML document with all data directly extracted from the original schema. This can be possible by using an SQL/XML language, specific for the operational system, that contains functions that help the user to create XML elements from relational data. The tool produces the following statement:

```
SELECT XMLELEMENT(
  name "orxml",
    XMLCONCAT(
      XMLAGG(
        XMLELEMENT(
          name "emp",
          XMLELEMENT(name "OIDEMP",e.OIDEMP),
          XMLELEMENT(name "firstName",e.firstName),
          XMLELEMENT(name "lastName",e.lastName),
          XMLELEMENT(name "offref",e.off),
          XMLELEMENT(
            name "address",
            XMLELEMENT(name "city", e.address..city),
                 XMLELEMENT(name "street",e.address..street)
               )
             )
           ),
           (SELECT XMLAGG(
              XMLELEMENT(
                name "office",
                XMLELEMENT(name "OIDOFF",d.OIDOFF),
                XMLELEMENT(name "offName",d.offName),
                XMLELEMENT(name "city",d.city)
              )
             )
            FROM OR_XML.OFFICE d)
       )
   )
   FROM OR_XML.EMP e;
```

The produced XML document will be:

```
<orxml>
    <emp>
        <OIDEMP>1</OIDEMP>
        <firstName>Mark</firstName>
        <lastName>Brown</lastName>
        <offref>2</offref>
        <address>
            <city>Rome</city>
            <street>Viale Marconi 1</street>
        </address>
    </emp>
    ...
    <office>
        <OIDOFF>2</OIDOFF>
        <offName>ROMA TRE</offName>
        <city>Rome</city>
    </office>
      ...
</orxml>
```

```
public class EmployeeDao{
        private int oidEmp;
        private String name;
        private String lastName;
        private Demartment dept;
        private Address addr:

        public EmployeeDao(){…}

        //setters and getters

        public int getOID {
                return this.oidEmp;
        }
        …

        //CRUD operations
        public EmployeeDao doRetrieveByOID(int oid) {
                DataSource ds = new DataSource();
                Connection connection = null;
                …
                return employee;
        }

        public void save(EmployeeDao emp){
                …
        }
        private void doInsert(Connection conn
                                EmployeeDao emo, DataSource dc){
                …
        }

                …
}
```

Figure 3.11: The produced Java class

Object-oriented views

In this last scenario we start from an object-relational schema in order to obtain a set of Java classes that allows an object-oriented access to the database. This example briefly sketches how the generation process of a piece of Java code from relational tables may be performed.

The example we show produces some classes that contain CRUD methods (create, retrieve, update, delete) to access the database. Thus, we are following the DAO (data access object) design pattern. We are also able to produce classes by referring to other technologies, for example using Hibernate annotations. Moreover, thanks to an object-oriented importer, we can import the schema from the Java classes and produce an object-relational database: this is very simple, in fact, inside our metamodel, the object-oriented model is entirely contained into the object-relational one, so we do not need any translation.

This problem has a lot of solutions in the literature, but MIDST-RT ensures flexibility: in fact, thanks to the internal set of rules, the user can decide to modify the source schema to obtain the preferred translation, or can perform a translation towards a model that presents some non-standard features.

Starting from the object-relational schema shown in Subsection 3.7, one possibility is the creation of three Java classes using the DAO pattern. So we will have the objects Emp, Office and Address. Figure 3.11 shows the source code of the Java class EmpDAO.

3.8 Conclusions

In this chapter we have shown a runtime enhancement of MIDST which led to the design of MIDST-RT tool. The chapter showed as, moving from original framework of MIDST, where translation are applied within a supermodel that physically contains data, it was possible to comprise the theory in a more efficient one. Executable statements are generated for any possible translation and so the approach aims at being a step forward in runtime implementations of model management operators. However, for the proposed algorithms, more complex reasonings on view update would be necessary.

CHAPTER 4

NoSQL systems

Relational database systems (RDBMSs) dominate the market by providing an integrated set of services that refer to a variety of requirements, which mainly include support to transaction processing but also refer to analytical processing and decision support. From a technical perspective, all the major RDBMSs on the market show a similar architecture (based on the evolutions of the building blocks of the first systems developed in the Seventies) and do support SQL as a standard language (even if with dialects that differ somehow). They do provide reasonably general-purpose solutions that balance the various requirements in an often satisfactory way.

Basically, relational systems are well suited for transactional work, consisting in a great number of small, short-lived transactions; in this disguise, they are often called OLTP systems. Other scenarios of use see relational databases as decision support systems, where a batch workload is present consisting in read-only queries on a huge sample of data often involving time-consuming aggregations and calculations. Indeed, although both these scenarios are properly addressed by relational systems with a lot of offered features, database configuration, modeling and tuning diverge a lot. What is noticeable is that in most cases modeling techniques and patterns are driven by architectural, then non-functional, reasons. A major example is the common dimensional modeling. In data warehouses, the designer gives up to normalized fashion of relational data and adheres its project to safe patterns, star schemas, snowflakes and so forth, that are known to convey the best performance within a relational storage system when tested against analytical queries. Although there are also business reasons motivating dimensional modeling, being the simplicity of inspection for non technical users, the main driver is performance in a typical analytical scenario.

Some concerns have recently emerged towards RDBMSs. First, it has been argued that there are cases where their performances are not adequate, while dedicated engines, tailored for specific requirements (for example decision support or stream processing) behave much better [SMA+07] and provide scalability [SC11]. Second, the structure of the relational model, while being effective for many traditional applications, is considered to be too rigid or not useful in other cases, with arguments that call for *semistructured* data (in the same way as it was discussed since the first Web applications and the development of XML [ABS00]). At the same time, the full power of relational databases, with complex transactions and complex queries, is not needed in some contexts, where “simple operations” (reads and writes that involve small amount of data) are enough [SC11]. Also, in some cases *ACID* consistency, the complete form of consistency guaranteed by RDBMSs, is not essential, and can be sacrificed for the sake of efficiency. It is worth observing that many Internet application domains, for example, that of social networking, require both scalability (indeed, Web-size scalability) and flexibility in structure, while being satisfied with simple operations and weak forms of consistency.

With these motivations, a number of new systems, not following the RDBMS paradigm (neither in the interface nor in the implementation), have recently been developed. Their common features are scalability and support to simple operations only (and so, limited support to complex ones), with some flexibility in the structure of data. Most of them also relax consistency requirements. They are often indicated as *NoSQL* systems, because they can be accessed by APIs that offer much simpler operations than those that can be expressed in SQL. Probably, it would be more appropriate to call them *non-relational*, but we will stick to common usage and adopt the term

NoSQL.

There are fields where relational systems offer poor performance, even when equipped with an ad hoc design. Critical applications involve text indexing, web pages generation, code storage and object serialization, document search, scientific calculations.

Nowadays the market is particularly appreciating these segments, firmly linked to the cloud service provision. Cloud approaches are meant to provide efficiencies to information providers by means of scale economy in the procurement of hardware and IT commodities, while providing software and data as a service. In information management, the current orientation of cloud service tends to reach a cost/benefit balance, by limiting the provided features in force of a minor cost of administration. Of course, cloud services are contexts where relational systems do not scale well but, actually, even in case they scaled, they would require a higher total cost of ownership leading to a slower responsiveness to business changes.

Therefore, the community is assisting to the rise of many specialized information management systems, each addressing one or more non functional requirements. In many of them, relational DBMS' are not IT- and cost-efficient and so are replaced by specialized products.

There is a variety of systems in the NoSQL arena [Cat10, SC11], more than fifty, and each of them exposes a different interface (different model and different API). Indeed, as it has been recently pointed out, the lack of standard is a great concern for organizations interested in adopting any of these systems [Sto11b]: applications and data are not portable and skills and expertise acquired on a specific system are not reusable with another one. Also, most of the interfaces support a lower level than SQL, with record-at-a-time approaches, which appear to be a step back with respect to relational systems.

Globally, it is possible to characterize data management systems according to some categories.

Feature-first. Systems such as Oracle, SQL server, DB2, PostgreSQL, etc. These products focus on consolidated enterprise functional requirements; enforcement of transactional properties and structure prevail on other non functional requirements. They are well suited for the largest range of problems and, generally try to fit all situations. Feature-rich solutions are not necessarily divergent from cloud services; Amazon RDS is an example of generalist database system in a cloud environment. The core data model in this category of systems is, of course, relational.

Scale-first. Systems where scalability is paramount. They are born to provide an infrastructure to highly concurrent, distributed and million-users applications. Examples are platforms like Facebook, Google, Twitter, Yahoo, Amazon, etc.

The characterizing point in this category is that scalability is much more important than features. In fact, needed functionalities are almost elementary and do not imply complex representation and transaction models; on the other hand, the number of concurrent users can and will scale in a small amount of time. Quite obviously, systems like these cannot be based on a unique relational DBMS instance.

The most direct approach to scale-first requirements is sharding, also known as horizontal partitioning. It consists in splitting table rows in different instances and route the specific queries to the right database instance according to partitioning criteria. This architecture draws the maximum benefit from distribution, since queries and application workload is effectively partitioned, using physical resources effectively. Geographical distribution is facilitated by physical distribution of machines and so data is where they are needed, minimizing network traffic. Sharding architecture gives up to any effective possibility of intersecting partitioned data. In fact, inter-instance joins or comparisons are excluded by definition. Even more, replication and synchronization may be needed to some extent and must be then carefully handled; automatic software support is seldom present and hand coding is often required. Sharding approach is not easy to implement and its difficulties mainly lie in the individuation of candidates for distribution that, sometimes, might even be impossible. Substantially, a sharded database can be implemented as a collection of independent relational databases. Globally considered, they do not have the properties of a single relational system, in particular they lack consistency. CAP Brewer's theorem asserts that there are three systemic requirements, consistency, availability and network partition tolerance that are in a particular relationship when designing distributed systems. This relationships specifies that only two of them can be guaranteed at a time, one must be dropped. Sharded architectures drop consistency since involved databases are in fact separated and these architectures are often referred to as shared-nothing. Thus, a sharded system is tolerant to network partition, since it continues to operate even if the nodes cannot reach one another; such a system is available even when the network is not working, since no synchronization among nodes is needed. By contrast, it not consistent, since a transaction cannot involve data distributed in several nodes while guaranteeing ACID properties. Notice that if a DTP protocol (such as 2PC commit) had been fostered, it would have exposed the architecture to network partitioning intolerance.

A second approach to scale-first architectures is giving up to network partition tolerance adopting a DTP protocol. DB2 parallel or Oracle RAC follow this path and try to offer a scalable relational database system. This architecture is indicated when rich features offered by relational systems are so necessary to address requirements and, in some sense, lie on the same level of scalability. Actually, relational features are too difficult to scale and solutions like these are only suitable for small scenarios.

The third approach consists in using a scalable key-value store system. A key-value store is a mainstream in NoSQL systems. They support simple read and write operations on data items, identified by a key. All operations span a single data item and no joins or relationships are present. From the transactional perspective, these systems tend to adopt a weaker, eventual, consistency resulting in higher availability. A key element in any scalable key-value store is the partitioning algorithm. The store must be capable of scaling up incrementally and so different systems foster various algorithm to distribute data on nodes. Similarly, retrieval algorithm must be able to individuate the actual node where the needed value resides. Such algorithms may involve key sharding on every node or the presence of a coordinator node, routing the requests to the appropriate server. Famous examples of this class of solutions are Amazon Dynamo and Redis.

Simple structured storage. There are contexts where the storage features and data model of relational systems are not needed by applications. They may only need to store pieces of data with fast access and low operational effort or performance overhead. BerkeleyDB is a major example in this field, while SimpleDB is what cloud world offers. In particular, SimpleDB tries to offer the lowest maintenance cost with the simplest access primitives achieved with a query language that is a simplified form of SQL.

Document stores. They are commonly adopted when applications would not benefit from the complete spectrum of RDBMS' features, but would only use their storage capabilities to save BLOBs. Data models of document stores are very simple, and basically offer features to query documents in their native formats. Also, document stores support the application of highly optimized programming models such as map reduce directly on documents. MongoDB is an important example. It allows to store document encoded in BSON, a derivation of JSON format. CouchDB, another example of document store, directly support JSON.

Wide column and column families. In some sense they can be classified as a subset of *Simple structured storage* systems, however they are more focussed on scalability and on high performance queries. Database systems falling in this category, tend to reconcile the benefits of row and column physical handling of data. This leads to conceptual data models that mix physical aspects. For example, Apache HBase structures data in tables and columns, as for relational systems. However, it stores them in column fashion and groups columns in families. Whereas pieces of data are structured in tables and columns according to a conceptual design of the domain model, family partitioning respond to performance and scalability previsions. Unlike relational model, wide-column databases only offer simple primitives, get and put, to retrieve data from tables. Apache Cassandra is another example of column families NoSQL database system that strongly adopts HBase assumptions.

One of the most heard of system in this category is Hadoop. Hadoop is the Apache programming framework for distributed elaboration of large data sets. It is based on a revised implementation and theorization of Map Reduce algorithm by Google. Hadoop project comprises an elaboration engine and Hadoop file system, HDFS. Over this file system, the engine provides two components: JobTracker and TaskTracker that, working in strict cooperation with the file system, allow to implement a highly specialized instance of the map reduce algorithm. Indeed, Hadoop is neither a database nor a data management system, so it is probably not completely correct to characterize it among NoSQL systems. However NoSQL database systems can be based on it. This is not the case of HBase though it might seem. In fact, HBase can only be used as input or output of Hadoop algorithms, but its kernel is not based on map reduce strategies. The most significant example is Hive, a data warehousing and business intelligence engine built on top of Hadoop algorithms. Hive reimplements relational algebra primitives such as JOIN and SQL features like aggregation and sorting in a distributed and parallel fashion using map reduce algorithms in Hadoop framework. This would not formally prevent Hive from exposing a relational interface to the user and, in facts, Hive language is a direct derivation from SQL. Howerver Hadoop backend privileging scalability and performance forces Hive to give up to transactional features of relational systems; since its target and audience is the data warehousing world, where transactionality is somehow deemed, Hive is a very interesting example of a formally justified architectural style for analytical data management.

Purpose-optimized stores. In [SMA$^+$07] Stonebraker pointed out how relational DBMS' performance can be easily beaten by many market products when referring to a specific field. He underlines the fact that relational system are the best option to face general data management requirements, whereas for special purpose calculations, ad hoc products beat relational stores in benchmarks by several factors. Examples are Vertica for data warehouse analytical queries, VoltDB for transactional processing or StreamBase for processing stream of media.

CHAPTER 5

Towards a uniform programming interface for NoSQL databases

Non-relational databases (often termed as NoSQL) have recently emerged and have generated both interest and criticism. Interest because they address requirements that are very important in large-scale applications, criticism because of the comparison with well known relational achievements. One of the major problems often mentioned is the heterogeneity of the languages and interfaces they offer to developers and users. Different platforms and languages have been proposed, and applications developed for one system require significant effort to be migrated to another one. Here we propose SOS as a common programming interface to NoSQL systems (and also to relational ones), in order to support application development by hiding the specific details of the various systems. It is based on a metamodeling approach, in the sense that the specific interfaces of the individual systems are mapped to a common one. The tool provides interoperability as well, since a single application can interact with several systems at the same time.

5.1 SOS platform

The observations of Chapter 1 have motivated us to look for methods and tools that can alleviate the consequences of the heterogeneity of the interfaces offered by the various NoSQL systems and also can enable interoperability between them together with ease of development (by improving programmers' productivity, following one of the original goals of the relational databases [Cod82]). As a first step in this direction, we present here *SOS (Save Our Systems)*, a programming environment where non-relational databases can be uniformly defined, queried and accessed by an application program.

