Forward Engineering Relational Schemas and High-Level Data Access from Conceptual Models

Gustavo L. Guidoni[1,2], João Paulo A. Almeida[1], and Giancarlo Guizzardi[1,3]

[1] Ontology & Conceptual Modeling Research Group (NEMO),
Federal University of Espírito Santo, Vitória, Brazil
[2] Federal Institute of Espírito Santo, Colatina, Brazil
[3] Free University of Bozen-Bolzano, Italy
gustavo.guidoni@ifes.edu.br, jpalmeida@ieee.org, gguizzardi@unibz.it

Abstract. Forward engineering relational schemas based on conceptual models is an established practice, with a number of commercial tools available and widely used in production settings. These tools employ automated transformation strategies to address the gap between the primitives offered by conceptual modeling languages (such as ER and UML) and the relational model. Despite the various benefits of automated transformation, once a database schema is obtained, data access is usually undertaken by relying on the resulting schema, at a level of abstraction lower than that of the source conceptual model. Data access then requires both domain knowledge and comprehension of the (non-trivial) technical choices embodied in the resulting schema. We address this problem by forward engineering not only a relational schema, but also creating an ontology-based data access mapping for the resulting schema. This mapping is used to expose data in terms of the original conceptual model, and hence queries can be written at a high level of abstraction, independently of the transformation strategy selected.

Keywords: Forward Engineering · Model-Driven Approach · Ontology-Based Data Access · Model Transformation.

1 Introduction

Forward engineering relational schemas based on conceptual models (such as ER or UML class diagrams) is an established practice, with a number of commercial tools available and widely used in production settings. The approaches employed establish correspondences between the patterns in the source models and in the target relational schemas. For example, in the *one table per class* approach [11, 16], a table is produced for each class in the source model; in the *one table per leaf class* approach [21], a table is produced for each leaf class in the specialization hierarchy, with properties of superclasses accommodated at the tables corresponding to their leaf subclasses. An important benefit of all these approaches is the automation of an otherwise manual and error-prone schema design process. Automated transformations capture tried-and-tested design decisions, improving productivity and the quality of the resulting schemas.

Despite the various benefits of automated transformation, once a database schema is obtained, data access is usually undertaken by relying on the resulting schema, at a level

of abstraction lower than that of the source conceptual model. As a consequence, data access requires both domain knowledge and comprehension of the (non-trivial) technical choices embodied in the resulting schema. For example, in the *one table per class* approach, joint access to attributes of an entity may require a query that joins several tables corresponding to the various classes instantiated by the entity. In the *one table per leaf class* approach, queries concerning instances of a superclass involve a union of the tables corresponding to each of its subclasses. Further, some of the information that was embodied in the conceptual model—in this case the taxonomic hierarchy—is no longer directly available for the data user.

In addition to the difficulties in accessing data at a lower level of abstraction, there is also the issue of the suitability of the resulting schema when considering its quality characteristics (such as usability, performance). This is because a transformation embodies design decisions that navigate various trade-offs in the design space. These design decisions may not satisfy requirements in different scenarios. Because of this, there is the need to offer the user more than one transformation to choose from or to offer some control over the design decisions in the transformation.

This paper addresses these challenges. We identify the primitive refactoring operations that are performed in the transformation of conceptual models to relational schemas, independently of the various transformation strategies. As the transformation advances, at each application of an operation, we maintain a set of traces from source to target model, ultimately producing not only the relational schema, but also a high-level data access mapping for the resulting schema using the set of traces. This mapping exposes data in terms of the original conceptual model, and hence queries can be written at a high level of abstraction, independently of the transformation strategy selected.

This paper is further structured as follows: Section 2 discusses extant approaches to the transformation of a structural conceptual model into a target relational schema, including the definition of the primitive "flattening" and "lifting" operations that are executed in the various transformation strategies; Section 3 presents our approach to generating a high-level data access mapping in tandem with the transformation of the conceptual model; Section 4 compares the performance of high-level data access queries with their counterparts as handwritten SQL queries; Section 5 discusses related work, and, finally, Section 6 presents concluding remarks.

2 Transformation of Conceptual Models into Relational Schemas

2.1 Extant Approaches

The relational model does not directly support the concept of inheritance, and, hence, realization strategies are required to preserve the semantics of a source conceptual model in a target relational schema [13]. Such strategies are described by several authors [2, 11, 16, 18, 21] under various names. We include here some of these strategies solely for the purpose of exemplification (other approaches are discussed in [13]).

