Gustavo Ludovico Guidoni

Transforming Ontology-Based Conceptual Models into Relational Schemas

Vitória, ES

2023

Gustavo Ludovico Guidoni

Transforming Ontology-Based Conceptual Models into Relational Schemas

Tese de Doutorado submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Doutor em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Programa de Pós-Graduação em Informática

Supervisor: Prof. Dr. João Paulo Andrade Almeida

Vitória, ES

2023

Ficha catalográfica disponibilizada pelo Sistema Integrado de Bibliotecas - SIBI/UFES e elaborada pelo autor

---

G948t

Guidoni, Gustavo Ludovico, 1977-
Transforming Ontology-Based Conceptual Models into Relational Schemas / Gustavo Ludovico Guidoni. - 2023.
154 f. : il.

Orientador: João Paulo Andrade Almeida.
Tese (Doutorado em Ciência da Computação) - Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Análise de sistemas. 2. Banco de Dados. 3. Ontologia. 4. Modelagem Conceitual. I. Almeida, João Paulo Andrade. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 004

Transforming Ontology-Based Conceptual Models into Relational Schemas

Gustavo Ludovico Guidoni

Tese de Doutorado submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Aprovada em 29 de março de 2023.

Prof. Dr. João Paulo Andrade Almeida
Orientador, participação remota

Prof. Dra. Monalessa Perini Barcellos
Membro Interno, participação remota

Prof. Dr. Vítor Estêvão Silva Souza
Membro Interno, participação remota

Prof. Dra. Maria Luiza Machado Campos
Membro Externo, participação remota

Prof. Dra. Fernanda Araújo Baião
Membro Externo, participação remota

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
Vitória/ES, 29 de março de 2023

Documento assinado eletronicamente nos moldes do art. 10 da MP 2200/01 e Lei 14063/20
[Hash SHA256] ee130e1169cd045f35482f71e40e4e54f9bb17e5774661823673a9153b06649e

Datas e horários baseados em Brasília, Brasil
Sincronizado com o NTP.br e Observatório Nacional (ON) em 03/04/2023 às 12:00:42

Folha de Aprovação - Gustavo L. Guidoni

Data e Hora de Criação: 29/03/2023 às 17:50:39

Documentos que originaram esse envelope:
- Folha de Aprovac¸a~o.pdf (Arquivo PDF) - 1 página(s)

Hashs únicas referente à esse envelope de documentos

[SHA256]: ee130e1169cd045f35482f71e40e4e54f9bb17e5774661823673a9153b06649e
[SHA512]: 17c52c7295a77c5fc0512fa71fbc2e4ea170d6f48baa49bdb740e4e1501fd1c992bd8a66ef4d23a1d9f51eab8114a3186d82c7ca5b322f34294a9e28d649350f

Lista de assinaturas solicitadas e associadas à esse envelope

ASSINADO - Fernanda Araújo Baião (fbaiao@puc-rio.br)
Data/Hora: 29/03/2023 - 19:20:35, IP: 139.82.8.110
[SHA256]: 7a372bf7a007202f70526181456dcb6bd85f8847f75364e80fb017693ba044a0

ASSINADO - Maria Luiza Machado Campos (mluiza@ppgi.ufrj.br)
Data/Hora: 01/04/2023 - 16:53:36, IP: 179.67.156.139, Geolocalização: [-23.003296, -43.369576]
[SHA256]: 3221fdc85c8bb9b47957a330ca2fb9c395128aaac56ac1a468d46e4840d62ece

ASSINADO - Monalessa Perini Barcellos (monalessa@inf.ufes.br)
Data/Hora: 03/04/2023 - 10:52:32, IP: 179.217.8.72
[SHA256]: 34dd7f990f24dd77dfc9ed01044dfa2e435bac0ee77cf33ad9bbdf4e618e96a5

ASSINADO - Vítor Estêvão Silva Souza (vitor.souza@ufes.br)
Data/Hora: 03/04/2023 - 11:01:14, IP: 187.36.172.106
[SHA256]: 2e15d633398982aa652b9b7fafcfbbd4b0cf8a0abd27f0bfa1f4647a17663ee9

ASSINADO - Joao Paulo Andrade Almeida (joao.p.almeida@ufes.br)
Data/Hora: 03/04/2023 - 12:00:42, IP: 179.105.121.105, Geolocalização: [-20.257826, -40.271693]
[SHA256]: f3455a6194f42cc82b9c352a12a8ea7f41d2aeccbef38c81f28e97d8c1689302

Histórico de eventos registrados neste envelope

03/04/2023 12:00:42 - Envelope finalizado por joao.p.almeida@ufes.br, IP 179.105.121.105
03/04/2023 12:00:42 - Assinatura realizada por joao.p.almeida@ufes.br, IP 179.105.121.105
03/04/2023 11:01:14 - Assinatura realizada por vitor.souza@ufes.br, IP 187.36.172.106
03/04/2023 11:01:02 - Envelope visualizado por vitor.souza@ufes.br, IP 187.36.172.106
03/04/2023 10:52:32 - Assinatura realizada por monalessa@inf.ufes.br, IP 179.217.8.72
01/04/2023 16:53:36 - Assinatura realizada por mluiza@ppgi.ufrj.br, IP 179.67.156.139
01/04/2023 16:53:05 - Envelope visualizado por mluiza@ppgi.ufrj.br, IP 179.67.156.139
29/03/2023 19:20:35 - Assinatura realizada por fbaiao@puc-rio.br, IP 139.82.8.110
29/03/2023 19:20:28 - Envelope visualizado por fbaiao@puc-rio.br, IP 139.82.8.110
29/03/2023 17:54:08 - Envelope registrado na Blockchain por joao.p.almeida@ufes.br, IP 179.105.121.105
29/03/2023 17:54:07 - Envelope encaminhado para assinaturas por joao.p.almeida@ufes.br, IP 179.105.121.105
29/03/2023 17:50:40 - Envelope criado por joao.p.almeida@ufes.br, IP 179.105.121.105

Documento em conformidade com o padrão de assinatura digital ICP-Brasil e validado de acordo com o Instituto Nacional de Tecnologia da Informação

Os registros de assinatura presentes nesse documento pertencem única e exclusivamente a esse envelope.
Documento final gerado e certificado por **Universidade Federal do Espírito Santo**

À minha esposa Amanda, às minhas filhas Lara e Luísa, aos meus pais Severino e Marlene e ao meu amigo Fabrício Roulin Bittencourt, por acreditar em mim antes mesmo que eu.