The programming model is based on a high-level description of the interfaces of non-relational systems by means of a generic and extensible meta-layer, based on principles that are inspired by those our group has used in the MIDST and MIDST-RT projects [ABBG09a, ACT^{+}08]. The focus in our previous work was on the structure of models and was mainly devoted to the definition of techniques to translate from a representation to another one. Here, the meta layer refers to the basic common structure and is then concerned with the methods that can be used to access the systems. The meta-layer represents a theoretical unifying structure, which is then instantiated (indeed, implemented) in the specific underlying systems; we have experimented with various systems and, in this work, we will discuss implementations for three of them with rather different features within the NoSQL family: namely, Redis,[1] MongoDB,[2] and HBase.[3]

Indeed, the implementations are transparent to the application, so that they can be replaced at any point in time (and so one NoSQL system can be replaced with another one), and this could really be important in the tumultuous world of Internet applications. Also, our platform allows for a single application to partition the data of interest over multiple NoSQL systems, and this can be important if the application has

[1]`http://redis.io`
[2]`http://www.mongodb.org`
[3]`http://hbase.apache.org`

contrasting requirements, satisfied in different ways by different systems. Indeed, we will show a simple application which involves different systems and can be developed in a rapid way by knowing only the methods in our interface and not the details of the various underlying systems.

To the best of our knowledge, the programming model we present is original, as there is no other system that provides a uniform interface to NoSQL systems. It is also a first step towards a seamless interoperability between systems in the family. Indeed, we are currently working at additional components that would allow code written for a given system to access other systems: this will be done by writing a layer to translate proprietary code to the SOS interface; then, the tool proposed here would allow for the execution on one system of code developed for another one.

5.2 The common interface

As we said, the goal of our approach is to offer a uniform interface that would allow access to data stored in different data management systems (mainly of the NoSQL family, but possibly also relational), without knowing in advance the specific one, and possibly using different systems within a single application. In this section we discuss on the desirable features of such an interface and then present our proposal for it. In the next section we will then describe the underlying architecture that allows for mapping it to the various systems.

NoSQL systems have been developed independently from one another, each with specific application objectives, but all with the general goal to offer rather simple operations, to be supported in a scalable way, with possibly massive throughput.[4] Indeed, there are many proposals for them, and effort have been devoted to the classification of them into various categories, the most important of which have been called key-value stores, document stores and extensible record stores, respectively [Cat10]. The three

[4]Our interest here in in the features related to how data is modeled and accessed. So, we will not refer further refer to the attention to scalability nor to another important issue, the frequent relaxation of consistency, which are both orthogonal to the aspects we discuss. Because of this orthogonality, our approach preserves the benefits of scalability and relaxation of consistency.

families share the idea of handling individual items ("objects" or "rows") and on the need for not being rigid on the structure of the items, while they differ on the features that allow to refer to internal components of the objects.[5] Most importantly, given the general goal of concentrating on simple operations, they are all based on simple operations for inserting and deleting the individual items, mainly one at the time, and retrieving them, one at the time or a set at the time. Therefore, it simple yet powerful common interface can be defined on very basic and general operations:

- **put** to insert objects
- **get** to retrieve objects
- **delete** to remove objects

Crucial issues in this interface are (i) the nature of the objects that can be handled, which in some systems are allowed to have a rather complex structure, not fixed in advance, but with components and some forms of nesting, and in others are much simpler and (ii) the expressivity of the get operation, which in some cases can only refer to identifiers of objects and others can be based on conditions on values. This second issue is related to the fact that operations can also be specified at high level with reference to a single field of a structured object. Our platform can perform this scenario inferring from the meta-layer all the involved structures and defining coherently the sequence of operations on them.

The simple interface we have just described is the core component of the architecture of the SOS system, which is organized in the following modules (Figure 5.1):

- the common interface that offers the primitives to interact with NoSQL stores
- the meta-layer that memorizes the form of the involved data

[5]We will give some relevant details for the three families and their representative systems in the next section, while discussing how the common interface can be implement in them, thus validating it.

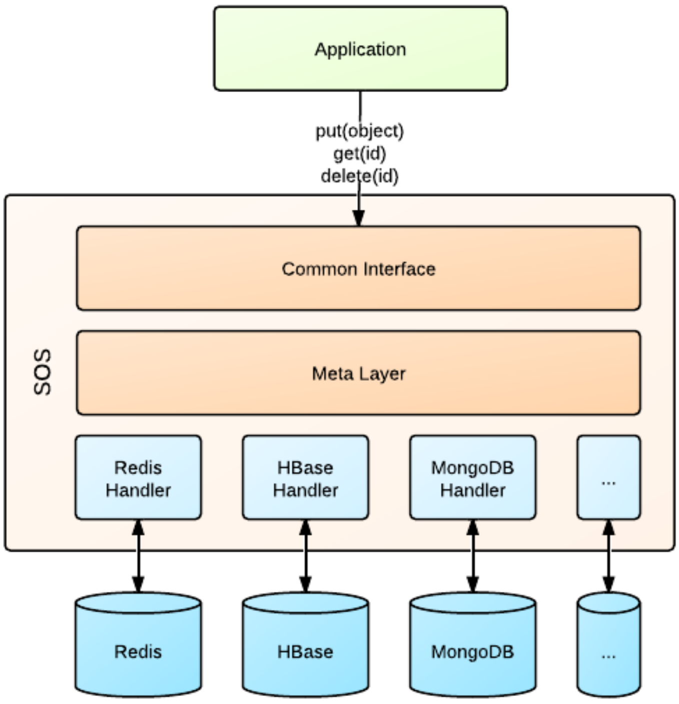


Figure 5.1: Architecture of SOS

- the specific handlers that generate the appropriate calls to the specific database system

The interface is the component that exposes methods to applications and that interact with the meta-layer allowing to store data and define high level operations with

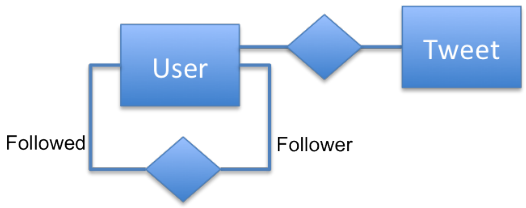


Figure 5.2: The schema for the data in the example

reference to general constructs. The meta-layer stores data and shows to the interface a uniform data model for performing operations on objects. The specific handlers support the low level interactions with specific NoSQL storage systems mapping meta-layer generic calls to system specific queries.

In order to show how MIDST-RT can support application development, in this chapter we will refer to an example regarding the definition of a simplified version of Twitter,[6] the popular social network application. We will adopt the perspective of a Web 2.0 development team that wants to benefit from the use of different NoSQL systems. Transactions are short-lived and involve little amount of data, so the adoption of NoSQL systems is meaningful. Also, let us assume that quantitative application needs have led the software architect to drive the decision towards the use of several NoSQL DBMSs, as it turned out that the various components of the application can benefit each from a different system.[7]

The data of interest for the example have a rather simple structure, shown in Figure 5.2: we have users, with login and some personal information, who write tweets; every user "follows" the tweets of a set of users and can, in turn, "be followed" by another set of users. In the example, in Section 5.5, we will show how this can be implemented by using three different NoSQL systems in one single application.

[6]`http://twitter.com/`

[7]For the sake of space here the example has to be simple, and so the choice of multiple systems is probably not justified. However, as the various systems have different performances and different behavior in terms of consistency, it is meaningful to have applications that are not satisfied with just one of them.

5.3 Meta-layer

In this Section we give some formal explanation of the meta-layer structure and we contextualize it with respect to the NoSQL system our platform handles.

Our approach leverages the genericity of the data model to allow for a standard development practice that is not bound to a specific DBMS API, but to a generic one.

Application programming interfaces are built over and in terms of the constructs of the meta-layer (without explicit reference to lower level constructs); in this way, programs are modular and independent of the particular data model; reuse is maximized.

According to the literature, a data model can be represented as the collection of its characterizing constructs, a set of constraints and a set of operations acting over them [TL82]. A construct is an entity that has a conceptual significance within the model. A construct can be imagined as the elementary outcome of a structural decomposition of a data model. A construct is relevant in a data model since it is a building block of its logical structures.

Thus, the aim of the meta-layer is to reconcile the relevant descriptive elements of mainstream NoSQL databases: key-value stores, document stores and record stores. In the following paragraphs, we will describe concisely their data models, and we will show how their constructs and operations can be effectively modeled through our meta-layer.

In key-value stores, data is organized in a map fashion: each value is identified by a unique key. Keys are used to insert, retrieve and remove single values, whereas operations spanning multiple ones are often not trivial or not supported at all. Values, associated to keys, can be whether simple elements such as Strings and Integers, or structured objects, depending on the expressive power of the DBMS considered. We chose Redis as a representative of key-value stores, being one of the richest in terms of data structures and operations. Redis supports various data types such as Set, List, Hash, String and Integer, and a collection of native operations to manipulate them.

Document stores handle collections of objects represented in structured formats such as XML or JSON. Each document is made of a (nested) set of fields and is associated to a unique identifier, for indexing and retrieving purposes. Generally, these DBMSs offer a richer query language than other NoSQL categories, being able to exploit the structuredness of the objects they store. Among document stores, MongoDB is one of the most adopted, offering a rich programming interface for manipulating both entire documents either single fields.

Extensible Record Stores offer a relaxation of the relational data model, allowing tables to have a dynamic number of columns, grouped in families. Column families are used for optimization and partitioning purposes. Within a table, each row is identified by a unique key: rows are usually stored in lexicographic order, which enables single accesses and sequential scans as well. HBase, being modeled after Google BigTable, belongs to this category and supports most of the features described above.

Moving from the data models described above, our meta-layer is designed for dealing with them effectively, allowing to manage collections of objects having an arbitrary nested structure. It turns out that this model can be founded on three main constructs: Struct, Set and Attribute.

Instances of each construct are given a name associated to a value. The structure of the value depends on the type of construct itself: Attributes contain simple values, such as Strings or Integers; Structs and Sets, otherwise, are complex elements whose values may contain both Attributes and Sets or Structs as well, as shown in Figure 5.3.

Each database instance is represented as a Set of collections. Each collection is a Set itself, containing an arbitrary number of objects. Each object is identified by a key, which is unique within the collection it belongs to.

As it turns out, specific non-relational models can be represented as a particular instance of the meta-layer, where generic constructs are used in well-defined combinations and, if necessary, renamed. Simple elements, such as key-values couples or single qualifiers can be modeled as Attributes. Groups of attributes, like HBase column families or Redis hashes, are represented by Structs. Finally, collections of

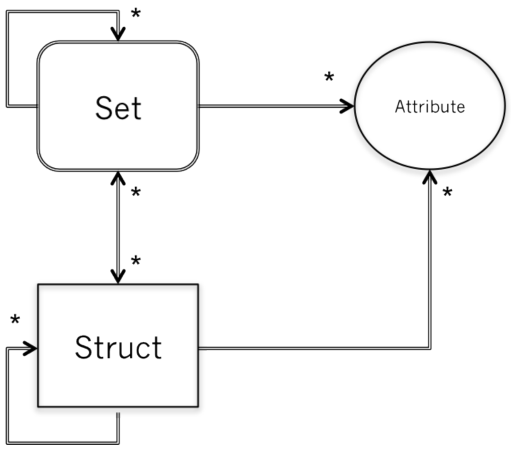


Figure 5.3: The metalayer

elements, as HBase tables or MongoDB collections are modeled by Sets. According to the specific structures the various NoSQL storage systems implement, from meta-layer constructs we will define a translation process to coherent system-specific NoSQL structures. In the remainder of this section it will be shown how this translation and data memorization works, moving from the model generic representation to the system-specific ones, with reference to the example introduced in Section 5.2. The meta-layer representation of the example is shown in Figure 5.4. A Struct Tweet represents a tweet sent by a User and a Set Tweets contains all the tweets of a User. Finally a simple Attribute is used for every non-structured information item of Users and Tweets (First Name, Last Name, ...).

MongoDB

MongoDB handles Collections of structured documents represented in BSON[8] and identified by a global key. In our translation, every SOS Collection is implemented as

[8]BSON: a binary encoding of the popular JSON format

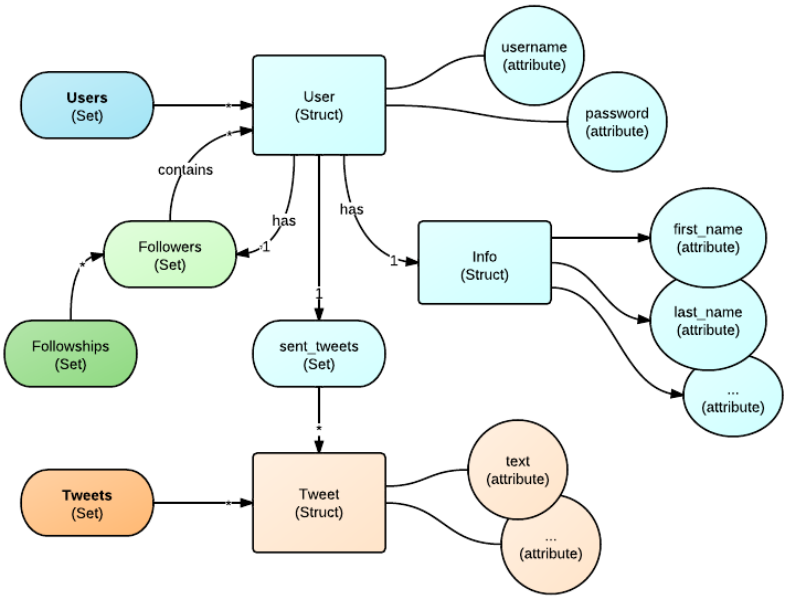


Figure 5.4: The meta-layer representation

a MongoDB Collection. Each object in the meta-layer is then represented as a single document within the collection itself. The structure of each object, implemented in JSON, fits closely the BSON format, allowing seamless translations between the two models.

HBase

In this context we model every SOS Collection as a Table. Objects within a Collection are implemented as records in the corresponding Table: the structure of each object establishes which HBase constructs are involved in the translation. Top-level Attributes are stored in a reserved column family named _top within a reserved qualifier named _value; top-level Sets generate a reserved column family (named _array[]) that groups

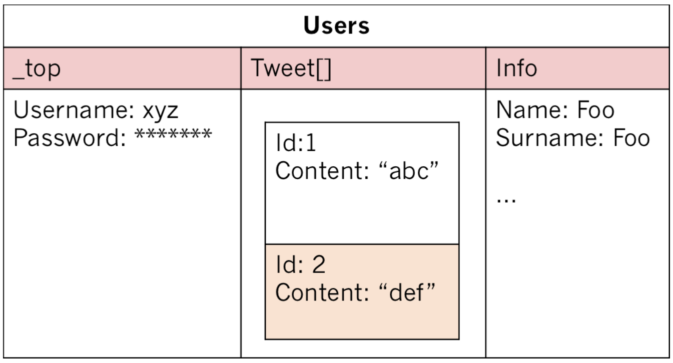


Figure 5.5: HBase Example

all the fields they further contain. Finally, each top-level Struct is represented as a single column family; deeper elements it contains are further stored in qualifiers. In order to store a nested tree in a flat structure (i.e. the qualifiers map) each field of the tree is given a unique qualifier key made of the whole path in the tree that goes from the root struct of the column family, to the field itself.