One common approach is the *one table per class* strategy, in which each class gives rise to a separate table, with columns corresponding to the class's features. Specialization between classes in the conceptual model gives rise to a foreign key in the table that

corresponds to the subclass. This foreign key references the primary key of the table corresponding to the superclass. This strategy is also called "class-table" [11], "vertical inheritance" [21] or "one class one table" [16]. In order to manipulate a single instance of a class, e.g., to read all its attributes or to insert a new instance with its attributes, one needs to traverse a number of tables corresponding to the depth of the whole specialization hierarchy. A common variant of this approach is the *one table per concrete class* strategy. In this case, an operation of "flattening" is applied for the abstract superclasses. In a nutshell, "flattening" removes a class from the hierarchy by transferring its attributes and relations to its subclasses. This reduces the number of tables required to read or to insert all attributes of an instance, but introduces the need for unions in polymorphic queries involving abstract classes.

The extreme application of "flattening" to remove all non-leaf classes of a taxonomy yields a strategy called *one table per leaf class*. In this strategy, also termed "horizontal inheritance" [21], each of the leaf classes in the hierarchy gives rise to a corresponding table. Features of all (non-leaf) superclasses of a leaf class are realized as columns in the leaf class table. No foreign keys emulating inheritance are employed in this approach.

A radically different approach is the *one table per hierarchy* strategy, also called "single-table" [11] or "one inheritance tree one table" [16]. It can be understood as the opposite of *one table per leaf class*, applying a "lifting" operation to subclasses instead of the "flattening" of superclasses. Attributes of each subclass become optional columns in the superclass table. This strategy usually requires the creation of an additional column to distinguish which subclass is (or which subclasses are) instantiated by the entity represented in the row (a so-called "discriminator" column). The "lifting" operation is reiterated until the top-level class of each hierarchy is reached.

Other approaches propose the combined and selective use of both "lifting" and "flattening". For example, the approach we have proposed called *one table per kind* [13] uses ontological meta-properties of classes to guide the "flattening" of the so-called *non-sortals* in the conceptual model and the "lifting" of all (non-kind) sortals.

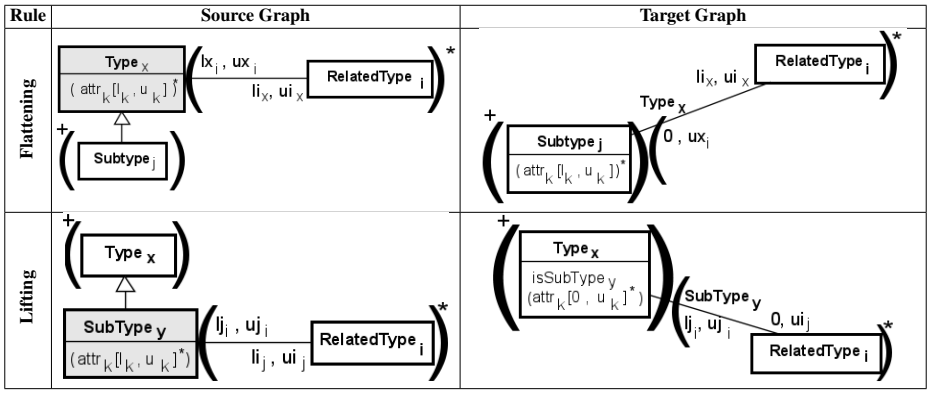
2.2 Flattening and Lifting Operations

In order to support the transformation strategies discussed in the previous section and their variations, we define our approach in this paper in terms of the "flattening" and "lifting" operations. Here we present these operations in detail, including their consequences to the existing attributes and associations in the model.

In the flattening operation, shown in the first row of Table 1, every attribute of the class that is to be removed from the model (in gray) is migrated to each of its direct subclasses. Association ends attached to the flattened superclass are also migrated to the subclasses. The lower bound cardinality of the migrated association end is relaxed to zero, as the original lower bound may no longer be satisfied for each of the subclasses.

In the lifting operation, shown in the last row of Table 1, every attribute of the class that is lifted (in gray) is migrated to each direct superclass, with lower bound cardinality relaxed to zero. Association ends attached to the lifted class are migrated to each direct superclass. The lower bound cardinality constraints of the association ends attached to classes other than the lifted class ($RelatedType_i$) are relaxed to zero.

Table 1. Transformation Patterns.



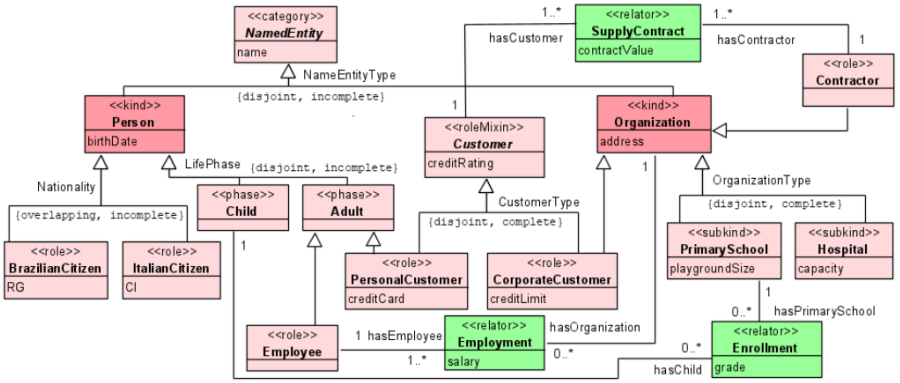
When no generalization set is present, a Boolean attribute is added to each superclass, to indicate whether the instance of the superclass instantiates the lifted class ($isSubType_j$). If a generalization set is used, a discriminator enumeration is created with labels corresponding to each $SubType_j$ of the generalization set. An attribute with that discriminator type is added to each superclass. Its cardinality follows the generalization set: it is optional for incomplete generalization sets (and mandatory otherwise); and multivalued for overlapping generalization sets (and monovalued otherwise).