Agradecimentos

Agradeço primeiramente a Deus por estar comigo nos caminhos que escolhi. Quero fazer um agradecimento especial ao meu orientador, Prof. João Paulo A. Almeida, pela paciência, dedicação e ensinamentos durante esta jornada. Agradeço aos meus colegas do Nemo, com os quais tive momentos inspiradores que me fizeram “abrir a mente” e enxergar problemas através de outras perspectivas. Agradeço ainda aos professores membros do Nemo, em especial ao professor Falbo (*in memoriam*) pela serenidade de me colocar nos trilhos da ontologia. Agradeço ao professor Giancarlo Guizzardi pelas contribuições na elaboração dos artigos. Agradeço aos membros da banca desta tese pelas importantes contribuições. Agradeço ainda ao Instituto Federal do Espírito Santo (Ifes), em especial aos meus colegas da Coordenadoria de Informática, por terem permitido que eu me afastasse das minhas atividades para me dedicar exclusivamente ao curso de doutorado. Em especial, agradeço à minha esposa pela paciência, carinho e incentivo. Por fim, e não menos importante, quero agradecer ao meu irmão por ser a luz nos momentos mais escuros desta jornada.

“Be stronger than your best excuse.”
(Unidentified author)

Resumo

Apesar das contribuições relevantes da modelagem conceitual baseada em ontologias e do amplo uso de esquemas relacionais na implementação de bancos de dados, a combinação da modelagem conceitual baseada em ontologias e dos esquemas relacionais ainda não recebeu a devida atenção. Dentre as tecnologias de modelagem conceitual, a OntoUML se destaca como linguagem para descrever o domínio do problema, tendo como nicho principal a formulação e propagação do conhecimento. Portanto, o modelo produzido pela OntoUML pode ser visto como um "ponto de partida" para outros artefatos, como o esquema relacional. Para produzir o esquema relacional a partir do modelo conceitual de maneira automatizada, é necessário compatibilizar uma série de construtos. A literatura atual disponibiliza algumas abordagens de transformação objeto-relacional que poderiam, em princípio, ser aplicadas a modelos conceituais orientados a ontologias, como aqueles produzidos em OntoUML. No entanto, há construtos importantes que não são cobertos por tais abordagens. A maioria das abordagens de transformação objeto-relacional não dá suporte a modelos conceituais que: (i) incluem generalizações ortogonais (*overlapping*) ou incompletas (*incomplete*); (ii) adotam classificação dinâmica; ou (iii) empregam herança múltipla. Isso ocorre porque muitas das abordagens discutidas na literatura assumem primitivas subjacentes às linguagens de programação orientadas a objetos (em vez da linguagens de modelagem conceitual). Para resolver essa lacuna, este trabalho visa compreender as forças que regem as estratégias clássicas de transformação das hierarquias de classes em esquemas relacionais, identificando meta-propriedades ontológicas que caracterizam as classes nesses modelos (como sortalidade e rigidez). As informações obtidas são utilizadas para orientar a transformação do modelo conceitual no esquema relacional a fim de evitar alguns problemas das abordagens existentes. Além de automatizar a geração do esquema relacional, também propusemos um mapeamento automatizado de acesso a dados baseado em ontologia para o esquema relacional resultante, com o objetivo de expor os dados em termos do modelo conceitual original, permitindo a produção de consultas em um alto nível de abstração (em SPARQL), independentemente da estratégia de transformação selecionada. Além disso, incorporamos restrições adicionais ao longo do processo de transformação objeto-relacional (implementadas através de *triggers*) para garantir que seja respeitada a semântica do modelo original. A abordagem proposta é contrastada com as abordagens de transformação dominantes na literatura a partir das perspectivas: (i) das primitivas de modelagem conceitual suportadas; (ii) do tamanho do esquema resultante; (iii) do desempenho quanto ao tempo de resposta em consultas; e (iv) da usabilidade do esquema resultante, para o qual é relatado um estudo empírico.

Palavras-chaves: mapeamento objeto-relacional, transformação, modelagem conceitual, modelos, ontologia, modelagem conceitual baseada em ontologia, preservação semântica.

Abstract

Despite the relevant contributions of ontology-based conceptual modeling and the widespread use of relational schemas, the combination of these two technologies has not yet received due attention. Among the conceptual modeling technologies, OntoUML stands out as a language to describe a domain of interest, having as its main niche the formulation and propagation of knowledge. Conceptual models produced with OntoUML can be seen as a "starting point" for other artifacts, such as relational schemas in a database realization. To produce a relational schema from the conceptual model in an automated way, it is necessary to bridge the gap between a series of constructs. The current literature provides some object-relational transformation approaches that could, in principle, be applied to ontology-driven conceptual models, such as those produced in OntoUML. However, there are important constructs that are not covered by such approaches that must be addressed. Most of the existing object-relational transformation approaches fail to support conceptual models that: (i) include overlapping or incomplete generalizations; (ii) support dynamic classification; (iii) have multiple inheritance; and (iv) have orthogonal hierarchies. This is because many of the approaches discussed in the literature assume primitives underlying object-oriented programming languages (instead of conceptual modeling languages). To solve this gap, this work aims to understand the forces that govern classical strategies for transforming class hierarchies into relational schemas, while raising some ontological meta-properties that characterize the classes in these models (like sortality and rigidity). The information obtained is used to guide the transformation of the conceptual model into a relational schema in order to avoid some problems in existing approaches, leading to the novel *one table per kind* strategy. In addition to automating relational schema generation, we also propose an automated ontology-based data access mapping for the resulting relational schema, in order to provide access data in terms of the original conceptual model, and hence queries can be written at a high level of abstraction (in SPARQL), independently of the transformation strategy selected. Further, we forward engineer additional constraints along with the transformed schema (ultimately implemented as triggers) to guarantee that the semantics of the source model is respected. The proposed approach is contrasted with dominant transformation approaches in the literature from the perspectives of: (i) the supported conceptual modeling primitives; (ii) size of the resulting schema; (iii) query answering performance; and (iv) usability of the resulting schema, for which an empirical study is reported.

Keywords: object-relational mapping, transformation, conceptual modeling, models, ontology, ontology-based conceptual modeling, semantic preservation.