An example of translation is shown in Figure 5.5. The choice of storing top-level elements into different column families is driven by some data modeling considerations. In HBase, column families correspond to the first-class concepts each record is made of; in fact, data partitioning and query optimization are tuned on a per-column-family basis, considering each column family as an independent conceptual domain. In tree documents, we assume that top-level structs and sets correspond to the most significant concepts in the model, and therefore we represent each of them as a single column family.

Redis

Redis, among the three DBMSs we consider, is arguably the most flexible and rich in constructs: it is a key-value store where values can be complex elements such as

Hashes, Sets[9] or Lists.

For every object of the SOS Collections of the meta-layer, in Redis two R-Sets are defined. The first one is used to store keys, therefore it indexes the elements. The second one contains data: a Hash named _top with Attributes directly related to the Struct and a different Hash for every Struct or Set it contains. The name of those Hashes will be the concatenation of the name of the elements and the contained Struct.

The top Hash will contain all the first level attributes as following:

```
hash key:          user:10001
hash values:       [_top, info, twees[], ]

hash key:          user:10001:\_top
hash values:       username: foo
                   password: bar

hash key:          user:10001:info
hash values:       firstName: "abc"
                   lastName:  "def"

hash key:          user:10001:tweets[]
hash values:       [0].tweet.id = "1234",
                   [0].tweet.content = "....",
                   [1].tweet.id = "1235"
                   [1] ...
```

Assumptions we made for defining Redis translation are somehow close to HBase ones, discussed above. In fact, in our translation, Redis Hashes roughly correspond to HBase column families (each containing the qualifiers map). Nevertheless, object data in Redis is spread throughout many keys, whereas in HBase it is contained in a single record. As a consequence, we have to define in Redis support structures to keep track of the keys associated to each object, such as the R-set containing the hashes names.

[9]For the sake of readability we will refer to Redis set as R-Set.

5.4 The platform

For the definition of SOSplatform we featured a Java API which exposes the following methods corresponding to the basic operations illustrated in Section 5.2:

- `void put (String collection, String ID, Object o)`
- `void delete (String collection, String ID)`
- `Object get (String collection, String ID)`
- `Set<Object> get (Query q)`

The core class is `NonRelationalManager`. It supports `put`, `delete` and `get` operations. Primitives are based on object identifiers, however multiple retrievals are also possible by means of simple conjunctive queries (the second form of `get`). These methods handle arbitrary Java objects and are responsible for their serialization into the target NoSQL system. This process is based on the meta-layer, the data model pivoting the access to the systems. It is implemented in JSON, as there are many off-the-shelf libraries for Java object serialization into JSON. The implementation is based on the following mapping between the meta-layer and JSON format:

- Sets are implemented by Arrays
- Structs by Objects
- Attributes by Values

As the final step, each request is encoded in terms of native NoSQL DBMS operations, and the JSON object is given a suitable, structured representation, specific for the DBMS used. The requests and the interactions are handled by technology-specific implementations acting as adapters for the DBMS API.

We have implementations for this interface in the three systems we currently support. The classes that directly implement the interface are the "managers" for the various systems, which then delegate to other classes some of the technical, more elaborate operations.

For example, the following code is the implementation of the `NonRelational-Manager` interface for MongoDb. It can be noticed that `put` links a content to a resource identifiers, indeed creates a new resource. The adapter wraps the conversion to a technical format (this responsibility is delegated to `objectMapper`) which is finally persisted in MongoDB.

```
public class MongoDBNonRelationalManager
               implements NonRelationalManager{

 public void put(String collection, String ID, Object object){
      DBCollection coll = db.getCollection(collection);

      ByteArrayOutputStream baos =
                    new ByteArrayOutputStream();
      this.objectMapper.writeValue(baos, object);

      this.mongoMapper.persist(coll, getId(ID),
               new ByteArrayInputStream(baos.toByteArray()));
}
```

As a second implementation of `NonRelationalManger`, let us consider the one for Redis. As for MongoDb, it contains the specific mapping of Java objects into Redis manageable resources. In particular, Redis needs the concept of collection, defining a sort of hierarchy of resources, typical in resource-style architectures. It can be seen that the hierarchy is simply inferred from the ID coming from the uniform interface.

```
public class RedisNonRelationalManager
                implements NonRelationalManager {

public void put(String collection, String ID, Object object){
  Jedis jedis = pool.getResource();

  try {

     // the object is stored in the meta-layer
     ByteArrayOutputStream baos =
               new ByteArrayOutputStream();
```

```
        this.objectMapper.writeValue(baos, object);

        ByteArrayInputStream bais =
                   new ByteArrayInputStream(baos.toByteArray());
        this.databaseMapper.persist(jedis, collection, ID, bais);

        baos.close();
        bais.close();

        } catch(JsonParseException ex){
             ex.printStackTrace();

        } finally {
             pool.returnResource(jedis);
        }
}
```

5.5 Application example

In this Section we present the real implementation of the Twitter example mentioned in Section 5.2.

The application can be implemented by means of a small number of classes, one for users, with a method for registering news ones and for logging in, one for tweets with methods for sending them, and finally one for the "follower-followed" relationship, for updating it and for the support to listening. Each of the classes is implemented by using one or more database objects, which are instantiated according to the implementation that is desired for it (MongoDB for users, Redis for tweets, and HBase for the relationship). More precisely, the database objects are indeed handled by a support class that offers them to all the other classes.

As an example, let us see the code for the main method, `sendTweet()` for the class that handles tweets. We show the two database objects of interest, `tweetsDB` and `followshipsDB` of the `NonRelationalManager` with the respective constructors, used for the storage of the tweets and of the relationships, respectively. Then, the operations that involve the tweets are specified in a very simple way, in terms of `put` and `get` operations on the "DB" objects.

```
     NonRelationalManager tweetsDB =
           new RedisNonRelationalManager();
     NonRelationalManager followshipsDB =
           new HBaseNonRelationalManager();
  ...

  public void sendTweet(Tweet tweet) {

     // ADD TWEET TO THE SET OF ALL TWEETS
     tweetsDB.put("tweets", tweet.getId(), tweet);

     // ADD TWEET TO THE TWEETS SENT BY THE USER
     Set<Long> sentTweets =
       tweetsDB.get("sentTweets", tweet.getAuthor());
     sentTweets.add(tweet.getId());
     tweetsDB.put("sentTweets", tweet.getAuthor(), sentTweets);

     // NOTIFY FOLLOWERS
     Set<Long> followers =
        followshipsDB.get("followers", tweet.getAuthor());

     for(Long followerId : followers) {
        Set<Long> unreadTweets =
           tweetsDB.get("unreadTweets", followerId);
        unreadTweets.add(tweet.getId());
     }

     tweetsDB.put("unreadTweets", followerId, unreadTweets);
  }
```

It is worth noting that the above code refers to the specific systems only in the initialization of the objects `tweetsDB` and `followshipsDB`. Thus, it would possible to replace an underlying system with another by simply changing the constructor for these objects.

In a technical context, it is clear that an application such as the one described above, can be easily implemented from scratch, given the managers for the various systems. It is important to notice that systems built on this programming model address modularity, in the sense that the NoSQL infrastructure can be easily replaced without affecting the client code.

5.6 Conclusions

The contribute of this chapter is the introduction of a programming model based on the meta-layer that sustains homogeneity in treating non relational schemas. The original contribution of the work is the design of a meta-level supporting programming interfaces. Great attention in the study was also devoted to supporting the simultaneous use of multiple NoSQL system, a more and more common scenario in modern applications. We chose three systems as the most representative of specific classes of NoSQL implementations.

CHAPTER 6

Benchmarking of indexing strategies in SimpleDB

This chapter in a complementary fashion with respect to the previous ones, faces a new software paradigm from an architectural perspective, paying attention to various alternatives.

Cloud computing has been massively adopted recently in many applications for its elastic scaling and fault-tolerance. At the same time, given that the amount of available RDF data sources on the Web increases rapidly, there is a constant need for scalable RDF data management tools.

A novel architecture for the distributed management of RDF data is described in this chapter, exploiting the Amazon Web Service (AWS) existing commercial cloud infrastructure. We study the problem of indexing RDF data stored within AWS, by using the key-value store provided by AWS for small data items, namely SimpleDB. The goal of the index is to efficiently identify the RDF dataset(s) which may have answers for a given query, and route the query only to those. We devised and experimented with several indexing strategies; we discuss experimental results and avenues for future work.

6.1 Introduction

Cloud computing has been massively adopted recently in many applications for the scalability, fault-tolerance and elasticity features it provides. Cloud-based platforms free the application developer from the burden of administering the hardware and provide resilience to failures, as well as elastic scaling up and down of resources according to the demand. The recent development of such environments has a significant impact on the data management research community, in which the cloud provides a distributed, shared-nothing infrastructure for scalable data storage and processing. Many works have relied on cloud infrastructures focusing on different aspects such as implementing basic database primitives in cloud services [BFG+08] or algebraic extensions of the MapReduce paradigm [DG04] for efficient parallelized processing of queries [AEH+11].

Within the wider data management field, significant effort has been invested in techniques for the efficient management of Web data. A constantly increasing number of sources expose and/or share their data represented in the W3C's Resource Description Format (or RDF, in short) [KC04]. A well-known interesting RDF data source is the billion triples of DBPedia (`http://aws.amazon.com/datasets/2319`), others are catalogued in the Linked Open Data Web site (`http://linkeddata.org`) etc. RDF has also been used in highly dynamic news management scenarios, such as the BBC's reporting on the World Football Cup [KAKK10]. Efficient systems have been devised in order to handle large RDF volumes in a centralized setting, with RDF-3X [NW09, NW10] being among the best-known.

To exploit ever-increasing volumes of data in a cloud, works such as [HKKT10, LH11, MYL10, SZ10], either focus on MapReduce or use cloud-based key-value stores to store RDF data. These works mostly target at designing parallel techniques for efficiently handling massive amounts of data.

In this work, we explore an alternative and possibly complementary approach. We envision an architecture where large amounts of RDF data reside in an elastic cloud-based store, and focus on the task of efficiently routing queries to only those datasets that are likely to have matches for the query. Selective query routing reduces the total work associated to processing a query, and in a cloud environment, total work also translates in financial costs! To achieve this, whenever data is uploaded in the cloud store, we index it and store the index in an efficient (cloud-resident) store for small key-value pairs. Thus, we take advantage of: large-scale stores for the data itself; elastic computing capabilities to evaluate queries; and the fine-grained search capabilities of a fast key-value store, for efficient query routing.

Our implementation relies on the Amazon Web Services (AWS) cloud platform [AWS], one of the most prominent commercial cloud platforms today, which has been used in other data management research works [BFG$^+$08, SDQR10]. We store RDF files in Amazon Simple Storage Service (S3) and use Amazon SimpleDB for storing the index. Finally, RDF queries are evaluated against the RDF files retrieved from S3, within the Amazon Elastic Compute Cloud (EC2). While our results only hold within the Amazon platform, the architecture is quite generic and could be ported to other similar cloud-based environment.

6.2 Preliminaries

In this section, we briefly introduce the basics of our supported data model and query language as well as give a description of the Amazon cloud.

RDF and SPARQL

In order to lay the background and fix the terminology of the chapter, here we briefly summarize some features of the Resource Description Framework (RDF) data model [KC04, MM04]. RDF is a data model and formalism recommended by W3C and designed for the exchange and reuse of structured data among web applications. It is based on the concept of *resource* which is everything that can be referred to through a Uniform Resource Identifier (URI). RDF data model is built on resources, properties (a.k.a. predicates) and values. Properties can be seen as relations linking resources and values. The values can be either URIs, constants from primitive types called literals (such as string or integers), or blank nodes. Blank nodes are identifiers for unknown values. We use an underscore-prefixed notation to refer to them, as in `_:bnodeID`.

Information about resources is encoded using RDF triples, also called statements. A statement is a *triple* of the form $(subject, property, object)$, abbreviated as (s, p, o). The subject of a triple identifies the resource that the statement is about, the property identifies an attribute describing the subject, while the object gives the value of the property.

More specifically, let U, L and B denote three (pairwise disjoint) sets of URIs, literals, and blank nodes, respectively. A (well-formed) *triple* is a tuple (s, p, o) from $(U \cup B) \times U \times (U \cup L \cup B)$, where s is the subject, p is the property and o is the object of the triple.

A set of triples comprises a *graph*, which can be also called a *dataset*. Indeed, a set of triples encodes a graph structure in which every triple (s, p, o) describes a directed edge labelled with p from the node labelled with s to the node labelled with o. A graph can be identified by a URI value. In RDF, graphs can be built from other graphs through a *merge* operation. This is particularly useful for traversing various data sources with queries through the integration of these data sources.

The *merge* of RDF graphs is as follows. If these graphs have no blank nodes in common, then merging them results in their union. Otherwise, the graphs do share blank nodes and merging them amounts to renaming the (shared) blank nodes within the graphs with fresh identifiers, so that we fall into the previous case.

SPARQL [PS08] is the W3C standard for querying RDF graphs. In this approach, we consider the Basic Graph Pattern (BGP) queries of SPARQL, i.e., its conjunctive fragment allowing to express the core Select-Project-Join database queries. The normative syntax of BGP queries is

$$\texttt{SELECT } ?v_1 \ldots ?v_m \texttt{ FROM } uri_1 \ldots \texttt{FROM } uri_n \texttt{ WHERE } \{t_1, \ldots, t_o\}$$

with $\{t_1, \ldots, t_o\}$ an RDF graph whose triples can also use variables, $?v_1 \ldots ?v_m$ a set of variables occurring in $\{t_1, \ldots, t_o\}$ that defines the output of the query, and $uri_1, \ldots, uri_n$ the URIs of the graphs whose merging must be queried. Here, the notion of triple is actually generalized to that of *triple pattern* (s, p, o) from $(U \cup B \cup V) \times (U \cup V) \times (U \cup L \cup B \cup V)$, where V is a set of variables. Observe that repeating a variable in a SPARQL query is the way of expressing joins.

In the following, when a query has no `FROM` clause, we assume that it must be evaluated against the merge of all the graphs whose URIs are known.

Let us now turn to the semantics of a BGP query. First, a mapping μ from $B \cup V$ to $U \cup B \cup L$ is defined as a partial function $\mu : B \cup V \to U \cup B \cup L$. If o is a triple pattern or a set of variables, $\mu(o)$ denotes the result of replacing the blank nodes and variables in o according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Let $q = \texttt{SELECT } ?v_1 \ldots ?v_m \texttt{ FROM } uri_1 \ldots \texttt{FROM } uri_n \texttt{ WHERE } \{t_1, \ldots, t_o\}$ be a BGP query and D the graph obtained by merging the datasets whose URIs are $uri_1, \ldots, uri_n$. The evaluation of q is: $eval(q) = \{\mu(?v_1 \ldots ?v_m) \mid dom(\mu) = varbl(q) \textit{ and } \{\mu(t_1), \mu(t_2), ..., \mu(t_n)\} \subseteq D\}$, with $varbl(q)$ the set of variables and blank nodes occurring in q.

Notice that evaluation *treats blank nodes in a query as non-distinguished variables*. That is, one could consider without loss of generality queries without blank nodes.