2.3 Example Transformation

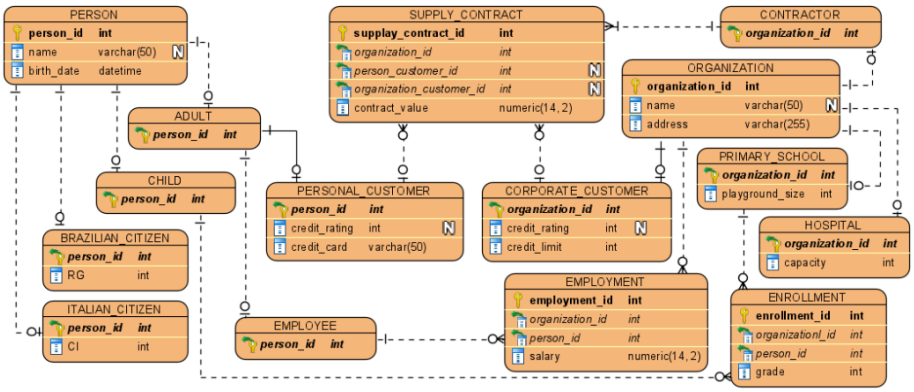
In order to illustrate the consequences of the transformation strategy selected on the resulting relational schema and on the production of queries, we apply here two different transformations to the model shown in Figure 1(a). The transformation approaches selected for illustration are *one table per concrete class* and *one table per kind*, and are used throughout the paper. The former relies on the flattening of abstract classes, and the latter makes use of OntoUML stereotypes [14] to guide the application of flattening and lifting. Flattening is applied to non-sortals (in this example, the classes stereotyped «category» and «roleMixin») and lifting is applied to sortals other than kinds (in this example, the classes stereotyped «subkind», «phase» and «role»).

Figure 1(b) presents the resulting schema in the *one table per concrete class* strategy. The abstract classes `NamedEntity` and `Customer` have been flattened out, and a table is included for each concrete class. Foreign keys are used to emulate inheritance (e.g., the table `HOSPITAL` corresponding to the concrete class `Hospital` has a foreign key `organization_id` which is the primary key of `ORGANIZATION`). The strategy generates a relatively large number of tables.

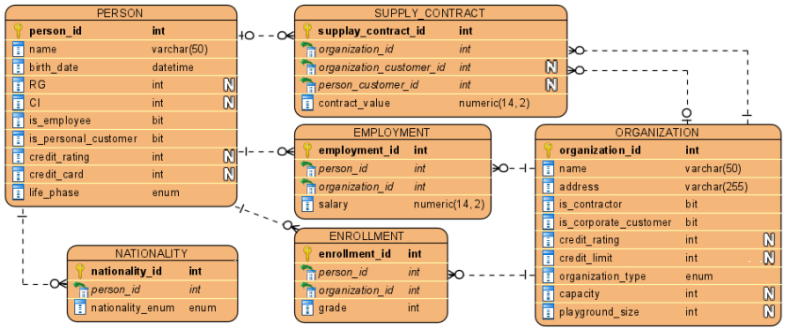
Figure 1(c) presents the result of applying the *one table per kind* strategy. Again, `NamedEntity` and `Customer` (abstract classes) are flattened out. In addition, all concrete classes are lifted until only one concrete class (the kind) remains. There is no emulation of inheritance with foreign keys; discriminators are used instead to identify the subclasses that are instantiated by an entity. No joins are required to access jointly attributes of an entity.



(a) Running Example



(b) Running example transformed by "*one table per concrete class*" strategy



(c) Running example transformed by "*one table per kind*" strategy

Fig. 1. Transformations.

2.4 Data Access

Once a database schema is obtained, data access needs to take into account the resulting schema. Consider, e.g., that the user is interested in a report with "the Brazilian citizens who work in hospitals with Italian customers". This information need would require two different queries depending on the transformation strategy. Listing 1 shows the query for the *one table per concrete class* strategy and Listing 2 for *one table per kind*.