List of Figures

List of Tables

List of abbreviations and acronyms

API	Application Programming Interface
DDL	Data Definition Language
ER	Entity-Relationship
EER	Extended Entity-Relationship
HQL	Hibernate Query Language
MDD	Model-Driven Development
OBDA	Ontology-Based Data Access
OCL	Object Constraint Language
ODCM	Ontology-Driven Conceptual Modeling
OOC-O	Object-Oriented Code Ontology
ORM	Object-Relational Mapping
OWL	Web Ontology Language
R2RML	RDB to RDF Mapping Language
RDB	Relational Database
RDF	Resource Description Framework
SPARQL	SPARQL Protocol And RDF Query Language
SQL	Structured Query Language
UFO	Unified Foundational Ontology
UML	Unified Modeling Language
VKG	Virtual Knowledge Graph

Contents

1 Introduction

This chapter presents an overview of this document and defines the basis for the subsequent chapters. It discusses the context in which this thesis is embedded, the motivation for conducting this work, its main challenges, objectives, and the methodological aspects that have guided the research. Finally, it presents the structure of this document.

1.1 Context

Conceptual modeling is responsible for elucidating and representing the problem domain (GRECA; MOREIRA, 2000). Conceptual models are thus produced with the purpose of ruling out ambiguous interpretations from the modeled domain, making them a key artifact for the propagation of knowledge. Various kinds of structural conceptual models have emerged with this purpose and have been consolidated in the Software Engineering and Knowledge Representation disciplines. Noteworthy examples include the Entity-Relationship (ER) diagrams (CHEN, 1976) and the Unified Modeling Language (UML) class diagrams (BOOCH; RUMBAUGH; JACOBSON, 1999).

Structural conceptual models play an important role in the design of relational databases, and are often used to guide the definition of relational schemas. Several systematic model transformation approaches to this end have been explored in the academic literature and incorporated in production-ready tools (TORRES et al., 2017). In this context, Model-Driven Development (MDD) and model transformation techniques (BRAMBILLA; CABOT; WIMMER, 2012; CZARNECKI; EISENECKER, 2000; CORRADINI et al., 1997; SENDALL; KOZACZYNSKI, 2003) allow the reuse of design decisions that are incorporated into automated transformations to alleviate designers from manual (error-prone) realization steps. In these approaches, elements and patterns of a resulting relational schema have their origin traced back to corresponding elements and patterns of a source (high-level) conceptual model.

A significant challenge of a model transformation approach is to preserve the semantics of a source model while obtaining a viable realization of it. This is because, often, source and target models are based on different paradigms, employ different concepts, which results in a variety of technical mismatches. A manifest example of this is the so-called *Object-Relational Impedance Mismatch* (IRELAND; BOWERS, 2015), which results from a "semantic gap" with object-oriented constructs (such as those underlying the UML) not bearing a direct correspondence with constructs in relational schemas. For one, relational schemas offer no counterpart to class specialization hierarchies found in the object-oriented world, and thus offer no special support for classification and polymorphism.

This semantic gap is even wider in the case of conceptual models built at a higher level of abstraction, as is the case of ontology-driven conceptual models (GUIZZARDI, 2005). Ontology-based conceptual models aim to represent the viewpoint of the user's problem domain solely. For this, they must embody abstractions that are based on human cognition and common sense. A suitable conceptual modeling language should provide a solid theoretical basis for problem domain representation, aiming at reducing misunderstandings and identifying potential inconsistencies (CARVALHO, 2016).

With this purpose in mind, Guizzardi (2005) proposed a *foundational ontology* establishing general ontological foundations for conceptual modeling, dubbed Unified Foundational Ontology (UFO). UFO defines a set of domain-independent categories intended to enable the representation of what sorts of things exist in the real world and what are their relationships to each other (GUIZZARDI et al., 2004). It has been successfully employed in several domains in the industry (MOREIRA et al., 2014; GONÇALVES; GUIZZARDI; FILHO, 2007; GUIZZARDI et al., 2015). An important application of UFO is the revision of a fragment of the UML class diagram called OntoUML. OntoUML emerged from a UML class diagram profile to become a language for structural conceptual modeling based on well-founded UFO concepts (GUIZZARDI et al., 2015), in other words, an ontology-driven conceptual modeling language. In OntoUML, the elements of the UML class diagram are categorized in order to gain a specialized semantics following UFO.

This search for a faithful representation of reality has a long history (CHEN, 1976; KENT; HOBERMAN, 2012), and as a result of this search, we can observe the emergence of models with a variety of constructs to represent key details of the problem domain, including object-oriented models (MARTIN; ODELL, 1994). On the other hand, the conceptualization of mainstream data storage implementations still follows predominantly the (ontologically-neutral) relational model (CODD, 1970). Accordingly, the object-relational transformations in the literature (FOWLER, 2002; TORRES et al., 2017; KELLER, 1997; AMBLER, 2003; YODER; JOHNSON, 2002) have not yet evolved to take full benefit of the constructs of ontology-driven conceptual modeling. This thesis addresses this gap in the literature and identifies opportunities to generate better relational schemas taking into account the ontological semantics of the source conceptual model.

1.2 Research Problem

Despite the existence of several strategies for transforming object-oriented models into relational schemas, there are a number of features of ontology-based conceptual models (such as those produced in OntoUML) that have not yet been taken into full account in the existing literature. Most approaches fail to support conceptual models that: (i) include sets of overlapping or incomplete generalizations; (ii) support dynamic classification; (iii) have multiple inheritance;

and (iv) have orthogonal hierarchies. This is because many of the approaches discussed in the literature assume primitives underlying object-oriented programming languages (instead of conceptual modeling languages). Hence, a key challenge is to propose an approach that supports such primitives. Ideally, this can lead to a well-founded transformation process that can be automated and thus capture tried-and-tested design decisions (ALMEIDA; IACOB; ECK, 2007), improving productivity and the quality of the resulting schemas, with the elimination of a manual and error-prone schema design process, which is a typical benefit of a model-driven approach (OMG, 2014).

An additional challenge in transformation-based approaches is that users interact with the results of the transformation, and hence, the usability of such results is important (in this case the usability of the resulting relational schema). 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.

An automated transformation process must deal with mismatches between primitives of the conceptual modeling language and those of the target technical space; in this case, the relational model. Because of this, some of the information that was embodied in the conceptual model may be lost in the transformation process. In the case of the transformation of class inheritance hierarchy, depending on the class-to-table transformation strategy, invariants that are expressed directly in the conceptual model through cardinality constraints and association end typing may be weakened. For example, consider a hierarchy with a class `Person` at the top and a single subclass `Student` with a mandatory attribute `enrollmentNumber`. In the *one table per hierarchy* approach (KELLER, 1997), mandatory attributes of subclasses (in this case `enrollmentNumber`) are implemented as *nullable* columns of the table representing the whole hierarchy (in this case of the table representing all persons). This is required in this approach, because not all persons have enrollment numbers—albeit all students do. This kind of cardinality constraint present in the source model is no longer enforced by the target relational schema, which admits a non-student person to be assigned an `enrollmentNumber` and a student to have a null `enrollmentNumber`. Foreign keys in other tables that should identify only students (for example, concerning the borrower of a book in the university library) now identify persons indistinctly. A transformation approach must address this challenge to avoid losing knowledge embedded in the source conceptual model.