Notation. From now on, to avoid writing long URIs, we use *namespaces*. Namespaces allow associating a short convenient prefix to the first part of a lengthy URI. Following the namespace usage, a URI can be replaced by the prefix to which is appended the last part of the URI. For example, the URI `http://xmlns.com/foaf/0.1/name` can be written `foaf:name`, provided that `foaf` has been declared as the namespace for `http://xmlns.com/foaf/0.1/`.

Running Example

Throughout the chapter we rely on a simple running example consisting of three datasets, representing: the articles published by Inria researchers, the books published

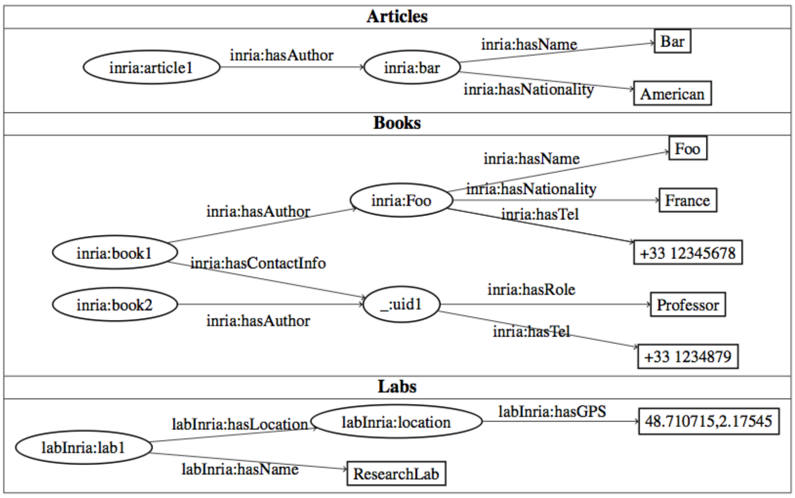


Figure 6.1: Graph representation of the example RDF data.

by Inria researchers, and the Inria labs. The content of these datasets is depicted in Figure 6.1.

The *articles* dataset describes the resource `inria:article1` whose author (`inria:hasAuthor`) is represented by the resource `inria:bar` whose name (`inria:hasName`) is “Bar” and whose nationality (`inria:hasNationality`) is “American”. The namespace `inria` is used to abbreviate the URI prefix `http://inria.fr/`.

The *books* dataset describes the resource `inria:book1` whose author (`inria:hasAuthor`) is `inria:Foo`, and for which there is an unknown (`_:uid1`) contact author (`inria:hasContact- Info`) whose role (`inria:hasRole`) is “Professor" and whose telephone number (`inria:hasTel`) is “+33 134879”. The resource `inria:Foo` has also a name (`inria:hasName`) which is “Foo", nationality (`inria:hasNationality`) “France” and telephone number (`inria:hasTel`) “+33 12345678”. The resource `inria:book2` is also described whose author

$$\begin{array}{lcl} db & = & dom+ \\ dom & = & (name, item+) \\ item & = & (key, attribute+) \\ attribute & = & (name, value) \end{array}$$

Figure 6.2: SimpleDB database layout.

(`inria:hasAuthor`) is the unknown author `_:uid1`.

Finally, the *labs* dataset describes the resource `labInria:lab1` whose name (`labInria:hasName`) is “ResearchLab” and whose location (`labInria:hasLocation`) described by the resource `http://labs.inria.fr/lab1/location` has the GPS coordinates (`labInria:hasGPS`) “48.710715,2.17545”. The namespace `labInria` is defined by the URI `http://labs.inria.fr /rdfExample/`.

Amazon cloud

Amazon Web Services (AWS) provides since 2006 a cloud-based services platform which organizations and individuals can take advantage of, in order to develop elastic scalable applications. For the purpose of this work, we mostly relied on the Amazon SimpleDB, a structured store for small atomic objects, and on Amazon's Simple Storage Service (S3).

Amazon SimpleDB

SimpleDB is a non relational data store provided by Amazon which focuses on high availability (ensured through replication), flexibility and scalability. SimpleDB supports a set of APIs to query and store items in the database. A SimpleDB data store is organized in *domains*. Each domain is a collection of *items* identified by their name. Each item contains one or more *attributes*; an attribute has a *name* and a set of associated *values*. Figure 6.2 outlines the structure of a SimpleDB database.

In the sequel, we can thus summarize the layout of data within SimpleDB as a four-level hierarchy $D|I|A|V$, where D is the domain name, I is the item name, A and V are attribute name and attribute value, respectively.

SimpleDB API. The main operations of SimpleDB API are the following:

- `ListDomains()` retrieves all the domains associated to one AWS account.
- `CreateDomain(D)` and `DeleteDomain(D)` creates a new domain `D` or deletes an existing one, respectively.
- `PutAttributes(D, k, (a,v)+)` inserts or replaces attributes `(a,v)+` into an item with name `k` of a domain `D`. If the item specified does not exist, SimpleDB will create a new item. `BatchPutAttributes` performs up to 25 `PutAttributes` operations in a single API call, which allows for obtaining a better throughput performance.
- `GetAttributes(D, k)` returns the set of attributes associated with item `k` in domain `D`.
- `select(expr)` operation queries a specified SimpleDB domain using query expressions similar to the standard SQL SELECT statements. We elaborate more about this API operation in the next section.

It is not possible to execute an API operation across different domains. Therefore, if required, the aggregation of results from API operations executed over different domains has to be done in the application layer. AWS ensures that operations over different domains run in parallel. Hence, it is beneficial to split the data in several domains in order to obtain maximum performance.

As most non-relational databases, SimpleDB does not adopt a strict transactional model based on locks or timestamps. It only provides the simple model of conditional puts. It is possible to update fields on the basis of the values of other fields. It allows for the implementation of elementary transactional models such as some entry level versions of optimistic concurrency control.

SimpleDB select statement. The `select` statement which can be used for querying SimpleDB is similar to the standard SQL select statements and has the following structure:

```
select (* | itemName() | count(*) | (attr1, ... attrN))
from domain_name
[where expression]
[sort_instructions]
[limit limit]
```

The expression can be any of the following:

```
(<simple comparison>)
(<select expression> intersection <select expression> )
(NOT <select expression>)
(<select expression>)
(<select expression> or <select expression>)
(<select expression> and <select expression>)
```

Comparison operators (=, ! =, >, ..., `like`, `in`, `is not null`, `is null`, etc.) are applied to a single attribute and are lexicographical in nature.

SimpleDB limitations. AWS imposes some size and cardinality limitations on SimpleDB. These limitations include:

- *D*omains number: the default settings of a AWS account allow for at most 250 domains. While it is possible to negotiate more, this has some overhead (one must discuss with a sale representative etc. - it is not as easy as reserving more resources through an online form).
- *D*omain size: the maximum size of a domain cannot exceed 10 GB.
- *I*tem name length: the name of an item should not occupy more than 1024 bytes.
- *N*umber of (attribute, value) pairs in an item: this cannot exceed 256. As a consequence, if an item has only one attribute, that attribute cannot have more than 256 associated values.
- *L*ength of an attribute name or value: this cannot exceed 1024 bytes.

In addition, when we execute a `select` query in a domain, there are also some limitations. Here we present only the ones related to our proposed architecture.

- The query cannot return more than 2500 items and the size of the result cannot exceed 1MB. If any of these conditions are not met, it is possible to retrieve the additional results by iterating the query execution using an identifier returned from the previous round.
- The maximum query execution time is 5 seconds, i.e., if a query takes longer than 5 seconds to be executed, it returns an error.

Amazon Simple Storage Service

Amazon S3 is a storage web service for raw data and hence, ideal for storing large objects or files. S3 stores the data in named buckets. Each object stored in a bucket has associated a unique name (key) within that bucket, metadata, an access control policy for AWS users and a version ID. The number of objects that can be stored within a bucket is unlimited.

If we want to retrieve an object from S3, we should access the bucket that contains it and request it by its name. S3 allows to access the metadata associated to an object without retrieving the complete entity. Unlike SimpleDB, there is no performance difference in S3 between storing objects in multiple buckets and storing them in just one.

S3 API. The S3 API includes the following basic operations:

- `ListBuckets()` returns the list of created buckets, `Create-Bucket (B)` creates a new bucket `B` and `DeleteBucket(B)` deletes an existing bucket.
- `PutObject(buck, key, obj, meta)` stores an object `obj` with name `key` and metadata `meta` within bucket `buck`.
- `GetObject(buck, key)` retrieves object `key` from a bucket `buck`.
- `GetObjectMetadata(buck, key)` retrieves only the object's metadata without fetching the actual object.

Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) is a virtual computing environment that allows the use of web service interfaces to launch virtual computer instances on which users' applications can be run. The virtual machine images are stored in the cloud and it is possible to configure them choosing hardware features such as RAM size, network access, etc. The utilization cost is calculated on the basis of the configuration of the machine, the running time of the application and the data transfer.

Amazon Simple Queue Service

Amazon Simple Queue Service (SQS) provides reliable and scalable queues that enable asynchronous message-based communication between the distributed components of an application. This service prevents an application from message loss and from requiring each component to be always available.

6.3 Architecture

Our envisioned architecture relies on the following services provided by Amazon: S3 for permanent storage, SimpleDB for storing structured information about the data stored in S3, EC2 for running our modules and SQS for the communication between the different modules.

RDF datasets are stored in S3 and each dataset is treated as a uninterpreted BLOB object. As explained in Section 6.2, it is necessary to associate a key to every resource stored in S3 in order to be able to retrieve it. For this reason, we assign to each dataset as a key the URI of the dataset. In general we indicate as $URI(ds_j)$ the URI associated to the dataset j. On the other hand, dataset indexes are instead kept in SimpleDB. In this way, we allow for fast retrieval of the URIs of the RDF datasets.

An overview of our system architecture is depicted in Figure 6.3. A user interaction with our system can be described as follows.

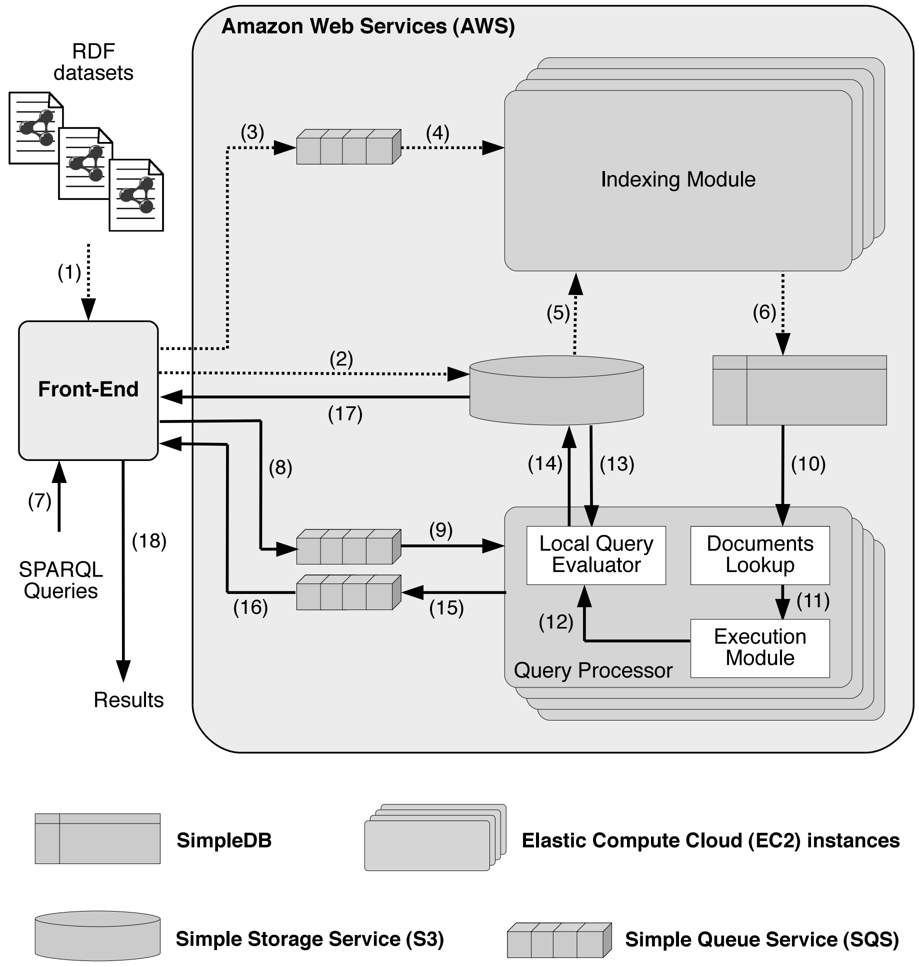


Figure 6.3: Proposed architecture.

The user submits to the *front-end* component RDF datasets (1) and the front-end module stores the file in S3 (2). Then, it creates a message containing the reference to the dataset and inserts it to the *loader request queue* (3). Any EC2 instance running our *indexing module* receives such a message (4) and retrieves the dataset from S3 (5).

The indexing module after transforming the dataset into a set of RDF triples, it creates the index data and inserts it in SimpleDB (6).

When a user submits a SPARQL query to the front-end (7), the front-end inserts the corresponding message into the *query request queue* (8). Any EC2 instance running our *query processor* receives such a message and parses the query (9). The query processor performs a lookup to the indexes kept within SimpleDB to find out the datasets that contain information to answer the query (10). Any processing required for merging or unioning the RDF datasets retrieved from SimpleDB is performed in the *execution module* (11). Then, the *local query evaluator* receives the final list of URIs pointing to the RDF datasets in S3 (12), retrieves them and evaluates the SPARQL query against these datasets (13). Then, it writes the results to S3 (14) and creates a message which is inserted into the *query response queue* (15). The front-end receives this message (16) and retrieves the results from S3 (17). Finally, the results are returned to the user (18).

6.4 Indexing strategies in SimpleDB

In this section we describe the RDF indexing strategies we have developed in SimpleDB that allow us to find out in a light way the RDF datasets that should be retrieved from S3 and then, queried to form the answer to the query. Since we use SimpleDB to store the indexes they should conform with data model of SimpleDB described in Section 6.2.

Notation. To simplify presentation, we will describe each indexing strategy in terms of the four levels of information that SimpleDB allows us to use, namely $(D|I|A|V)$ (see Section 6.2). To index RDF, we may use the values of subjects (S), properties (P) and objects (O) occurring in RDF triples, as well as the URIs (U) of the RDF datasets. Moreover, we will also use a set of three token strings, which we denote by $\underline{S}$, $\underline{P}$ and $\underline{O}$, and which we may insert in the index to specify whether some piece of data is to be treated as a subject, property, or object, respectively. In addition, we will use the symbol || to denote string concatenation. In cases where there is no confusion we may omit it (e.g., $\underline{SP}$ denotes the concatenation of the string values "subject" and "property"). Similarly, we will use a token string denoted by $\underline{D}$ to represent a constant domain name. As we will show, each indexing strategy can be represented by a concatenation of four |-separated symbols, specifying which information item is used in the domain name, item name, attribute name and attribute value, respectively.

Attribute-based strategy

In the following we describe our first indexing strategy of RDF datasets in SimpleDB and the query processing algorithm which utilizes the indices in order to retrieve datasets that are relevant to the SPARQL query. These datasets can be used then to form the answer to the query.