Listing 1. Query on the schema of Figure 1(b), adopting *one table per concrete class*

```
select  p.name brazilian_name, o.name organization_name,
        sc.contract_value, p2.name italian_name
from    brazilian_citizen bc
join    person p
        on  bc.person_id        = p.person_id
join    employment em
        on  p.person_id         = em.person_id
join    organization o
        on  om.organization_id  = o.organization_id
join    hospital h
        on  o.organization_id   = h.organization_id
join    supply_contract sc
        on  o.organization_id   = sc.organization_id
join    person p2
        on  sc.person_id        = p2.person_id
join    italian_citizen ic
        on  p2.person_id        = ic.person_id
```

Listing 2. Query on the schema of Figure 1(c), adopting *one table per kind*

```
select  p.name brazilian_name, o.name organization_name,
        sc.contract_value, p2.name italian_name
from    person p
join    nationality n
        on  p.person_id              = n.person_id
        and n.nationality_enum       = 'BRAZILIANCITIZEN'
join    employment em
        on  p.person_id              = em.person_id
join    organization o
        on  em.organization_id       = o.organization_id
        and o.organization_type_enum = 'HOSPITAL'
join    supply_contract sc
        on  o.organization_id        = sc.organization_id
join    person p2
        on  sc.person_id             = p2.person_id
join    nationality n2
        on  p2.person_id             = n2.person_id
        and n2.nationality_enum      = 'ITALIANCITIZEN'
```

Note that the second query trades some joins for filters. In the *one table per concrete class* strategy, many of the joins are used to reconstruct an entity whose attributes are spread throughout the emulated taxonomy. In the *one table per kind* strategy, filters are applied using the discriminators that are added to identify the (lifted) classes that are instantiated by an entity. The different approaches certainly have performance implications (which will be discussed later in this paper). Regardless of those implications, we can observe that the database is used at a relatively low level of abstraction, which is dependent on the particular realization solution imposed by the transformation strategy. This motivates us to investigate a high-level data access mechanism.

3 Synthesizing High-Level Data Access

We reuse a mature Ontology-Based Data Access (OBDA) technique [19] to realize data access in terms of the conceptual model. OBDA works with the translation of high-level queries into SQL queries. Users of an OBDA solution are required to write a mapping specification that establishes how entities represented in the relational database should be mapped to instances of classes in a computational (RDF- or OWL-based) ontology. The OBDA solution then enables the expression of queries in terms of the ontology, e.g., using SPARQL. Each query is automatically rewritten by the OBDA solution into SQL queries that are executed at the database. Results of the query are then mapped back to triples and consumed by the user using the vocabulary established at the ontology.

In our approach, instead of having the OBDA mapping specification written manually, we incorporate the automatic generation of this mapping specification into the transformation. Therefore, our transformation not only generates a target relational schema, but also generates an OBDA mapping specification to accompany that schema. Although the source models we consider are specified here with UML (or OntoUML), a transformation to OWL is used as part of the overall solution; this transformation preserves the structure of the source conceptual model, and hence SPARQL queries refer to classes in that model. The overall solution is implemented as a plugin to Visual Paradigm[4] (also implementing the *one table per kind* transformation from [13]).

3.1 Tracing Flattening and Lifting

In order to generate the OBDA mapping specification, we keep a set of *traces* at each application of the operations of flattening and lifting. In sum, a trace establishes the class that is flattened or lifted and the target class to which its attributes are migrated. With the final set of traces, we are able to synthesize the OBDA mapping for Ontop [5], the OBDA solution we adopt.

As discussed in Section 2.2, the flattening operation consists of removing a superclass and migrating its attributes to each subclass, with association ends attached to the flattened superclass also migrated to each subclass. Each time the flattening operation is executed, one trace is produced for each pair of flattened superclass and subclass. For example, the flattening of `NamedEntity`, depicted by the green dashed arrows in Figure 2, creates one trace from this class to `Person` and another to `Organization`. Likewise, the flattening of `Customer`, creates one trace from this class to `PersonalCustomer` and another to `CorporateCustomer`.

Naturally, tracing for lifting occurs in the opposite direction in the hierarchy. For every class lifted, traces are created from the lifted subclass to its superclasses. Differently from flattening, the traces for lifting require the specification of a "filter" determining the value of the discriminator. This filter is used later to preserve information on the instantiation of the lifted subclass. For example, the lifting of `Child` creates a trace from that class to `Person` (represented in blue arrows in Figure 2). However, not every `Person` instance is a `Child` instance. The added filter thus requires the discriminator `lifePhase='Child'`.