Finally, the literature on transforming object-oriented models into relational schemas addresses quality aspects of the resulting relational schema, such as time performance and understandability. However, many of the existing works fail to addresses these qualities systematically. For example, Philippi (2005) assumes that understandability is a qualitative aspect which also depends on the individual background of a developer, which cannot be measured objectively. The comments presented by the author are based on his experience, as he does not present systematic data collection to as rationale to substantiate the adopted design decisions. The same discussion based on the authors' experience can be seen in (KELLER, 1997; AMBLER, 1997; AMBLER, 2003) (the authors do not indicate how data were obtained). Ideally, adoption of a particular transformation strategy should be backed up by evidence of its quality.

1.3 Research Hypothesis

Interest in the use of ontologies to produce conceptual models of problem domains has been growing since the late 1980s (GUIZZARDI et al., 2015). In the early 2000s, the term ontology gained popularity with the emergence of the Semantic Web. This is because ontology models provides a solid theoretical basis for defining domain elements and their relationships, restricting their interpretations and, consequently, providing more accurate models.

In the last decades, it has become clear that ontological principles and corresponding constructs in methods such as OntoClean (GUARINO; WELTY, 2004) and foundational ontologies such as UFO (GUIZZARDI, 2005; GUIZZARDI et al., 2015) can support the production of high-quality conceptual models. We believe that *some of these principles and constructs can be used to guide decision making in object-relational transformation process* (hypothesis 1). For example, identifying that some classes are 'anti-rigid' (GUARINO; WELTY, 2004) can lead to special treatment due to dynamic classification.

In addition, the literature on transformation of conceptual models into relational schemas has a number of approaches with varying quality. The transformation processes found in the literature lead to relational schemas with distinct quality characteristics. For example, according to Philippi (2005), the *one table per class* strategy results in an easy to understand relational schema, while the *one flattened table per class* strategy has better performance in data extraction. We believe that *principles and constructs of ontology-driven conceptual modeling can be explored to further improve the quality of resulting relational schemas, with suitable performance and better understandability* (hypothesis 2).

Last, the popularization of the Semantic Web led to the emergence of several tools, such as databases ('triple stores' (DING et al., 2007)) and specific query languages (such as SPARQL (PÉREZ; ARENAS; GUTIERREZ, 2009)). These tools explore the computational facet of ontology implementations and bring some ontology features to the level of database realization. Since data storage in the Semantic Web differs from the dominant relational model that is adopted in most commercial applications, several proposals for integration between these technologies have also emerged, under the banner of Ontology-Based Data Access (ODBA) (POGGI et al., 2008; BAGOSI et al., 2014). ODBA requires a modeler to write mappings from the relational schema to the ontology that is used to consume the data, which can be a difficult task. We believe that a *transformation approach can facilitate the OBDA process by automating* *the generation of the required mappings, and hence easily support data access at a high-level of abstraction, i.e., in terms of the domain ontology* (hypothesis 3).

1.4 Objectives

The general objective of this work is to enable the transformation of ontology-driven conceptual models into relational schemas, based on the distinctions provided by a foundational ontology. In this way, we hope to address the gap left by existing object-relational transformation strategies. We approach this objective from the foundational ontology perspective, which allows us to understand the semantics of the various classes present in a conceptual model (called "ontological semantics") as captured in the OntoUML 2.0 language (GUIZZARDI, 2005; GUIZZARDI et al., 2018). In sum, we provide guidelines for what could be called *ontological-relational transformation.*

To achieve the general objective, the following specific objectives have been defined:

1. Propose a new strategy to transform inheritance hierarchies of conceptual models into relational schemas based on ontological semantics of the OntoUML 2.0 language. This new strategy should address the primitives not covered by current strategies for transforming object-oriented models into relational schemas. The strategy should result in relational schemas that offer performance characteristics compatible with other inheritance hierarchy transformation strategies.

2. Create a systematic process to transform conceptual models into relational schemas. This process should operationalize the new transformation strategy.

3. Conduct an empirical study on the understandability of relational schemas produced by the new transformation strategy relative to an existing transformation strategy.

4. Automate the generation of high-level data access support, to allow data access in terms of the ontological model in addition to the usual data access in terms of the resulting schema. Again, high-level data access should offer adequate time performance.

5. Incorporate invariants into the resulting relational schema to ensure that the data stored according to the resulting relational schema respects the semantics of the source conceptual model, i.e., that it does not violate constraints that may be lost in bridging the semantic gap.

1.5 Approach

There are three main classes of approaches to carry out the transformation between the object model and a relational schema (CABIBBO, 2004; PHILIPPI, 2005):

i **Forward engineering** approaches (also called object-relational mappings), in which the relational schema is generated from the class model that must be persisted (often together with the necessary code to propagate object persistence to the database);

ii **Meet-in-the-middle** approaches, in which conceptual model and relational schema are designed, implemented and evolved separately, requiring some middleware to perform the correspondence between the objects and the database.

iii **Reverse engineering** approaches (also called relational-object mappings), in which classes are produced from the existing relational structure;

As our goal is to transform ontology-based conceptual models into corresponding relational schemas, our approach is clearly positioned in the "forward engineering" camp. The literature refers to the connection between the conceptual model and the relational schema as *object-relational mapping* (ORM). However, we will use the term "transformation" to refer to the automated production of the relational schema from the conceptual model, rather than the term "mapping" which could be interpreted as connecting two independently conceived models.

In order to drive the development of a suitable solution to carry out the transformation of the conceptual model into the relational schema, a literature review was conducted of existing object-relational transformations (TORRES et al., 2017), and their main challenges (IRELAND et al., 2009a). With that, it was possible to identify the transformation patterns applied in the inheritance hierarchy, as well as identifying the UML class diagram constructs not covered by classical ORM strategies and some of their inadequacies when applied to certain types of hierarchies, such as those that allow multiple inheritance.