Indexing. According to this strategy, for each dataset three indexes are created: one for the subjects, one for the properties and one for the objects. Each index resides in a different SimpleDB domain. Then, for each dataset, an item named after the dataset is inserted in the respective domain. The name of the dataset is also the URI that allows us to access the RDF graph stored in S3. Using our notation we therefore have the following indexes:

1. $(\underline{S}|U|\underline{S}|S)$, which enumerates subjects using attribute-value pairs where the attribute name is the word "subject" and the value is the subject itself.
2. $(\underline{P}|U|\underline{P}|P)$, which enumerates properties using attribute-value pairs where the attribute name is the word "property" and the value is the property itself.
3. $(\underline{O}|U|\underline{O}|O)$, which enumerates objects using attribute-value pairs where the attribute name is the word "object" and the value is the object itself.

Handling SimpleDB limitations. As already discussed, SimpleDB can manage up to 256 attribute-value elements for each item. This means that a given dataset, confined in a single item by this strategy, can have up to 256 distinct subjects, 256 distinct properties and 256 distinct objects. While for properties this might not be a problem in many cases, for subjects and objects the limit is quickly reached. We have two dimensions of growth at our disposal to cope with this:

1. We may assign "partition" URIs $URI|1$, $URI|2$, …, $URI|k$ for a given real dataset URI URI. When e.g., the $\underline{S}$ domain overflows for the first time and a given dataset URI, we create the artifficial $URI|1$ and register the subsequent items as belonging to the (fictitious) dataset $URI|1$. (This also amounts to a virtual partitioning of the input dataset in several slices.) When $URI|1$ overflows, say, in the $\underline{S}$ or $\underline{O}$ index, we move to $URI|2$ and so on. To ensure complete index look-ups, a secondary index tracks all the partition URIs associated to a given URI.

2. We may use more domains. Let B denote the maximum number of SimpleDB domains available to us ($B = 250$ for a regular AWS account). We partition these domains according to their usage: some domains, denoted $\underline{S}_1$, $\underline{S}_2$, . . ., $\underline{S}_i$ will be used for the subjects, and similarly properties $\underline{P}_1$, $\underline{P}_2$, . . ., $\underline{P}_j$, respectively, $\underline{O}_1$, $\underline{O}_2$, . . ., $\underline{O}_l$ for the objects, with $i, j, l \geq 1$ and $3 \leq i + j + l \leq B$. For a given dataset, whenever we reach the 256 attribute-value limitation in an item in a domain $\underline{S}_i$ (or $\underline{P}_j$, or $\underline{O}_l$), we create a *new* item for this dataset in the "next" domain $\underline{S}_{i+1}$ (respectively, $\underline{P}_{j+1}$, or $\underline{O}_{l+1}$). The domain $\underline{S}_{i+1}$ is created the first time that the limits of the domain $\underline{S}_i$ are reached (and similarly for $\underline{O}$ and $\underline{P}$). Increasing the number of domains favors parallelism, since index entries for a given dataset, that are partitioned over several domains, can be simultaneously filled in, and consulted.

Our current implementation first, expands to several domains in order to maximize parallelism, and then, once the maximum number of domains has been taken, introduces partition URIs. We plan to further the analysis of the trade-offs between the two techniques in our future work.

subject domain i	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	($\underline{\text{S}}$, s'_{ds_1}), ($\underline{\text{S}}$, s''_{ds_1}), ...
$URI_k(ds_2)$	($\underline{\text{S}}$, s'_{ds_2}), ...
property domain j	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	($\underline{\text{P}}$, p'_{ds_1}), ($\underline{\text{P}}$, p''_{ds_1}), ...
$URI_k(ds_2)$	($\underline{\text{P}}$, p'_{ds_2}), ($\underline{\text{P}}$, p''_{ds_2}), ...
object domain l	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	($\underline{\text{O}}$, o'_{ds_1}), ($\underline{\text{O}}$, o''_{ds_1}), ...
$URI_k(ds_2)$	($\underline{\text{O}}$, o''_{ds_2}), ...

Table 6.1: Outline of the attribute-based strategy

Table 6.1 outlines data organization in SimpleDB using this strategy. The data shown in Figure 6.1 leads to the index shown in Table 6.2. For this small example, we use only three domains and one item per dataset in each domain.

Querying. For each constant (URI or literal) of a SPARQL query a SimpleDB `select` query is submitted to the $\underline{S}$ (or $\underline{P}$, or $\underline{O}$) domain(s), depending on the position of the constant in the query. Each such look-up retrieves the URIs of the dataset containing the respective subject (or object, or property) value. For each triple pattern, the results of all the `select` queries based on constants of that triple need to be intersected. The intersection leads to a set of URIs obtained out of a given triple pattern. The union of all URI sets thus obtained from the triples in a SPARQL query is the set of datasets on which the query must be evaluated.Using our running example assume that we want to evaluate the following SPARQL query:

Listing 6.1: Example SPARQL query

```
PREFIX inria: <http://inria.fr/>
SELECT ?s
WHERE {
    ?s inria:hasAuthor "Foo" .
    ?s inria:hasContactInfo ?o .
}
```

subject domain	
item key	(attr. name, attr. value)
articles	(S, inria:article1), (S, inria:bar)
books	(S, inria:book1), (S, inria:Foo), (S, _:uid1), (S, inria:book1)
labs	(S, labInria:lab1), (S, labInria:location)
property domain	
item key	(attr. name, attr. value)
articles	(P, inria:hasAuthor), (P, inria:hasName), (P, inria:hasNationality)
books	(P, inria:hasAuthor), (P, inria:hasContactInfo), (P, inria:hasRole)(P, inria:hasTel), (P, inria:hasNationality), (P, inria:hasRole)
labs	(P, labInria:hasLocation), (P, labInria:hasName), (P, labInria:hasGPS)
object domain	
item key	(attr. name, attr. value)
articles	(O, inria:bar), (O, "Bar"), (O, "American")
books	(O, inria:Foo), (O, "Foo"), (O, "+33 12345678"), (O, "France"), (O, "+33 1234879"), (O, "Professor")
labs	(O, labInria:location), (O, "ResearchLabs"), (O, "48.710715,2.17545")

Table 6.2: Attribute-based strategy applied to the running example

The corresponding SimpleDB queries that are required in order to retrieve the corresponding datasets is the following:

```
q1: select itemName()
    from property_domain
    where property = inria:hasAuthor;

q2: select itemName()
    from object_domain
    where object = "Foo";

q3: select itemName()
    from property_domain
    where property = inria:hasContactInfo;
```

The datasets retrieved from SimpleDB queries `q1` and `q2` will be intersected and the resulted datasets will be merged with the datasets retrieved from query `q3`. The query will be then evaluated on final set of the merged datasets.

Analytical cost model. In this section, we analyze the cost of the index strategy with respect to the size of the index as well as the number of required lookups while processing a SPARQL query. In both cases we present analytical costs for the worst case scenario.

Let n be the number of datasets stored in S3 and T the maximum size of a dataset in terms of the number of triples it consists. Hence, if T_{ds_i} is the size of dataset ds_i then $T = max(T_{ds_1}, T_{ds_2}, ..., T_{ds_n})$. We assume that the number of distinct subjects, properties and objects values appearing in a dataset is equal to the size of the dataset itself, and thus equals to the number of triples (worst case scenario). For each triple in a dataset we create three entries in SimpleDB. The size of the index for this strategy will be $3 \times n \times T$.

For the query processing, let q be the number of triple patterns of a BGP SPARQL query, then in the worst case scenario, the number of constants a query can have is at most $3 \times q$ (i.e., in case of a boolean query). Using this strategy, one lookup per constant in a query is performed to the appropriate domain type. For the case where the SimpleDB limit has not been reached and we thus have only one domain for the subjects, properties and objects, the number of lookups to SimpleDB is $3 \times q$.

In the case where we have reached the SimpleDB limitation and more than one domain for either of the subject, property, object domains are created, we need to perform one lookup to each domain separately. If d is the total number of allocated domains, the number of lookups to SimpleDB in the worst case scenario equals to $3 \times q \times d$.

Attribute-pair strategy

Indexing. This strategy uses three indexes, one for each pair of attributes in an RDF triple:

1. The first index is $(\underline{SP}|U|S|P)$, which enumerates the relation `(s, p)`, asserting whenever a subject has a property.
2. The second index is $(\underline{PO}|U|P|O)$, which enumerates the relation `(p, o)`, asserting whenever a property refers to an object.
3. Similarly, the third index is $(\underline{OS}|U|O|S)$, which enumerates the relation `(o, s)` asserting whenever an object is connected to a subject.

Handling SimpleDB limitations. SimpleDB attribute-value limitation leads to constraining a single dataset to 256 distinct subject-property pairs, 256 distinct property-object pairs and 256 distinct object-subject pairs. To overcome this limitation, we use the same technique as we described in the previous indexing strategy. We increase the number of each type of domain up to the number we have available. At the point where this number is reached we start filling the already existing domains by adding new items for the same datasets. The general organization of this indexing strategy in SimpleDB is depicted in Table 6.3. Using our running example of Figure 6.1, we have the index organization as shown in Table 6.4.

Querying. For each SPARQL triple pattern having at least one constant, we evaluate one SimpleDB `select` query on the corresponding domain(s) depending on the position of the constants in the triple. In case a triple pattern has one bound value we define a query where the corresponding attribute of the domain should not be null. In case a triple pattern has two bound values c_1 and c_2 we define a SimpleDB query whose `where` clause asks that c_1 equals to c_2. The query will be evaluated against the union of the datasets returned by the SimpleDB `select` queries corresponding to each triple.

subject-property domain i	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	(s'_{ds_1}, p'_{ds_1}), (s'_{ds_1}, p''_{ds_1}), (s''_{ds_1}, p'_{ds_1}) ...
$URI_k(ds_2)$	(s'_{ds_2}, p'_{ds_2}), (s'_{ds_2}, p''_{ds_2}), (s''_{ds_2}, p'_{ds_2}) ...
property-object domain j	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	(p'_{ds_1}, o'_{ds_1}), (p''_{ds_1}, o''_{ds_1}) , ...
$URI_k(ds_2)$	(p'_{ds_2}, o'_{ds_2}), ...
object-subject domain l	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	(o'_{ds_1}, s'_{ds_1}), (o''_{ds_1}, s'_{ds_1}), ...
$URI_k(ds_2)$	(o'_{ds_2}, s'_{ds_2}), ...

Table 6.3: Outline of the attribute-pair strategy

Using our running example and the example SPARQL query of Listing 6.1 we show the corresponding SimpleDB queries that are required in order to retrieve the corresponding datasets:

```
q1: select itemName()
    from property_object_domain
    where inria:hasAuthor = "Foo";

q2: select itemName()
    from property_object_domain
    where inria:hasContactInfo is not null;
```

The datasets retrieved from queries `q1` and `q2` will be merged and the query will be then evaluated against the merged datasets.

Analytical cost model. Similarly with the attribute-based index, for each triple of a certain dataset three entries are created in SimpleDB. In order to compute the size of this index, we assume that the number of distinct values of the subject-property pairs, property-object pairs and object-subject pairs appearing in a dataset equal to the the number of triples of the dataset (worst case scenario). Then, the size of the index for the attribute-pair strategy equals to $3 \times n \times T$, where n is the number of datasets and T the maximum number of triples of a dataset as introduced previously.

subject-property domain	
item key	(attr. name, attr. value)
articles	(inria:article1, inria:hasAuthor), (inria:bar, inria:hasName), (inria:bar, inria:hasNationality)
books	(inria:book1, inria:hasAuthor), (inria:book, inria:hasContactInfo), (inria:Foo, inria:hasName), (inria:Foo, inria:hasNationality), (inria:Foo, inria:hasTel), (_:uid1, inria:hasRole), (_:uid1, inria:hasTel) (inria:book2, inria:hasAuthor)
labs	(labInria:lab1, labInria:hasLocation), (labInria:lab1, labInria:hasName), (labInria:location, labInria:hasGPS)
property-object domain	
item key	(attr. name, attr. value)
articles	(inria:hasAuthor, inria:bar), (inria:hasName, “Bar”), (inria:hasNationality, “American”)
books	(inria:hasAuthor, inria:Foo), (inria:hasContactInfo, _:uid1), (inria:hasTel, “+33 12345678”), (inria:hasNationality, “France”), (inria:hasRole, “Professor”), (inria:hasAuthor, _:uid1)
labs	(labInria:hasLocation, labInria:location), (labInria:hasName, “ResearchLabs”), (labInria:hasGPS, “48.710715,2.17545”)
object-subject domain	
item key	(attr. name, attr. value)
articles	(inria:bar, inria:article), (“Bar”, inria:bar), (“American”, inria:bar)
books	(inria:Foo, inria:book), (_:uid1, inria:book), (“Foo”, inria:Foo), (“France”, inria:Foo), (“+33 12345678”, inria:Foo), (“Professor”, _:uid1), (“+33 1234879”, _:uid1), (_:uid1, inriafr:Foo)
labs	(labInria:location, labInria:lab1), (“ResearchLabs”, labInria:lab1), (“48.710715,2.17545”, labInria:location)

Table 6.4: Attribute-pair strategy applied to the running example

In the query processing procedure of this indexing strategy, at least one lookup is performed for each triple pattern of a SPARQL query. Certainly, this holds only in the case where the SimpleDB limit has not been reached. Then, the number of lookups to SimpleDB when using the attribute-pair strategy is equal to q, where q is the number of triple patterns of the SPARQL query.

In the case where we have reached the SimpleDB limitation and created more than one domain for either type of domain, we need to perform one lookup for each such domain. Then, the number of lookups to SimpleDB for the worst case scenario is $q \times d$, where q is the number of triple patterns of the SPARQL query and d is the total number of created domains in SimpleDB.

Attribute-subset strategy

Indexing. This strategy encodes each triple `(s, p, o)` by a set of seven patterns `(s)`, `(p)`, `(o)`, `(s, p)`, `(s, p, o)`, `(p, o)` and `(s, o)` corresponding to all non-empty attribute subsets. For each triple and each of these seven patterns, a new SimpleDB item is created and named after the pattern. As attribute name, we use the URI of the dataset containing this pattern; as attribute value, we use ϵ.

Using our notation, the indexes we create can be described as: $(\underline{D}|\underline{S}S|U|\epsilon)$, $(\underline{D}|\underline{P}P|U|\epsilon)$, $(\underline{D}|\underline{O}O|U|\epsilon)$, $(\underline{D}|\underline{SP}SP|U|\epsilon)$, $(\underline{D}|\underline{PO}PO|U|\epsilon)$, $(\underline{D}|\underline{SO}SO|U|\epsilon)$ and $(\underline{D}|\underline{SPO}SPO|U|\epsilon)$.

The general organization of this index is illustrated in Table 6.5. The data from our running example leads to the index configuration outlined in Table 6.6.

attribute-subset domain	
item key	(attr. name, attr. value)
$\underline{S}\|\|subject$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
$\underline{P}\|\|property$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{O}\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{SP}\|\|subject\|\|property$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),....
$\underline{PO}\|\|property\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{SO}\|\|subject\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{SPO}\|\|subject\|\|property\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...