[4] `http://purl.org/guidoni/ontouml2db`

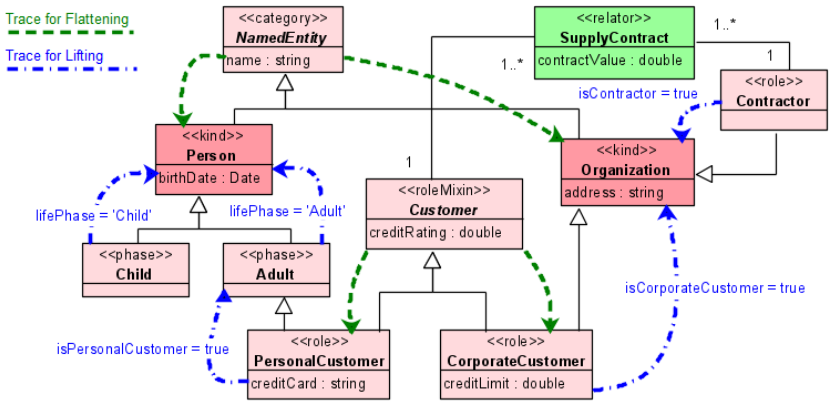


Fig. 2. Tracing example for each execution of Flattening and Lifting.

3.2 Generating the Data Mapping

For each tracing path from a class in the source model to a 'sink' class (one with no outgoing traces), a mapping is generated by composing all the traces along the path. Any discriminator filters in a path are placed in a conjunction. For example, there are two end-to-end traces for `Customer`, mapping it to (i) the `PERSON` table (through `PersonalCustomer` and `Adult`), provided `isPersonalCustomer=true` and `lifePhase='Adult'` and also to (ii) the `ORGANIZATION` table, provided `isCorporateCustomer=true`. Table 2 shows all composed traces in the application of the *one table per kind* strategy in the running example.

For each end-to-end trace and for each class that is neither flattened nor lifted, an Ontop mapping is produced. Mappings are specified in Ontop as *target* Turtle templates for the creation of an instance of the ontology corresponding to *source* tuples that are obtained by simple SQL queries in the relational database. In the following, we present examples of these mappings generated for three classes of the running example in the *one table per kind* strategy: (i) a class that is neither flattened nor lifted (`Person`); (ii) a class that is flattened (`NamedEntity`) and (iii) a class that is lifted (`ItalianCitizen`).

Listing 3 shows the Ontop mapping generated for the kind `Person`. Given the absence of flattening and lifting, the mapping establishes the straightforward correspondence of a source entry in the `PERSON` table and a target instance of `Person`; the primary key of the `PERSON` table is used to derive the URI of the instance of `Person` corresponding to each entry of that table. Labels between brackets in the target template of the mapping (`{person_id}` and `{birth_date}`) are references to values in the source pattern select clause. Corresponding attributes (`birthDate`) are mapped one-to-one.

Listing 3. OBDA mapping for the `Person` class in Ontop.

```
mappingId  RunExample-Person
target     :RunExample/person/{person_id} a :Person ;
           :birthDate {birth_date}^^xsd:dateTime .
source     SELECT person.person_id, person.birth_date
           FROM person
```

Table 2. End-to-end traces involving flattened or lifted classes.

Trace	Source Class	Target Table	Discriminator Conditions
1	NamedEntity	PERSON	-
2	NamedEntity	ORGANIZATION	-
3	Customer	ORGANIZATION	is_corporate_customer = true
4	Customer	PERSON	is_personal_customer = true
			life_phase_enum = 'ADULT'
5	BrazilianCitizen	PERSON	nationality_enum = 'BRAZILIANCITIZEN'
6	ItalianCitizen	PERSON	nationality_enum = 'ITALIANCITIZEN'
7	Child	PERSON	life_phase_enum = 'CHILD'
8	Adult	PERSON	life_phase_enum = 'ADULT'
9	Employee	PERSON	is_employee = true
			life_phase_enum = 'ADULT'
10	PersonalCustomer	PERSON	is_personal_customer = true
			life_phase_enum = 'ADULT'
11	PrimarySchool	ORGANIZATION	organization_type_enum = 'PRIMARYSCHOOL'
12	Hospital	ORGANIZATION	organization_type_enum = 'HOSPITAL'
13	CorporateCustomer	ORGANIZATION	is_corporate_customer = true
14	Contractor	ORGANIZATION	is_contractor = true

Listing 4 shows the mappings generated for a class that is flattened: `NamedEntity`. Because the class is flattened to two subclasses, two mappings are produced, one for each table corresponding to a subclass (`PERSON` and `ORGANIZATION`). Since attributes of the flattened superclass are present in each table, one-to-one mappings of these attributes are produced.

Listing 4. OBDA mapping for the flattened `NamedEntity` class in Ontop.

```
mappingId  RunExample-NamedEntity
target     :RunExample/person/{person_id} a :NamedEntity ;
           :name {name}^^xsd:string .
source     SELECT person.person_id, person.name
           FROM person

mappingId  RunExample-NamedEntity2
target     :RunExample/organization/{organization_id} a :NamedEntity;
           :name {name}^^xsd:string .
source     SELECT organization.organization_id, organization.name
           FROM organization
```

Listing 5 shows the mapping generated for a class that is lifted (`ItalianCitizen`) to the `Person` class, again in the *one table per kind* strategy. Here, the filter captured during tracing are included in the SQL query to ensure that only instances of the lifted superclass are included. Because of the multivalued discriminator employed to capture the overlapping generalization set `Nationality`, a join with a discriminator table is required (otherwise, a simple filter would suffice). For performance reasons, an index

is created in the transformation for enumerations corresponding to generalization sets. The complete specification with resulting mappings for this example can be obtained in `https://github.com/nemo-ufes/forward-engineering-db`.