Many of the primitives not covered by current ORM strategies are widely used by ontology-based conceptual languages such as OntoUML. Among many ontological languages, OntoUML has great acceptance in the academic and corporate world (GUIZZARDI et al., 2015), and is recognized as being fast learning (VERDONCK, 2018). So, we adopted the concepts provided by UFO (GUIZZARDI, 2005) and the latest taxonomy (version 2.0) of OntoUML (GUIZZARDI et al., 2021a), to guide the analysis performed. This study culminated in the proposal of a novel transformation strategy called "one table per kind" as a main contribution of this work (related to specific objective 1). The novel strategy is fully implemented into an OntoUML tool (specific objective 2) and tested extensively with models in the OntoUML repository.

We have conducted an empirical experiment involving software industry professionals and academics to obtain data related to understanding of the relational schema produced from the conceptual model. This experiment consists in providing two relational schemas resulting from the same conceptual model, but generated by different transformation strategies. We asked participants to answer questions about both models and asked which model was easier to work with. With this, we were able to shed light on the relative quality of the transformation strategies that were applied (to achieve the specific objective 3).

In addition to improving quality and eliminating the manual and error-prone schema design process, the transformation process can be expanded to provide access to the relational schema from the conceptual model, that is, to enable queries to be written in terms of the conceptual model instead of the resulting schema. The benefit of this approach is that data access only requires knowledge of the conceptual model, abstracting the (non-trivial) technical choices embodied in the resulting schema. This approach leverages ODBA technology, which is based on the abstraction of the relational schema through a set of assertions that capture a 'mapping' between the data source and the ontology. We instrument the transformation process with a tracing structure to be able to generate an OBDA mapping that relates the resulting schema to the source conceptual model (achieving the specific objective 4).

All transformation strategies were conceived through only two operations on the hierarchies: (i) *flattening*, when the attributes and associations of the superclasses migrate to their subclasses; (ii) *lifting*, when attributes and associations of subclasses are migrated to their superclasses, removing the subclasses from the hierarchy. In this way, we can identify the original constraints that are no longer in place after the application of these operations and, consequently, include what must be validated in the resulting model to ensure the realization respects the constraints of the conceptual model. Thus, we produce invariants (implemented through database triggers), that prevent inserting or updating data that violates the conceptual modeling restrictions, (achieving the specific objective 5).

1.6 Outline of this Document

This document is structured as follows:

- Chapter 2. Background. This chapter describes various established object-relational transformation strategies found in the literature, considering how they deal with inheritance hierarchies. It also presents the concepts that underlie the OntoUML 2.0 taxonomy and that will be used in the formulation of the transformation strategy. In this way, we present all the necessary concepts that will guide our ontology-based transformation process.
- Chapter 3. A Novel Ontology-Based Transformation Strategy. This chapter presents two primitive transformations operations (*lifting* and *flattening*) to be used in any inheritance hierarchy and sets out the consequences of applying transformations to certain UFO categories, allowing to establish a roadmap for transforming class models into relational schemas. Thus, a new transformation process for inheritance hierarchies is proposed, named *one table per kind*.

- Chapter 4 Evaluation of the Novel Strategy. This chapter discusses the consequences of different transformation strategies on query performance and presents some data to support the feasibility of the *one table per kind strategy* from the perspective of time performance. This chapter also presents the results obtained from empirical research about the understanding of two relational schemas produced by different transformation strategies.

- Chapter 5. High-Level Data Access. This chapter explains how to leverage the transformation process to manage the traces between the original and target model in order to create the OBDA mapping. Through the technology provided by Ontop (CALVANESE et al., 2017), we were able to maintain the conceptual model as an "access point" for data stored in the relational schema. We also compare the performance of automatically-generated queries generated with those written manually in order to show the performance overhead of the proposed approach.

- Chapter 6. Preserving Conceptual Model Semantics. This chapter identifies the constraints that are lost in the transformation process, independently of the class-to-table transformation strategy applied. It discusses the consequences of *flattening* and *lifting* refactoring operations and shows what must be done at each application of these operations to produce the invariants (implemented through database triggers) to guarantee that the constraints of the source conceptual model are respected.

- Chapter 7. Conclusions and Future Work. This chapter summarizes the research contributions and outlines future work.

An overview of structure of this thesis is presented in Figure 1 below.

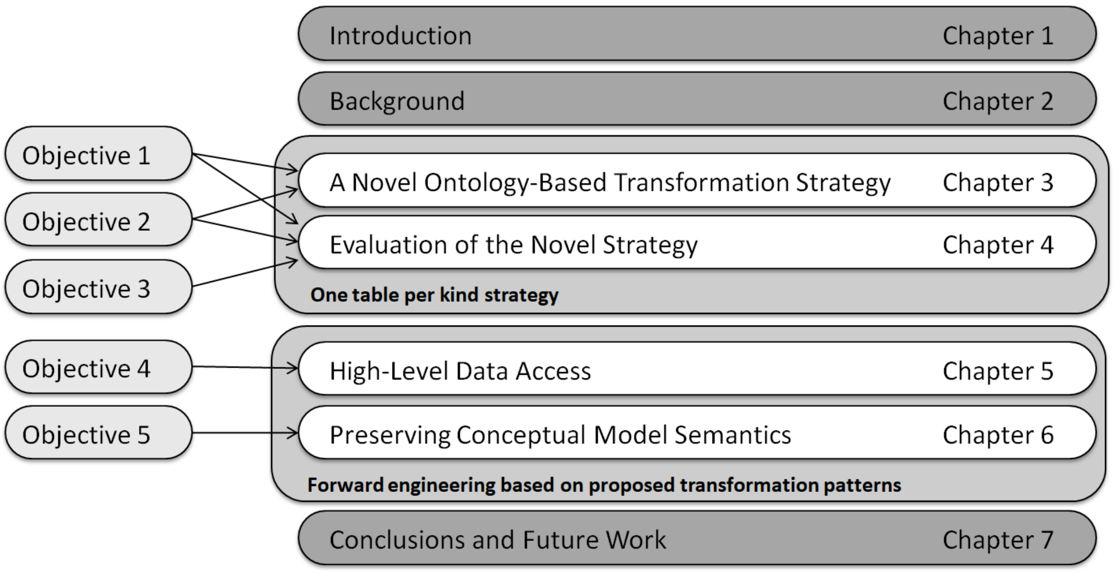

Introduction
Chapter 1
Background
Chapter 2
Objective 1
Objective 2
Objective 3
A Novel Ontology-Based Transformation Strategy
Chapter 3
Evaluation of the Novel Strategy
Chapter 4
One table per kind strategy
Objective 4
Objective 5
High-Level Data Access
Chapter 5
Preserving Conceptual Model Semantics
Chapter 6
Forward engineering based on proposed transformation patterns
Conclusions and Future Work
Chapter 7

Figure 1 – Overview of the thesis structure relating the objectives of this thesis with the chapters in which they are accomplished

2 Background

This chapter describes the key concepts used to substantiate this work. It starts in Section 2.1 by identifying key primitives in a structural (object-oriented) conceptual model and presents the main object-relational transformation strategies found in the literature. The result and consequences of each transformation strategy are presented based on a running example, which is used throughout this work. Section 2.2 presents the additional ontological primitives introduced by the Unified Foundational Ontology (UFO) and incorporated in OntoUML (GUIZZARDI, 2005). These primitives will be used to support the choices of our transformation strategy presented later in Chapter 3.