Table 6.5: Attribute-subset indexing strategy

attribute-subset domain	
item key	(attr. name, attr. value)
S||inria:article1	(articles, ϵ)
S||inria:bar	(articles, ϵ)
S||inria:book1	(books, ϵ)
S||inria:book2	(books, ϵ)
S||inria:Foo	(books, ϵ)
S||inria:_:uid1	(books, ϵ)
S||labinria:lab1	(labs, ϵ)
S||labInria:location	(labs, ϵ)
P||inria:hasAuthor	(articles, ϵ), (books, ϵ)
P||inria:hasName	(articles, ϵ), (books, ϵ)
P||inria:hasNationality	(articles, ϵ), (books, ϵ)
P||inria:hasTel	(books, ϵ)
P||inria:hasRole	(books, ϵ)
P||inria:hasContactInfo	(books, ϵ)
P||labInria:hasName	(labs, ϵ)
P||labinria:hasLocation	(labs, ϵ)
P||inria:hasLocation	(labs, ϵ)
P||labInria:hasGPS	(labs, ϵ)
O||inria:bar	(articles, ϵ)
O||“Bar”	(articles, ϵ)
O||“American”	(articles, ϵ)
...	...
O||“48.710715,2.17545”	(labs, ϵ)
SP||inria:article1||inria:hasAuthor	(articles, ϵ)
SP||inria:bar||inria:hasName	(articles, ϵ)
SP||inria:bar||inria:hasNationality	(articles, ϵ)
...	...
SP||labInria:lab1||labInria:hasName	(labs, ϵ)
PO||inria:hasName||“Bar"	(articles, ϵ)
PO||inria:hasNationality||“American”	(articles, ϵ)
PO||inria:hasAuthor||inria:Bar	(articles,ϵ)
...	...
PO||labInria:hasGPS||“48.710715,2.17545”	(labs, ϵ)
SO||inria:article1||inria:Bar	(articles, ϵ)
SO||inria:bar||“Bar”	(articles, ϵ)
SO||inria:bar||“American”	(articles, ϵ)
...	...
SO||labInria:location||“48.710715,2.17545”	(labs, ϵ)
SPO||inria:bar||inria:hasName||“Bar”	(articles, ϵ)
...	...

Table 6.6: Attribute-subset indexing strategy in our example

Handling SimpleDB limitations. In this indexing strategy the limits of SimpleDB are exceeded when we have more than 256 datasets stored in S3 and all these datasets have one triple element value or a combination of them in common. Although this situation is not so often in various application scenarios, we cope with this by creating an extra domain each time the limitation is reached. In addition, when more than 10^9 distinct values of all triple elements combinations appear in the datasets stored in S3, the limit of the number of items allowed in a domain is surpassed. In this case we follow the same technique of adding a new domain.

Querying. This index cannot be queried directly using SimpleDB `select` statements, since one cannot use them to search and retrieve data according to an item key. For this reason, we exploit this index through the `GetAttributes(D, k)` SimpleDB API call, where D is the domain name and k is the item name. This call returns the set of attributes associated with that item.

For each triple pattern of a SPARQL query the corresponding `GetAttributes` call is generated, giving as item name a concatenation of the bound values of the triple pattern. The URIs obtained through all the `GetAttributes` calls resulting from each triple pattern are those of the datasets on which the query must be evaluated.

For example, for the SPARQL query of Listing 6.1 we need to perform the following SimpleDB API calls:

```
GetAttributes(attribute-subset, PO||inria:hasAuthor||"Foo")
GetAttributes(attribute-subset, P||inria:hasContactInfo)
```

If more than one domains have been created due to the limitation of SimpleDB, then we execute the `GetAttributes` to every domain. As in the previous cases, we then evaluate the SPARQL query to the the merge of the retrieved datasets.

Analytical cost model. Since for each triple of a dataset we create seven entries in SimpleDB, the size of the index of this strategy, let it be I_3, is $7 \times n \times T$, where n is the number of datasets and T is the maximum number of triples in a dataset.

For the query processing, we perform one lookup for each triple pattern appearing in the SPARQL query in the case where the SimpleDB limit has not been reached. In this case the number of lookups to SimpleDB when using the attribute-subset strategy is equal to q.

In the case where we have reached the SimpleDB limitation, we create more than one domain. Let d be the total number of domains in SimpleDB. Then, the number of lookups to SimpleDB equals to $q \times d$.

Domain-per-dataset strategy

Indexing. According to this strategy, a SimpleDB domain is allocated for and named after each dataset with URI URI_{ds_i}. We use the subject, property, object values of each triple in the dataset as the item names. Within our notations, for each dataset U we create the following indexes:

1. $(U|S|\underline{P}P|O)$
2. $(U|P|\underline{O}O|S)$
3. $(U|O|\underline{S}S|P)$

The organization of this index is illustrated in Table 6.7 while Table 6.8 shows a the organization of the index for a specific dataset of our example.

URI_{ds_i}	
item key	(attr. name, attr. value)
subject	($\underline{\text{P}}$||property, object)
property	($\underline{\text{O}}$||object, subject)
object	($\underline{\text{S}}$||subject, property)

Table 6.7: Domain-per-dataset indexing strategy

Handling SimpleDB limitations. SimpleDB limitations leads to constraining a single dataset to 256 property-object value pairs for each distinct subject value, 256 object-subject value pairs for each property value and 256 subject-property value pairs

articles	
item key	(attr. name, attr. value)
inria:article1	(P||inria:hasAuthor, inria:bar)
inria:hasAuthor,	(O||inria:bar, inria:article1)
inria:bar	(S||inria:article1, inria:hasAuthor)
inria:bar	(P||inria:hasName, “Bar”),
	(P||inria:hasNationality, “American”)
inria:hasName	(O||“Bar”, inria:bar)
inria:hasNationality	(O||“American”, inria:bar)
“American”	(S||inria:bar, inria:hasNationality)
“Bar”	(S||inria:bar, inria:hasName)

Table 6.8: Domain-per-dataset index example

for each distinct object value. This means that each subject, property, object value can appear 256 times inside a dataset. In addition, each dataset is constrained to having 10^9 distinct triple element values. In case any of the above situations occur for a dataset we add a new domain for this dataset.

Querying. For each triple pattern appearing in a given SPARQL query, a `select` SimpleDB query is defined and submitted to each existing domain. The resulting URI sets (one URI set for each triple pattern) are unioned and the query will be evaluated on the union of all such sets.

For instance, for the SPARQL query of Listing 6.1, we define the following SimpleDB queries for each domain i:

```
q1: select *
    from domain_i
    where property||inria:hasAuthor = "Foo";

q2: select *
    from domain_i
    where property||inria:hasContactInfo is not null;
```

The datasets retrieved from queries `q1` and `q2` then merged and the SPARQL query is evaluated against the merged result.

Analytical cost model. In this indexing strategy we create three entries to SimpleDB for each triple of a dataset. Therefore, the size of the index of this strategy is $3 \times n \times T$, where n is the total number of datasets stored in S3 and T is the maximum size of a dataset.

For the query processing part of this strategy, we perform one lookup to each domain for each triple pattern appearing in the SPARQL query (if the SimpleDB limit has not been reached). If q is the number of triple patterns of a SPARQL query, the number of lookups to SimpleDB when using the domain-per-dataset strategy is $q \times n$.

In the case where we have reached the SimpleDB limitation, we create more domains per dataset. If d is the total number of domains in SimpleDB, then the number of lookups to SimpleDB is $q \times d$.

6.5 Experiments

We have fully implemented our RDF data management architecture using all indexing strategies previously described. In this section, we describe a preliminary set of experiments conducted by deploying our system in the Amazon Web Services (AWS) environment.

Implementation and set up

We have used Java 1.6 to implement all the modules described in Section 6.3. The EC2 instance where we run our indexing module and query processor was part of the Ireland AWS facility and consisted of a 64-bit machine with 7.5 GB of memory, 2 virtual core with 4 EC2 Compute Units. An EC2 Compute Unit is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. For the local RDF query evaluation we used the query processor ARQ 2.8.8 with Jena 2.6.4.

We used synthetic RDF data generated by the SP^{2}Bench generator [SHLP09] which produces data based on the DBLP bibliography schema. We created datasets from 10.000 to 100.000 triples and used a set of queries which are SPARQL BGP

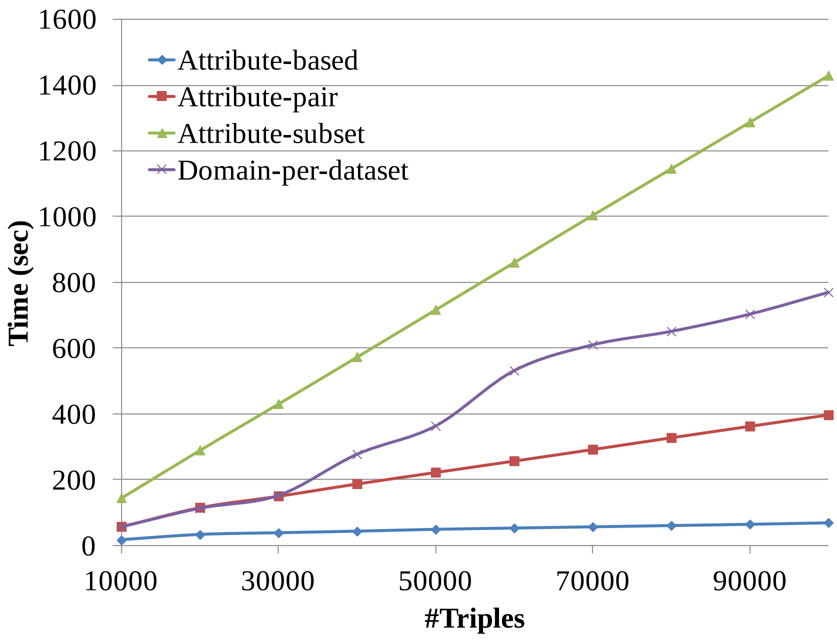


Figure 6.4: Indexing time for our four strategies.

queries obtained by some simplification of the SP^{2}Bench queries. For instance, we limited the queries to their BGP part, or modified them so that they would all have non-empty results when evaluated directly on the data[1]. The queries we used had from 1 to 8 triple patterns.

Indexing

In this section we study the performance of our four RDF indexing strategies, by measuring the performance of inserting index entries into SimpleDB. In this set of

[1]The full semantics of a SPARQL query on an RDF database should contain answers both from the explicit triples, and the implicit ones which are derived from the explicit triples using various RDF inference rules [PS08]. In the work described here, we have not yet considered cloud-based reasoning; this is part of our future work.

experiments we used 10 datasets of 10.000 triples each (i.e., 100.000 triples in total). We had 120 SimpleDB domains available to us for the experiments described here. Therefore, for the attribute-based and attribute-pair indexes the maximum number of domains that can be allocated for each domain type (i.e., S/P/O for attribute-based or SP/PO/OS for attribute-pair) was set to 30. After indexing all 10 datasets, the numbers of domains allocated were as follows:

- For the attribute-based strategy: 9 S domains, 1 P domain and 22 O domains
- For the attribute-pair strategy: 30 SP domains, 22 PO domains and 30 OS domains
- For the attribute-subset index: 1 domain;
- For the domain-per-dataset index: 10 domains, one for each dataset.

Because the SimpleDB limitations for the first two strategies are very restrictive even for small datasets, the domains were partitioned throughout our indexing experiments (starting from the smallest dataset of 10.000 triples).

For each indexing strategy we measure the time from the moment we start indexing the data, until the moment the index has been completely built in SimpleDB. Figure 6.4 shows the time required to build each index as the number of stored RDF triples increases. Note that we have used the `BatchPutAttributes` operation provided by SimpleDB which inserts to a single domain 25 items at a time, in a transactional fashion. We observe from the graph that the time required for the index construction grows linearly with the number of triples stored. As already shown by the analytical cost model of our indexing strategies, the attribute-subset index is the most time-consuming one since for each triple it inserts seven items into SimpleDB. On the other hand, the attribute-based index which defines only one attribute name for each item is more efficient. The attribute-pair strategy uses more domains and creates more unique attribute-value pairs that should be inserted in SimpleDB, and thus, is more expensive than the simple attribute-based approach. Finally, the domain-per-dataset

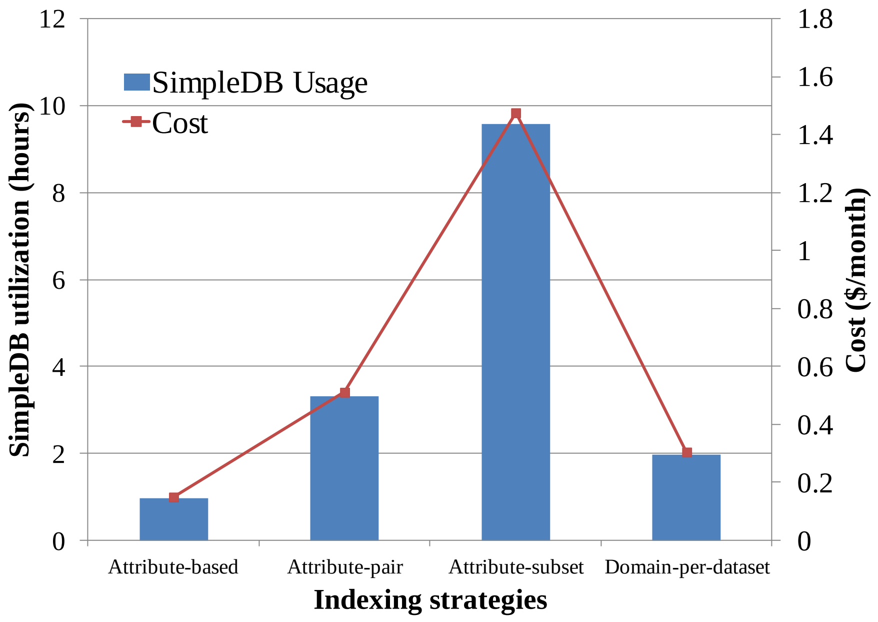


Figure 6.5: SimpleDB utilization and cost.

index inserts each time data to a specific domain and does not scale well as the number of datasets stored in S3 increases. While it performs similar to the attribute-pair index up to 3 datasets, the time then increases rapidly for more datasets.

Figure 6.5 shows the total machine utilization of SimpleDB together with the cost for indexing all 10 datasets. Amazon charges 0.154 dollars per hour of utilization of a SimpleDB machine located in their Ireland facility. The attribute-based indexing strategy requires less machine utilization time and is thus more cost-efficient as well. On the other hand, the attribute-subset index is more expensive since it creates many more entries in SimpleDB than the rest of the indexes. Finally, while the attribute-pair and domain-per-dataset indexes create about the same attribute-value pairs to insert in SimpleDB, in the attribute-pair index we create many more domains and thus consume more of SimpleDB resources.

Querying

Query	**#q**	**#c**	Attrib. based	Attrib. pair	Attrib. subset	Domain per dataset
Q1	1	1	1	22	1	10
Q2	2	3	2	44	2	20
Q3	2	2	2	44	2	20
Q4	2	3	24	44	2	20
Q5	3	4	25	66	3	30
Q6	4	5	26	88	4	40
Q7	8	9	30	176	8	80

Table 6.9: Number of index look-up calls to SimpleDB, for each query and indexing strategy.

In this section, we present our preliminary results when evaluating various SPARQL BGP queries. The characteristics of the queries we used are shown in Table 6.9, where $\#q$ is the number of triple patterns and $\#c$ is the number of constant values each query contains. For this set of experiments, we have stored 10 RDF datasets in S3, each one consisting of 10.000 triples, and built the four indexes for this data. The number of domains created for each index is as described in Section 6.5.