Listing 5. OBDA mapping for the lifted class `ItalianCitizen` in Ontop.

```
mappingId RunExample-ItalianCitizen
target    :RunExample/person/{person_id} a
          :ItalianCitizen ; :CI {ci}^^xsd:string .
source    SELECT person.person_id, person.ci
          FROM person
          JOIN nationality
            ON person.person_id = nationality.person_id
           AND nationality.nationality_enum = 'ITALIANCITIZEN'
```

4 Performance of Data Access

In order to evaluate the performance of data access in our approach, we have created a randomly populated database using the models in Figure 1. We have employed the two transformation strategies as discussed before. Our main objective was to consider the overhead of the high-level data access approach. Because of that, we have contrasted the time performance of handwritten SQL queries with those automatically rewritten from SPARQL. A secondary objective was to validate our motivating assumption that the different transformation strategies lead to different time performance characteristics.

The database was populated with synthetic instances, including: 50k organizations (30.6k hospitals and 19.4k primary schools); 200k persons (about 45% Brazilian citizens, 45% Italian citizens and 10% with double nationality, over 161k adults, 38k children, 129k employees); about 252k supply contracts; 170k employments (30% of the employees with more than one employment) and 61k enrollments (40% of the children with more than one employment). The database size was 91.34 MB for the *one table per concrete class* strategy and 77.22 MB for the *one table per kind* strategy, in MySql 8.0.23. Measurements were obtained in a Windows 10 notebook with an i3 1.8 GHz processor, 250 GB SSD and 8 GB RAM.

The following queries were written to retrieve: (1) the credit rating of each customer; (2) the name of each child, along with the playground size of the schools in which the child is enrolled; (3) the names of Brazilian citizens working in hospitals with Italian customers; this query reveals also the names of these customers and the contract values with the hospital; (4) all data of organizations regardless of whether it is registered as a Hospital or Primary School; (5) given the CI of an Italian citizen, the name of the Hospital with which he/she has a contract and the value of that contract.

The queries were designed to capture different query characteristics. Query 1 represents an example of polymorphic query with reference to the abstract class `Customer` and retrieves an attribute defined at that abstract class. Queries 2 and 3 involve navigation through associations in the conceptual model. Query 3 is the most complex one and corresponds to the realizations we have shown earlier in Listings 1 and 2. Its representation in SPARQL is shown in Listing 6. Query 4 is polymorphic with reference to `Organization` and, differently from query 1, retrieves all attributes of organizations, including those defined in subclasses. Query 5 retrieves data of a specific person.

Listing 6. SPARQL query

```
PREFIX : <https://example.com#>
SELECT ?brazilianName ?organizationName ?value ?italianName {
  ?brazilianPerson    a                 :BrazilianCitizen ;
                      :name             ?brazilianName .
  ?employment         a                 :Employment ;
                      :hasEmployee      ?brazilianPerson;
                      :hasOrganization  ?hospital .
  ?hospital           a                 :Hospital .
  ?hospital           a                 :Contractor ;
                      :name             ?organizationName .
  ?contract           a                 :SupplyContract ;
                      :hasContractor    ?hospital ;
                      :hasCustomer      ?personalCustomer ;
                      :contractValue    ?value.
  ?personalCustomer   a                 :PersonalCustomer .
  ?personalCustomer   a                 :ItalianCitizen ;
                      :name             ?italianName .}
```

Table 3 shows the results obtained, comparing the performance of manually written queries with those rewritten from SPARQL by Ontop (version 2.0.3 plugin for Protégé 5.5.0). In order to exclude I/O from the response time, which could mask the difference between the approaches, we have performed a row count for each query effectively "packaging" it into a "select count (1) from (query)". Further, in order to remove any influence from caching and other transient effects, each query was executed in a freshly instantiated instance of the database three times; the values presented are averages of these three measurements. Values in bold represent the best performing alternative.

The results indicate the performance varies as expected for the two transformation strategies, depending on the characteristics of the queries. In the *one table per concrete class* strategy, the performance of the automatically transformed queries is roughly equal to the manual queries (2, 3 and 5), and some overhead is imposed for queries 1 and 4. This overhead is imposed whenever unions are required in polymorphic queries, as Ontop needs to add columns to the select to be able to translate the retrieved data back to instances of classes in the high-level model. In the *one table per kind* strategy, significant overhead is imposed for queries 1, 4 and 5. Upon close inspection of the generated queries, we were able to observe that the automatically rewritten queries include filters which are not strictly necessary and that were not present in the manual queries. For example, in query 1, we assume in the manual query that only adults enter into supply contracts, as imposed in the conceptual model. However, Ontop adds that check to the query, in addition to several `IS NOT NULL` checks. As a result, the Ontop queries are

Table 3. Performance comparison of relational schemas (in seconds).