2.1 From Conceptual Models to Relational Schemas

An object-relational transformation comprises the compatibility of a series of concepts that exist uniquely in each paradigm, such as inheritance and *n*-to-*n* associations, belonging to object-oriented conceptual models, and the occurrence of primary and foreign keys, belonging to relational schemas. The conceptual gap between the two paradigms is addressed in the literature as *Impedance Mismatch* (IRELAND; BOWERS, 2015; IRELAND et al., 2009b; AMBLER, 1997; SMITH; ZDONIK, 1987). Over the years, a number of object-relational transformation strategies have been discussed in the literature, in many cases, assuming different primitives for the source conceptual model. We review here these primitives, present a running example that explores all of them, and discuss the implications for the transformation strategies in the literature. We focus on the strategies used in the transformation of class hierarchies.

2.1.1 Primitives of the Source Conceptual Model

We assume that the basic elements of a taxonomy in a structural conceptual model are *classes* and their relations of *specialization* (also called “is-a”, *subclassing*, or *inheritance* relations). Classes are used to capture common properties of entities they classify, and, in a taxonomic hierarchy, more general classes are specialized into more specific (sub-)classes, which “inherit” attributes and associations of their superclasses (for brevity, we call here both the attributes and associations of a class its “features”). We assume conceptual modeling approaches share these ground notions, nevertheless, there are variations including additional supporting mechanisms, their semantics and their possible range of use, as discussed in the remainder of this section.

Multiple Inheritance. A first source of variation concerns the possibility of a subclass to specialize more than one superclass. In a taxonomic hierarchy with multiple inheritance, a class can be a subclass of different classes (CARDELLI, 1984). A subclass in such a hierarchy inherits the properties of all its superclasses. Multiple inheritance has been avoided in some programming languages as it leads to some implementation difficulties. In conceptual modeling, however, multiple inheritance is hardly dispensable, as it enables opportunities for modularity and reusability (CARRÉ; GEIB, 1990).

Overlapping Classification. Another variation concerns whether an object can simultaneously instantiate multiple classes which are not related by specialization. For example, a person may instantiate both the `BrazilianCitizen` and the `ItalianCitizen` subclasses of `Person`. In UML, this can be explicitly supported with the so-called *overlapping generalization sets*, in which a set of non-disjoint classes specialize the same superclass. Additionally, this kind of scenario can be supported with different (orthogonal) hierarchies that specialize a common superclass based on different criteria. For example, persons may be classified according to their age and according to citizenship status. In this setting, a Brazilian adult would instantiate both the `BrazilianCitizen` and the `Adult` subclasses of `Person` (each from a different generalization set).

Non-Exhaustive Classification. A related variation concerns whether specializing subclasses "cover" the specialized superclass, i.e., whether they jointly exhaust all the classification possibilities for the superclass. In UML, this can be explicitly supported with the so-called *complete* generalization sets, which are opposed to *incomplete* generalization sets. In the case of an incomplete generalization set, it is possible for an instance of the superclass not to instantiate any of the subclasses in the set. For example, it is possible for a person to be stateless (in the sense of not being considered a national by any State), and hence a nationality generalization set could be marked as "incomplete" even in the case all known nationalities are explicitly modeled.

Dynamic Classification. Another variation concerns whether instances can change the set of classes they instantiate throughout their existence. For example, a `Person` may be reclassified from `Child` to `Adult` with the passing of time. This is not possible if static classification is assumed. Many modeling languages support only static classification given their roots in object-oriented programming languages that likewise only support static classification; in these languages, the class that an object instantiates is defined at object instantiation time, and remains fixed throughout that object's life cycle. Nevertheless, in conceptual modeling, dynamic classification has been considered an important feature and studied by several authors (WIERINGA; JONGE; SPRUIT, 1995; STEIMANN, 2000; STEIMANN, 2007; ALBANO et al., 1993; GOTTLOB; SCHREFL; RÖCK, 1996). Dynamic classification enlarges the realm of classes to include those which apply contingently or temporarily to their instances. Examples include the ontological notions of *phases* (such as `Child` and `Adult`), and *roles* (such as `BrazilianCitizen`, `ItalianCitizen`, `Employee` and `Customer`).

Abstract and Concrete Classes. Finally, we assume that the conceptual modeling technique may distinguish between abstract and concrete classes. Abstract classes have no "direct" instances, i.e., all of their instances are also instances of specializing subclasses. Concrete classes in their turn are not bound by this constraint (and thus can have "direct" instances).

2.1.2 Running Example

To discuss and better expose the characteristics of the existing strategies, we propose a *Running Example* in a UML class diagram, depicted in Figure 2. This model addresses some aspects of conceptual models that are under-explored in the ORM literature, including: (i) an overlapping and incomplete generalization set, in which `Persons` are specialized according to—none or more than one—enumerated countries of citizenship; (ii) a generalization set orthogonal to the first one, in which `Persons` are classified dynamically according to life phase; (iii) multiple inheritance, with each `PersonalCustomer` being both a `Customer` and an `Adult`, as well as each `CorporateCustomer` being both a `Customer` and an `Organization`); (iv) orthogonal classification hierarchies (with `Organization` being classified as a `CorporateCustomer` when it establishes a relation with another `Organization` and also possibly being classified as a `PrimarySchool` in which children may be enrolled or as a `Hospital`); (v) an abstract class `NamedEntity`, which is specialized into `Person` and `Organization` and another abstract class `Customer`, which is specialized into `PersonalCustomer` and `CorporateCustomer` concrete classes.