In Table 6.9 we also depict the number of SimpleDB calls that were made for retrieving the appropriate URIs in each indexing strategy, i.e., `select` queries for the attribute-based, attribute-pair and domain-per-dataset index, and `GetAttributes` calls for the attribute-subset index. This table verifies our analytical cost model which shows that the number of lookups depends on the number of triple patterns in each query, as well as on the number of created domains. For example, query Q7 which consists of 8 triple patterns requires the largest number of lookups compared to the rest of the queries in all indexing strategies. Moreover, since the attribute-subset index consists of only one domain, the number of calls performed to SimpleDB for any SPARQL query is smaller than the calls performed by the other indexing

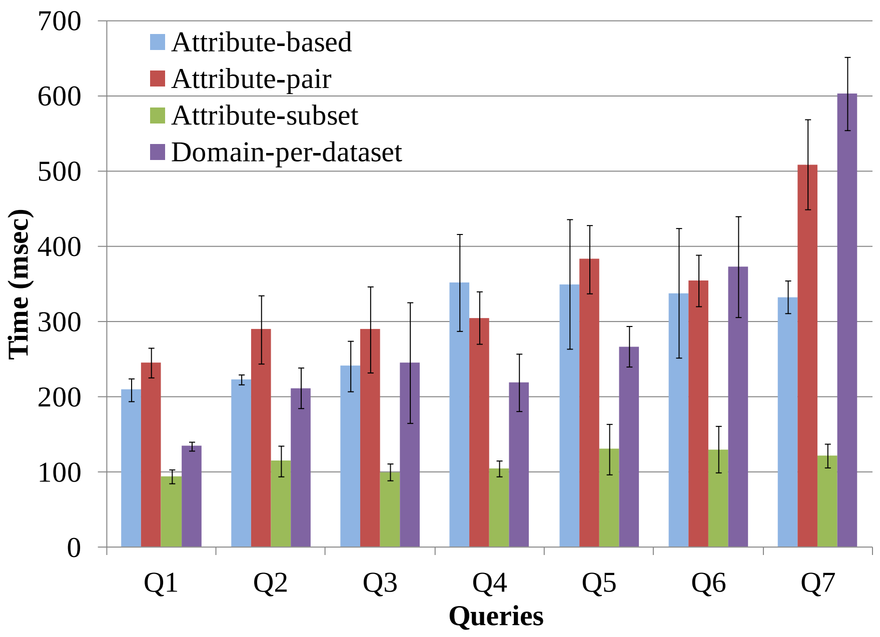


Figure 6.6: Index exploitation time for different queries.

strategies. The attribute-pair index, which allocates the largest number of SimpleDB domains, sustains a great amount of SimpleDB calls for any kind of queries. Observe that within AWS, calls to different SimpleDB domains are evaluated in parallel, and this parallelization also benefit our work, however, we did not attempt to split and parallelize computations beyond that; we are interested to do so as part of our future work.

In this experiment, we have also measured the time required for retrieving the final set of URIs required to evaluate a SPARQL query (index exploitation time). We show the index exploitation time in Figure 6.6 for each indexing strategy and for various SPARQL queries. This time includes the time to build the appropriate `select` queries or `GetAttributes` calls, the time required by SimpleDB to provide the results of these queries/calls, and the time for intersecting the URI sets thus obtained, whenever the look-up strategy requires such post-processing.

All measurements are averaged over 10 runs. Since previous studies established that the performance of Amazon EC2 performance may vary significantly over time [SDQR10, IYE11], we also depict the 95% confidence intervals. The attribute-subset indexing strategy outperforms the other strategies for all queries because it imposes the least amount of calls to SimpleDB. The attribute-based and attribute-pair indexes exhibit a similar performance with the former to perform slightly better because of the smaller amount of SimpleDB lookups. However for some queries (e.g., Q5 and Q6) the confidence intervals have a large overlap meaning that a sorting between the two strategies for such queries is not possible. Finally, although the domain-per-dataset indexing strategy gives better results than the aforementioned two indexes for most of the queries, it exhibits a very poor performance for queries which contain more than 5 triple patterns. This results from the large number of `select` queries posed to the same domains, which after a certain number of requests become a bottleneck.

Experiments conclusion

Our preliminary results show the feasibility and efficiency of our architecture as well as the performance of our proposed indexing strategies. Comparing the indexing strategies highlights a trade-off between the cost of the creation of an index (both monetary and time cost) and the efficiency on the lookup process. A prominent example of this trade-off is the attribute-subset indexing strategy which is the most expensive index to build but gives the best performance while querying. Using bigger datasets and more heterogeneous data is our next step of experimentation.

6.6 Discussion

The contribution of this work is the presentation of a novel architecture for the distributed management of RDF data stored in cloud infrastructures. We designed indexing techniques for retrieving the appropriate RDF files related to a specific query. We chose Amazon Web Services as a platform and implemented all our indexing strategies and in particular SimpleDB as No-SQL storage system for storing index data. We presented an analytical cost model and an experimental evaluation of both the indexing and the querying process our strategies are based on.

CHAPTER 7

Related Work

In this last section we introduce a general comparison between MIDST and other works in the literature. In the second section we compare MIDST approach with model driven architecture in particular considering GER (Generic Entity/object Relationship [lH89]). In the third section we describe NoSQL systems and finally in the fourth section we analyze NoSQL stores offered by Amazon Web Services and RDF indexing methodologies.

7.1 Model Management

The described approach illustrates a general approach to model management and relies on our previous work on model-generic schema and data translation [ACB05a, ACB06, ACG07, ACT+08] describing our conception and implementation of the Modelgen operator. There are many proposals addressing model management problems which have been put forward since the original formulation of the problem.

In [BHJ+00] Bernstein et al. recognize the possibility of a generic metadata approach to model management: their theoretical formalizations [Ber03] and later studies converged into Rondo, a programming platform for model management [Mel04]. However their approach is not supported by a description of models and so they pursue model independence without a concrete characterization of models and they cannot associate schemas with models. Conversely, MIDST (and now MISM) uses a dictionary of models and schemas to actually represent models and allows transparent transformations on them.

Our approach shares some analogies with Clio [FKMP03, FKP05, HHH$^+$05, MHH00, VMP03] too. Clio is aimed at building a completely defined mapping between two schemas, given a set of user-defined correspondences. As for our translations, these mappings could be translated into directly executable SQL, XQuery or XSLT transformations. However, in the perspective of adopting Clio in order to exchange data between two heterogeneous schemas, the needed mappings should be defined manually; moreover, there is no kind of model-awareness in Clio, which operates on a generalized nested relational model. Although this model can be shown to subsume a considerable amount of models, in a real application scenario a preliminary translation and adaptation of the operational system should be performed, leading to the problems of the initial MIDST approach.

A recent approach to schema evolution is PRISM [CMZ08]. Citing the authors, PRISM provides an intuitive, operational interface, used by the database administrator to evaluate the effect of possible evolution steps with respect to redundancy, information preservation, and impact on queries. In detail, the administrator can use a Schema Modification Operators (SMO) [BGMN08] language in order to specify schema changes and check whether such a modification could cause information loss, introduce redundancy, or grant invertibility. Moreover, the system allows for an automatic migration of the data, grants compatibility with old queries (i.e. against an old schema), and maintains the schema history. We propose something wider in which this approach can fit well: with reference to our running example, for instance, we could use similar techniques in order to constrain the evolutionary step between implementation schemas, thus granting the aforementioned desirable properties.

Our approach, together with Bernstein's, is more general and proposes a global platform for model management where the generation of executable mappings, like Clio's or PRISM's, is a complementary feature.

7.2 Model Driven Architectures

Hainaut's [Hai06] approach to translations does not initially face the theoretical issue of formulating a schema translation problem, instead moves from the practical engineering problem of modeling by successive abstractions.
This practice is indeed common in database engineering as well as in many other software design fields such as Model Driven Architectures. As a clear evidence, most databases rely on a layered architecture with an increasing level of abstraction: physical layer, logical layer and conceptual layer. In each layer a different model is adopted, in the sense that elementary constructs that are used to model the structures are different (entities and attributes, tables and columns, records and fields). In this context there is an engineering process that allows a schema of one model to be translated into a schema of another model. For example an ER schema used to conceptually model the database can be translated into the logical specification by means of a series of well known and consolidated rules. Of course, these translations preserve the information: indeed the semantics intended in the conceptual definition is preserved in the logical layer.
A similar process is followed in MDA's, where the designer draws technical UML and classes and code are directly derived from it without loss of semantics. Hainaut individuates the engineering context for translations and formerly deals with the practical aspects of translations. In this perspective, the need for a generic model acting as a pivot comes out. GER is indicated as a possible choice for this generic model. Indeed, Hainaut shows how any complex transformation can be defined in terms of GER constructs. Hainaut approach, indeed practical and concrete is theoretically formalized with the ERM (extended relational model). GER is an extended entity-relationship model, including entity types, domain, attributes, keys, relationships and constraints. The level is layered, in the sense that it is hierarchically defined in three levels of abstraction: conceptual model, logical model and physical model. The common enterprise data processing experience suggests the existence of hundreds of operators.

In GER, the key for expressing them lies in ERM formalism, while in GER they are expressed in a conceptual fashion. In this way we can classify: mutation transformations, other elementary transformations, compound transformations, predicate-driven transformations and model-driven transformations.
Mutation transformations change the nature of a model object: for example a relationship becomes a table (in relational language) or an attribute turns into a table; a table being split into two tables can be also classified as a mutation transformation. Hainaut points out the fact that mutations can solve the most of database engineering problems, since these kind of transformations summarize what is commonly known as translation. Other elementary transformations can directly involve data types, for example arrays and data sets. Compound transformations refer to the fact that elementary ones can be combined in a chain to build an aggregate and semantically complex transformation. This is similar to the definition of a composition operator, as it happens in schema mappings context. In that scenario, transformations are defined by mappings and compositions of mappings are mappings themselves.
Remaining in the data exchange parallel, predicate-driven transformations are a good paraphrases for the conjunction of atoms in the left part of a TGD. Indeed, given a more general representation for a set of data models (like the one GER is), it is possible to drive transformations with respect of the structural role played by source constructs (being the source and the target the operands and the output of a translation). GER allows to define predicates on the constructs in such a way that they can be involved in a transformation only if they meet certain requirements: for example being of a specific type, being connected to other constructs in a particular fashion and so on. A trivial example is: convert only N-ary relationships into entities. It is particularly interesting the fact that the inverse translation is not easily meaningful in this perspective. Model-driven transformations are an evolution of predicate-driven ones. In fact, they are the adoption of predicates to build a path of translations translating a source model into a target one. The key is the definition of a structural predicate asserting the set of constructs that can belong to a model. By difference, this defines

a set of constructs that are not allowable in a model. Therefore, given a source and a target model, all the constructs in the source that are not allowed in the target must be removed by a model-driven translation. This removal is called transformation plan that is the practical form, the algorithm defining the core of the translation.
The role of ERM can be finally clarified. It is a theory defining instances of GER in terms of logical propositions. Therefore we find the concept of axioms, theorems, interpretations and so on, as for the classical relational model. Moreover, common relational constraints such as functional dependencies and multi-value dependencies are supported. Models are represented in GER as propositions in ERM. Hainaut build a system of signatures that are logical statements allowing to infer if a schema belongs to a model and build an inference engine for transformations. In particular, model-driven transformations can be guided by a logical inference process leading from the axioms of one model to the ones of another.

7.3 NoSQL Systems

In this work we describe SOS, a uniform programming interface for NoSQL databases.

To the best of our knowledge SOS is the first proposal that aims to provide a solution to handle the heterogeneity of NoSQL databases.

The approach for SOS we present here uses the meta-layer as the principal means to support the heterogeneity of different data models. The idea of a pivot model finds its basis in the MIDST and MIDST-RT tools [ACT$^+$08, ABBG09a]. In MIDST, the core model (named "supermodel"), is the one to which every other model converges. Whereas MIDST faces heterogeneity through explicit translations of schemas, in SOS schemas are implied and translations are not needed. The pivot model, the meta-layer, is used as a common interface. Several other differences exist between the two approaches, as we already remarked in the introduction.

MIDST suffers from being a completely off line approach to schema and data translation and, indeed, MIDST-RT overcomes this problem. On the other hand, the need for a runtime support to interoperability of heterogeneous systems based on model and schema translation was pointed out by Bernstein and Melnik [BM07] and proposals in this direction, again for traditional (relational and object-oriented) models were formulated by Terwilliger et al. [TMB08] and by Mork et al. [MBM07a]. With reference to NoSQL models, SOS is the first proposal in the runtime direction: in fact, the whole algorithm takes place at runtime and direct access to the system is granted.

From a theoretical point of view, the need for a uniform classification and principle generalization for NoSQL databases is getting widely recognized; it was described by by Cattell [Cat10], reporting a detailed characterization of non-relational systems.

Stonebraker [Sto11b] presents a radical approach tending to diminish the importance of NoSQL systems in the scientific contest. Actually, Stonebraker denounces the absence of a consolidated standard for NoSQL models. Also, he uses the absence of a formal query language as a supplementary argument for his thesis. Here we move from the assumption that non-relational systems have a less strict data model which cannot be subsumed under a fixed set of rules as easily as for the relational system. However, we notice that commonalities and structural concepts among the various systems can be individuated and leverage this to build a common meta-layer.

Actually the meta-layer is used here as an aid for translation; in the future, it could be the basis to define a NoSQL standard query language.

7.4 RDF Stores and Amazon Web Services

Significant attention has been paid recently to RDF stores using cloud-based services. One system closely related to our work is Stratustore [SZ10], an RDF store that uses Amazon's SimpleDB as an RDF store back-end in combination with Jena's API. Stratustore indexes all triples in SimpleDB using the subjects of the triples as items, the properties as attribute names and the objects as the values of the attributes. A drawback of this approach is that SPARQL queries having a variable in the property position cannot be answered. The authors propose to insert one more entry per triple having as attribute names the objects with values the properties but this leads to a great increase in storage. The evaluation of Stratustore is performed using the Berlin SPARQL Benchmark [BS09]. Queries were executed with up to 20 simultaneously instances of Stratustore. Results show that performance is not competitive with other RDF stores such as Virtuoso. This is caused by the joins required for complex queries which have to be performed at the client side. However, as the number of Stratustore instances grows, the throughput of the system also increases.

The CumulusRDF [LH11] system uses Apache Cassandra, a nested key-value store, as a triple store back-end and proposes two different indexing strategies for storing RDF triples in Cassandra. The authors of [LH11] propose a hierarchical indexing scheme using supercolumns where all six combinations of subject, property, object are built-in indexes. In the second indexing scheme, called flat layout, simple columns are used where three main indexes are required together with a secondary index for several cases. CumulusRDF is evaluated in 8 machines using an instance of the DBPedia dataset and the queries used were only single triple pattern lookups. The authors conclude that their flat layout approach outperforms the hierarchical one. However, both Stratustore and CumulusRDF focus on providing full indexing capabilities in order to be able to answer SPARQL queries from indexes. Different from this approach, our main concern is to use the indexes for efficiently retrieving a smaller subset of datasets from which we are able to extract the answer to SPARQL queries using any in-memory RDF store.

Dydra [Dyd] is an RDF store relying on the Amazon EC2 infrastructure which provides a SPARQL endpoint to query the data stored. Although Dydra addresses an RDF data management problem similar to our, there is not much information available revealing the details of their approach.

Various works using MapReduce and related technologies have appeared in the literature as well. These works focus on developing large-scale RDF stores using the MapReduce paradigm. [MT08] is one of the first works to introduce cloud computing in the area of Semantic Web. It gives some preliminary experimental results using Apache Hadoop, a very popular implementation of MapReduce and Pig, a tool that translates queries expressed in Pig Latin to MapReduce jobs. In [HKKT10] the authors use Hadoop and propose a specific storage scheme that partitions RDF files into smaller ones to be stored in HDFS, the file system of Hadoop. They also use summary statistics to determine the best plan to evaluate a SPARQL query. [MYL10] considers the evaluation of SPARQL basic graph pattern queries in a MapReduce framework. Specifically, the authors propose a multi-way join algorithm to process SPARQL queries efficiently, as well as two methods to select the best query plan for executing the joins. Experiments were conducted with Cloudera's Hadoop distribution on the Amazon EC2. Finally, [SPZL11] presents a method to map SPARQL queries to Pig Latin queries.