Query	One Table per Concrete Class		One Table per Kind	
	Manual Query	**Ontop Query**	**Manual Query**	**Ontop Query**
1	**0.286**	0.333 (+16.4%)	0.385	1.906 (+394.63%)
2	7.436	7.523 (+1.3%)	**1.906**	2.245 (+17.8%)
3	121.797	122.943 (+0.9%)	**102.646**	134.318 (+30.9%)
4	0.610	0.625 (+2.5%)	**0.078**	0.094 (+19.6%)
5	**0.166**	0.167 (+0.4%)	0.271	1.526 (+463.8%)

'safer' than the manual ones. Removing the additional check from the rewritten queries significantly reduces the overhead: query 1 reduces to 0.463s (now +20.2% in comparison to the manual query), query 2 to 1.942s (+1.9%), query 3 to 106.573s (+3.8%), query 4 to 0.078s (0.0%) and query 5 to 0.338s (+25.0%). This makes the imposed overhead always lower than 25% of the response time.

5 Related Work

There is a wide variety of proposals for carrying out data access at a high level of abstraction. Some of these rely on *native graph-based representations*, instead of relational databases. These include triplestores such as Stardog, GraphDB, AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, 4Store and OrientDB. A native graph-based solution has the advantage of requiring no mappings for data access. However, they depart from established relational technologies which are key in many production environments and on which we aim to leverage with our approach.

Some other OBDA approaches such as Ultrawrap [20] and D2RQ [3], facilitate the *reverse engineering* of a high-level representation model from relational schemas. Ultrawrap [20] works as a wrapper of a relational database using a SPARQL terminal as an access point. Similarly to Ontop it uses the technique of rewriting SPARQL queries into their SQL equivalent. It includes a tool with heuristic rules to provide a starting point for creating a conceptual model through the reverse engineering of a relational schema. D2RQ [3] also allows access to relational databases through SPARQL queries. It supports automatic and user-assisted modes of operation for the ontology production. Virtuoso [10] also supports the automatic conversion of a relational schema into an ontology through a basic approach. It allows complex manual mappings which can be specified with a specialized syntax. There are also a range of bootstrappers like [6, 15] that perform automatic or semi-automatic mapping between the relational schema and the ontology. However, these bootstrappers assume that the relational schema exists and provide ways to map it into an existing ontology or help to create an ontology.

Differently from these technologies, we have proposed a forward engineering transformation, in which all mapping is automated. Combining both reverse and forward engineering is an interesting theme for further investigation, which could serve to support a conceptual model based reengineering effort.

Calvanese et. al. propose in [4] a two-level framework for ontology-based data access. First, data access from a relational schema to a domain ontology is facilitated with a manually written OBDA mapping. A second ontology-to-ontology mapping—also manually specified—further raises the level of data access to a more abstract reference ontology. An interesting feature of this approach is that, based on the two mappings, a direct mapping from the abstract reference ontology to the relational schema is produced. Such a two-level schema could be combined with our approach to further raise the level of abstraction of data access.

There are also different approaches that aim at supporting forward engineering of relational databases from logical languages typically used for conceptual modeling. These include approach that propose mappings from OCL to SQL, but also approaches that propose mappings from OWL to relational schemas. In the former group, some

of these approaches, e.g., OCL2SQL [7], Incremental OCL constraints checking [17] and OCL_{FO} [12] are restricted to just mapping Boolean expressions, while others such as SQL-PL4OCL [8] and MySQL4OCL [9] are not limited in this way; in the latter group, we have approaches such as [1] and [23], which implement a *one table per class* strategy, thus, mapping each OWL class to a relational table.

Our proposal differs from these approaches as it is applicable to different transformation strategies. In particular, by leveraging on the ontological semantics of OntoUML, it is unique in implementing the *one table per kind* strategy. Moreover, as empirically shown in [22], this language favors the construction of higher-quality conceptual models—quality attributes that can then be transferred *by design* to the produced relational schemas. Finally, unlike [1, 23], our proposal also generates an ontology-based data access mapping for the transformed database schema.

6 Conclusions and Future Work

We propose an approach to automatically forward engineer a data access mapping to accompany a relational schema generated from a source conceptual model. The objective is to allow data representation in relational databases and its access in terms of the conceptual model. Since the approach is defined in terms of the operations of flattening and lifting, it can be applied to various transformation strategies described in the literature. The approach is based on the tracing of the transformation operations applied to classes in the source model. It is implemented as a plugin to Visual Paradigm; it generates a DDL script for relational schemas and corresponding mappings for the Ontop OBDA platform. Ontop uses the generated mappings to translate SPARQL queries to SQL, execute them and translate the results back. Although we adopt OBDA technology, it is only part of our solution. This is because by using solely OBDA, the user would have to produce an ontology and data access mappings manually. Instead, these mappings are generated automatically in our approach, and are a further result of the transformation of the conceptual model.