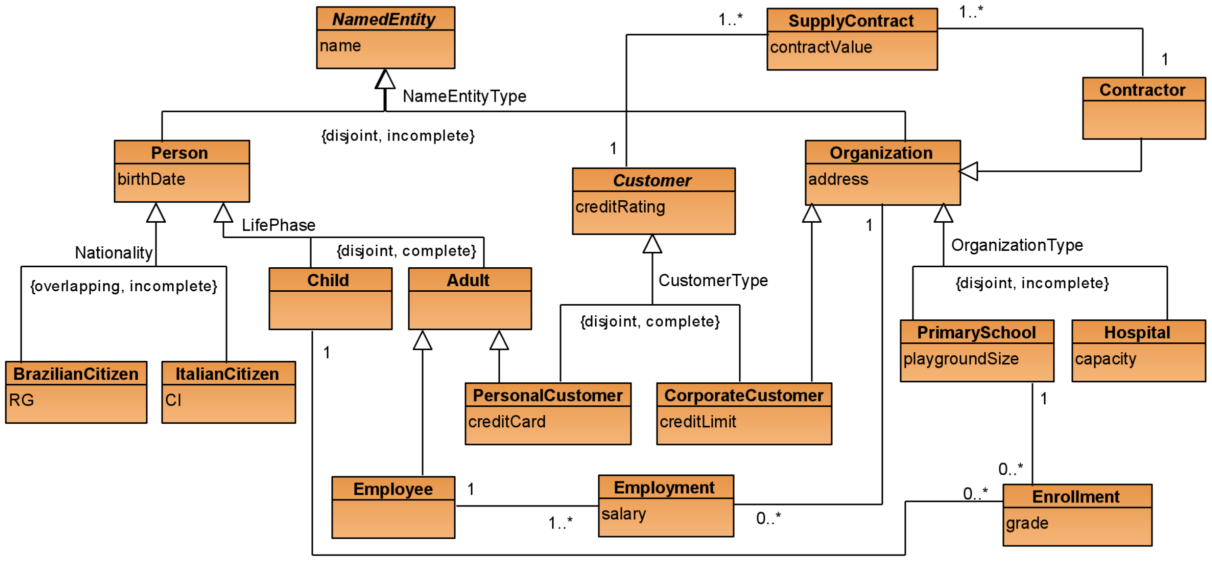


Figure 2 – Running example

In the sequel, we present the following strategies and discuss their application to the running example:

- *One table per class*;

- *One table per concrete class*;
- *One flattened table per concrete class*;
- *One flattened table per leaf class*;
- *One table per hierarchy*;
- *Generic structure* (also called ‘adaptive object model’).

2.1.3 One Table per Class

In the *one table per class* strategy, also called “class table inheritance” (FOWLER, 2002), “vertical inheritance” (TORRES et al., 2017) or “one class one table” (KELLER, 1997), each class gives rise to a separate table, with columns corresponding to the class’s features. Abstract classes and concrete classes are treated alike. Specialization between classes in the conceptual model gives rise to a foreign key in the table that corresponds to the subclass (henceforth “subclass tables” for simplicity). This foreign key references the primary key of the table corresponding to the superclass (henceforth “superclass table” for simplicity). For example, when applying this strategy under the hierarchy formed by the classes `NamedEntity`, `Person`, `BrazilianCitizen` and `ItalianCitizen` of Figure 2, all classes are transformed into tables. Foreign keys in the `BRAZILIAN_CITIZEN` and `ITALIAN_CITIZEN` tables refer to the primary key of the `PERSON` table, which its foreign key references the primary key of `NAMED_ENTITY`. To ensure a `1:1` relationship between the subclass and superclass tables, the foreign keys of subclass tables emulating inheritance are also primary keys. The resulting relational schema of this strategy directly reflects the organization of classes in the conceptual model, and no restriction on the primitives of the model are imposed, as can be seen in Figure 3.

Multiple inheritance can be supported by using a composite foreign key in subclass tables and each foreign key value must be unique across the table. For example, in Figure 3, `PERSONAL_CUSTOMER` has a composite key referencing the primary keys of the `ADULT` and `CUSTOMER` tables and make them unique columns in `PERSONAL_CUSTOMER`. Otherwise, a person’s primary key could be in the `PERSONAL_CUSTOMER` table linked to the primary key of other consumers, generating a design error regarding the conceptual model.

Constraints on a generalization set (overlapping and non-exhaustive classification) are reflected in integrity constraints concerning the cardinality of entries in the subclass tables for a particular row in each superclass table. However, just the relationship of the superclass table to the subclass tables does not guarantee the `disjoint` constraint of the generalization set, that is, the resulting relational schema accepts that the primary key of the `ORGANIZATION` table can be in the `HOSPITAL` and `PRIMARY_SCHOOL` tables at the same time, which is not allowed by the conceptual model.