CHAPTER 8

Conclusions

In this work, we covered the problem of heterogeneity under a number of perspective, presenting the novel results in terms of achieved homogeneity and model-independence.

With regard to Model Management, as an original result, in this work we discussed a paradigm and an application platform, allowing for model-independent solution to a wide range of problems. This system has been defined as a Model Management System and, as a major contribution, MISM has been proven to be an effective foundation to assemble definitions and implementations of model management operators. The research presented in this work showed how a correct orchestration of these operators can effectively lead to a model-independent solution to a wide variety of model management problems.

Important problems in model management were discussed in this work. Round-trip engineering was considered as the representative of a whole class of issues. A major target of the model management research is the development of an advanced software system managing all the involved problems (model management system). Such a system aims at providing applications with an abstraction layer towards data programmability issues, that is, the whole spectrum of application problems concerning data manipulation. The approach presented in this work lies in this direction.

MIDST represents a framework for model management problems; MISM is an enhanced version, where operators and solving procedures are specifically designed to maximize the abstraction level together with an effective and sound representation of schemas and models. In parallel, we are working on the development of runtime strategies and algorithms in order to make our solutions in step with large operational databases as well as compliant with the most expressive data models.

The presented algorithms are designed in an off line fashion, meaning that they need transformation steps in the supermodel where data and metadata have to be preliminary imported. Once the transformation is finished, they are exported back into the target system.

This is a point of attention affecting all MIDST architecture and our original metalevel methodology. Then in this work we introduced the runtime enhancement. It was presented with explicit reference to ModelGen operator; clearly a point of future investigation will pursue the extension of the runtime approach to every model management operator.

As far as MIDST-RT is concerned, an important result that is presented here is the generation of executable statements out of translation rules. The approach aims at being general, in the sense that the final objective is to derive an executable statement for any possible translation. Then, we have also shown some scenarios which may benefit from the usage of MIDST-RT, in order to allow flexibility and customization.

A major issue is the query language. It is necessary to specify a language capable of interacting with all the involved models homogeneously. Although, in some cases, such a single language would be available, other situations are more complex and need further investigation. Examples are the ones involving translations from object-relational to XML and vice versa. We have used here combinations of languages including SQL/XML and XQuery/SQL, over one single platform. In fact, the described solution actually refers to transformations taking place in a single system, offering the logical support to both models. Indeed, it may be the case that more systems are involved; however the adoption of the appropriate middleware solutions might offer working solutions based, for example, on a common exchange format.

Moreover, we have shown examples of relational views. These views have an intrinsic problem: in fact, when we define a relational view, it is quite probable that it will not be updatable. A possible solution to this problem (and possible future work) is the introduction of the concept of “reverse mapping” [MBM07b], a mapping that keeps trace of the origin of data shown by views in order to modify the source database when the user tries to modify a view.

Let us conclude by discussing a few issues where our approach shows some limitations that we are working to overcome. From the implementation point of view, it is clear that the target system will have some restrictions on how it deals with views (including materialization, persistence, update propagation). However this limitation is related to the specific target system and it comes as a direct consequence of the runtime perspective where no third-party actors interfere. From a theoretical point of view, open issues are related with the generality and correctness of the approach. As for generality of modeling, MIDST metamodel collects all the constructs most commonly used in models and can be extended whenever necessary. Extensions could also go towards a richer representation of semantics, where integrity constraints are described and supported, in the sense that their satisfaction is verified and reasoning on them can be performed. Clearly, this would require a different approach on the management of the supermodel, which would require additional features beside and beyond the relational implementation. In this respect, we are considering approaches based on description logics [BCM$^+$07].

The general solution archived in the research presented in this work allows for a more efficient treatment of translation and, theoretically, could be also thought of as a means of speeding up the algorithms we have proposed for Model Management.

For example, round-trip engineering, that we presented as a series of transformations within the supermodel, could be performed directly in the target system. However, the proposed algorithms are based on model management operators applied to materialized intermediate results for the sake of simplicity. A formulation of the procedures in a runtime fashion, will require additional reasonings about view update.

Finally, in the last part of this work, we put aside the traditional perspective of model management, to face heterogeneity in the emerging NoSQL systems. The adoption of a metalevel approach also in this case is a core result and acts as the foundation to build a uniform programming interface enabling a homogeneous treatment of non relational schemas. We provided a meta-layer that allows the creation and querying of NoSQL databases defined in MongoDB, HBase and Redis, representative examples of the main classes of specialized systems. A simple interface comprising a set of simple atomic operation was prepared. Finally we described an example where, the designed interface enables the contemporary use of NoSQL database transparently for the application and for the programmers.

An example of use of NoSQL system is given with the indexing strategies on SimpleDB that represents a valuable application within a cloud context.

In conclusion, our work aims at being a theoretical attempt to address data heterogeneity in all its modeling aspects, while adopting metalevel-based platforms to build applications on. Much future effort will be devoted to expanding our metalevel to not covered models and paradigms.

Indeed now, with increased awareness, we can firmly agree with David Wheeler: “All problems in computer science can be solved by another level of indirection”.

Acknowledgments

It was not my intention at first to include this paragraph in my work: I have always thought that acknowledgements are sad and I am not a sad person. Nonetheless I cannot conclude this work without expressing my gratitude to people that contributed to this thesis and made my life the way it is now.

At first I would like to thank my parents who gave me the opportunity to study. My mother really hates computers and everything related to them, computer science included! I think that it has been really hard for her to support me and my "crazy study" for the duration of this experience.

Huge and special thanks to Luigi for supporting me, my choices and my ideas every day of the last N years.

I would like to express my sincere gratitude to Prof. Paolo Atzeni who guided me in these years of my Phd. and even before. A special thank also to Prof. Luca Cabibbo and Paolo Papotti and to my colleagues Celine, Daniele, Giorgio, Lorenzo, Mirko, Piero, Roberto, Stefano. Don't worry I am going to continue bringing some sweets for supporting us during our work in Mais laboratory.. but only after six or seven months of holidays!

A word of gratitude is owed to Ioana Manolescu who gave me a last-minute internship at INRIA/LRI in Paris and supervised me during the four months I spent there. I had the honor to take part in the research work of an extraordinary group; thanks to all the friends who helped me during my stay: Alexandra, Andrè, Asterios, Coralie, Federico, Yannis, Jesus, and Konstantinos. *Merci beaucoup de votre encouragement!* It is a promise: I am going to study French.

I can't close this section without a special thanks to my best friend Francesca (we share the name and all the experiences) who helped me despite the troubles she had to overcome during the last two years.

Finally an uncommon and particular thank you to the only one who doesn't understand a lot of my problems or worries but who feels them before the others and who supported me every day, thank you my second best friend Tuono!

Bibliography

[ABB$^+$12] P. Atzeni, L. Bellomarini, F. Bugiotti, F. Celli, and G. Gianforme. A runtime approach to model-generic translation of schema and data. In *Inf. Syst.*, pages 269–287, 2012.

[ABBG08] P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. From schema and model translation to a model management system. In *BNCOD*, pages 227–240, 2008.

[ABBG09a] P. Atzeni, L. Bellomarini, F. Bugiotti, and Gianforme G. A runtime approach to model-independent schema and data translation. In *EDBT '09*, pages 275–286. ACM, 2009.

[ABBG09b] P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. MISM: A platform for model-independent solutions to model management problems. In *J. Data Semantics*, pages 133–161, 2009.

[ABR12a] P. Atzeni, F Bugiotti, and L. Rossi. SOS: A uniform programming interface for non-relational systems. *Caise*, 2012.

[ABR12b] P. Atzeni, F Bugiotti, and L. Rossi. SOS: A uniform programming interface for non-relational systems. *EDBT (demo)*, 2012.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann, 2000.

[ACB05a] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Modelgen: Model independent schema translation. In *ICDE Conference*, pages 1111–1112. IEEE Computer Society, 2005.

[ACB05b] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. A multilevel dictionary for model management. In *ER Conference, LNCS 3716*, pages 160–175, 2005.

[ACB06] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Model-independent schema and data translation. In *EDBT Conference, LNCS 3896*, pages 368–385. Springer, 2006.

[ACG07] Paolo Atzeni, Paolo Cappellari, and Giorgio Gianforme. MIDST: model independent schema and data translation. In *SIGMOD Conference*, pages 1134–1136. ACM, 2007.

[ACT$^+$08] P. Atzeni, P. Cappellari, R. Torlone, P.A. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB Journal*, 17:1347–1370, 2008.

[AEH$^+$11] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models. In *BTW*, 2011.

[AGC09] P Atzeni, G. Gianforme, and P. Cappellari. *A Universal Metamodel and Its Dictionary*, pages 38–62. Springer-Verlag, 2009.

[AT93] Paolo Atzeni and Riccardo Torlone. A metamodel approach for the management of multiple models and translation of schemes. *Information Systems*, 18(6):349–362, 1993.

[AT96] Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. In *EDBT Conference, LNCS 1057*, pages 79–95. Springer, 1996.

[AWS] AWS. Amazon Web Services. http://aws.amazon.com/.

[BCM$^+$07] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2007.

[Ber03] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR Conference*, pages 209–220, 2003.

[BFG$^+$08] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *SIGMOD*, 2008.

[BGKM12] F Bugiotti, F. GoasdouŔ, Z. Kaoudi, and I. Manolescu. RDF data management in the amazon cloud. *Danac*, 2012.

[BGMN08] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing mapping composition. *The VLDB Journal*, pages 333–353, 2008.

[BHJ$^+$00] P. Bernstein, L. Haas, M. Jarke, E. Rahm, and G. Wiederhold. Panel: Is generic metadata management feasible? In *VLDB*, pages 660–662, 2000.

[BHP00] Philip A. Bernstein, Alon Y. Halevy, and Rachel Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[BK06] C. Bauer and G. King. *Java Persistence with Hibernate.* Manning Publications Co., 2006.

[BM07] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*, pages 1–12, 2007.

[BS95] M. L. Brodie and M. Stonebraker. *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann Publishers Inc., 1995.

[BS09] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[Cab98] L. Cabibbo. The expressive power of stratified logic programs with value invention. *Inf. Comput.*, 147:22–56, 1998.

[Cat10] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, pages 12–27, 2010.

[CMZ08] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.*, pages 761–772, 2008.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod82] E. F. Codd. Relational database: A practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982.

[CRV00] M. J. Carey, S. Rielau, and B. Vance. Object view hierarchies in DB2 UDB. In *EDBT*, EDBT '00, pages 478–492, 2000.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design and Implementation*, 2004.

[Dyd] Dydra. Dydra. http://dydra.com/.

[FKMP03] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.

[FKP05] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[FKPT07] R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Quasi-inverses of schema mappings. In *ACM SIGMOD-SIGACT-SIGART*, PODS '07, pages 123–132. ACM, 2007.

[Haa07] Laura M. Haas. Beauty and the beast: The theory and practice of information integration. In Thomas Schwentick and Dan Suciu, editors, *ICDT*, volume 4353 of *Lecture Notes in Computer Science*, pages 28–43. Springer, 2007.

[HAB$^+$05] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael J. Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal, and Vishal Sikka. Enterprise information integration: successes, challenges and controversies. In *SIGMOD Conference*, pages 778–787, 2005.

[Hai06] Jean-Luc Hainaut. The transformational approach to database engineering. In *GTTSE, LNCS 4143*, pages 95–143. Springer, 2006.

[HHH$^+$05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 805–810. ACM, 2005.

[HK87] R.B. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HKKT10] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. Data Intensive Query Processing for Large

RDF Graphs Using Cloud Computing Tools. In *3rd International Conference on Cloud Computing*, pages 1–10, 2010.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Sixteenth International Conference on Very Large Data Bases, Brisbane (VLDB'90)*, pages 455–468, 1990.

[IYE11] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2011.

[Jor03] D. Jordan. *Java Data Objects.* O'Reilly, 2003.

[KAKK10] Reto Krummenacher, Karl Aberer, Atanas Kiryakov, and Rajaraman Kanagasabai. Workshop on semantic data management: a summary report. *SIGMOD Record*, 39(3):24–26, 2010.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004.

[lH89] Jean luc Hainaut. A generic entity-relationship model. In *in Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an in-depth analysis, North-Holland*, 1989.

[LH11] Gunter Ladwig and Andreas Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In *7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, 2011.

[MAB07] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. In *ACM SIGMOD*, pages 461–472, 2007.

[MBM07a] P. Mork, P.A. Bernstein, and S. Melnik. A schema translator that produces object-to-relational views. Technical Report MSR-TR-2007-36, Microsoft Research, 2007. http://research.microsoft.com.

[MBM07b] Peter Mork, Philip A. Bernstein, and Sergey Melnik. Teaching a schema translator to produce O/R views. In *ER Conference, LNCS 4801*, pages 102–119. Springer, 2007.

[McG59] W. C. McGee. Generalization: Key to successful electronic data processing. *J. ACM*, 6(1):1–23, 1959.

[Mel04] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*. Springer-Verlag, 2004.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[MM04] Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, February 2004.

[MP99] Peter McBrien and Alexandra Poulovassilis. A uniform approach to inter-model transformations. In *CAiSE Conference, LNCS 1626*, pages 333–348, 1999.

[MRB03] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A programming platform for generic model management. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD Conference*, pages 193–204. ACM, 2003.

[MT08] Peter Mika and Giovanni Tummarello. Web Semantics in the Clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[MYL10] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In *Workshop on Massive Data Analytics on the Cloud*, pages 6:1–6:6, 2010.

[NW09] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[NW10] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1), 2010.

[PS08] E. Prud'hommeaux and A. Seaborn. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[PT05] Paolo Papotti and Riccardo Torlone. Heterogeneous data translation through XML conversion. *J. Web Eng.*, 4(3):189–204, 2005.

[RB01] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB JOURNAL*, 10:2001, 2001.

[SC11] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, pages 72–80, 2011.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3, September 2010.

[SHLP09] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *29th International Conference on Data Engineering*, 2009.

[SM08] A. Smith and P. Mcbrien. A generic data level implementation of modelgen. In *BNCOD '08*, pages 63–74. Springer-Verlag, 2008.

[SMA$^+$07] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[SPZL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *International Workshop on Semantic Web Information Management*, pages 4:1–4:8, 2011.

[Sto10] M. Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53:10–11, April 2010.

[Sto11a] M. Stonebraker. Stonebraker on nosql and enterprises. *Commun. ACM*, 54:10–11, August 2011.

[Sto11b] M. Stonebraker. Stonebraker on NoSQL and enterprises. *Commun. ACM*, pages 10–11, 2011.

[SZ10] Raffael Stein and Valentin Zacharias. RDF On Cloud Number Nine. In *4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable & Dynamic*, pages 11–23, May 2010.

[TL82] D. Tsichritzis and F.H. Lochovski. *Data Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[TMB08] J. F. Terwilliger, S. Melnik, and P. A. Bernstein. Language-integrated querying of XML data in SQL server. *Proc. VLDB Endow.*, 2008.

[VMP03] Yannis Velegrakis, Renée J. Miller, and Lucian Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.