We present a pilot study of the performance aspects of the approach. We show that the overhead imposed by the generated mappings and Ontop's translation varies for a number of queries, but should be acceptable considering the benefits of high-level data access. Further performance studies contrasting various transformation strategies should be conducted to guide the selection of a strategy for a particular application. In any case, the writing of queries in terms of the conceptual model can be done independently of the selection of a transformation strategy.

We also intend to support operations other than reads (creation, update and deletion of entities), which are currently not supported by Ontop. Our plan is to generate SQL templates that can be used for these operations. We expect performance of update operations to reveal interesting differences between transformation strategies, in particular, as dynamic classification comes into play.

Acknowledgments

This research is partly funded by Brazilian funding agencies CNPq (grants numbers 313687/2020-0, 407235/2017-5) and CAPES (grant number 23038.028816/2016-41).

References

1. Afzal, H., Waqas, M., Naz, T.: OWLMap: Fully automatic mapping of ontology into relational database schema. Int. J. Advanced Computer Science and Applications **7**(11) (2016)
2. Ambler, S.W.: Agile database techniques: effective strategies for the agile software developer. Wiley (2003)
3. Bizer, C., Seaborne, A.: D2RQ-treating non-RDF databases as virtual RDF graphs. In: ISWC 2004 (posters) (2004)
4. Calvanese, D., Kalayci, T.E., Montali, M., Santoso, A., van der Aalst, W.M.P.: Conceptual schema transformation in ontology-based data access. In: European Knowledge Acquisition Workshop. Lecture Notes in Computer Science, vol. 11313, pp. 50–67. Springer (2018)
5. Calvanese, D., et al.: Ontop: Answering SPARQL queries over relational databases. Semantic Web **8**(3), 471–487 (2017). https://doi.org/10.3233/SW-160217
6. de Medeiros, L.F. et al.: MIRROR: automatic R2RML mapping generation from relational databases. In: Proc. 15th ICWE. vol. 9114, pp. 326–343. Springer (2015)
7. Demuth, B., Hußmann, H.: Using UML/OCL constraints for relational database design. In: Proc. UML'99. LNCS, vol. 1723, pp. 598–613. Springer (1999)
8. Egea, M., Dania, C.: SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language. In: Proc. MODELS'17, Austin, TX, USA, Sep.17-22, 2017. p. 54 (2017)
9. Egea, M., Dania, C., Clavel, M.: Mysql4ocl: A stored procedure-based mysql code generator for OCL. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **36** (2010)
10. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Auer, S., Bizer, C., Müller, C., Zhdanova, A.V. (eds.) Proc. CSSW'07, Leipzig, Germany. LNI, vol. P-113. GI (2007)
11. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc. (2002)
12. Franconi, E. at al.: Logic foundations of the OCL modelling language. In: Proc. 14th European Conf. Logics in Artificial Intelligence (JELIA). vol. 8761, pp. 657–664 (2014)
13. Guidoni, G., Almeida, J.P.A., Guizzardi, G.: Transformation of ontology-based conceptual models into relational schemas. In: Proc. ER 2020. pp. 315–330. Springer (2020)
14. Guizzardi, G. et al.: Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: Proc. ER 2018. pp. 136–150. Springer (2018)
15. Jiménez-Ruiz, E. et al.: Bootox: Practical mapping of RDBs to OWL 2. In: Proc. 14th Int. Semantic Web Conf. ISWC 2015 - Part II. vol. 9367, pp. 113–132. Springer (2015)
16. Keller, W.: Mapping objects to tables: A pattern language. In: EuroPLoP 1997: Proc. 2nd European Conf. Pattern Languages of Programs. Siemens Tech. Report 120/SW1/FB (1997)
17. Oriol, X., Teniente, E.: Incremental checking of OCL constraints through SQL queries. In: Proc. MODELS 2014). CEUR Workshop Proceedings, vol. 1285 (2014)
18. Philippi, S.: Model driven generation and testing of object-relational mappings. Journal of Systems and Software **77**, 193–207 (2005)
19. Poggi, A., Lembo, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Linking data to ontologies. J. Data Semantics **10**, 133–173 (2008)
20. Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL execution on relational data. J. Web Semant. **22**, 19–39 (2013)
21. Torres, A. et al.: Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. Information & Software Technology **82** (2017)
22. Verdonck, M. et al.: Comparing traditional conceptual modeling with ontology-driven conceptual modeling: An empirical study. Information Systems **81**, 92–103 (2019)
23. Vyšniauskas, E. et al.: Reversible lossless transformation from owl 2 ontologies into relational databases. Information technology and control **40**(4), 293–306 (2011)