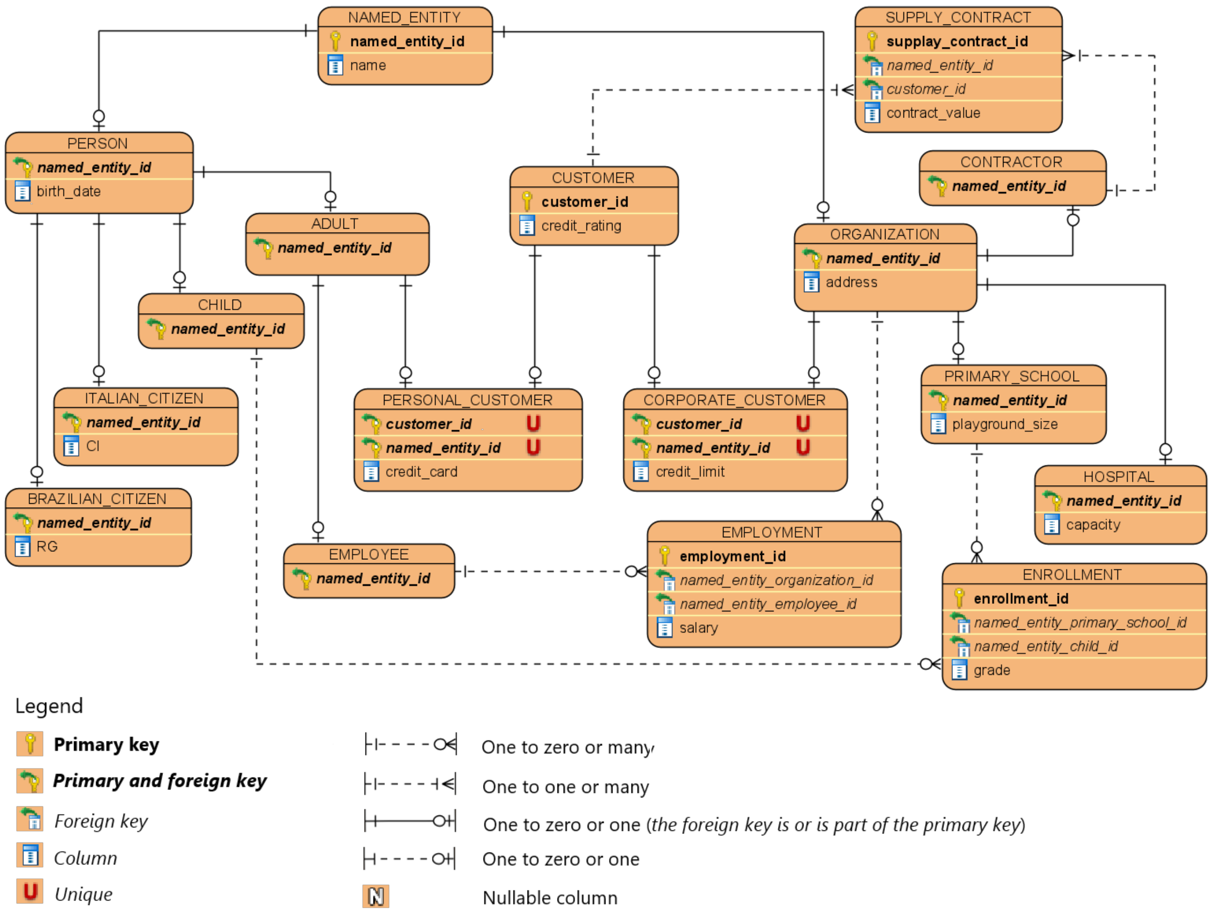


Figure 3 – Running example transformed with the *one table per class* strategy.

Dynamic classification is implemented by deletion of a row in a subclass table (cascaded to further subclass tables) and, possibly, insertion in another, which is cumbersome to implement and may require complex transactions. The main drawback of the *one table per class* strategy is the performance characteristics of the resulting schema. In order to manipulate data concerning 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. For example, consulting the `name` and `credit_card` of a `Person` one needs to traverse four(!) tables, namely `NAMED_ENTITY`, `PERSON`, `ADULT` and `PERSONAL_CUSTOMER` (and even more tables if we are also interested in a person's nationality).

2.1.4 One Table per Concrete Class

The *one table per concrete class* strategy turns into tables only the concrete classes of the inheritance hierarchy. In this way, the abstract classes are not considered in relational schema generation. and their attributes only migrate to their immediate concrete subclass tables. For example, when applying this strategy under the hierarchy formed by the `NamedEntity` and `Person` classes of Figure 2, the attributes and references of the `NamedEntity` class are flattened only for `Person` class and the `NamedEntity` class is not considered in relational schema generation., as seen in Figure 4.

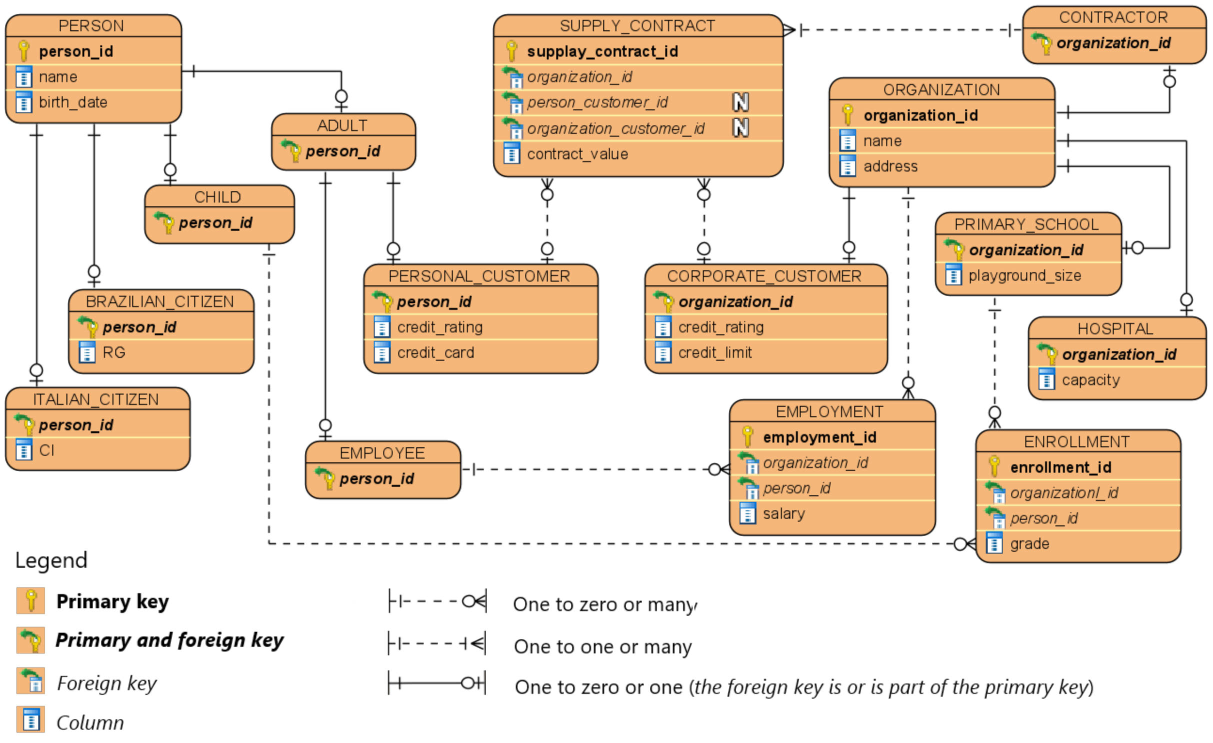


Figure 4 – Running example transformed with the *one table per concrete class* strategy.

One table per concrete class is similar to the *one table per class* strategy, with the same issues concerning multiple inheritance, generalization set constraints and dynamic classification. Although there is some optimization regarding the number of tables in the relational schema, this strategy has the same performance problem discussed for the *one table per class* strategy.

2.1.5 One Flattened Table per Concrete Class

In the *one flattened table per concrete class* strategy, also called "one inheritance path one table" (KELLER, 1997), "concrete-table" (TORRES et al., 2017) or "concrete table inheritance" (FOWLER, 2002), abstract classes are not considered in relational schema generation and their features propagated to subclass tables of the hierarchy. Each subclass table contains the sum of the attributes of all its superclasses. For example, when applying this strategy under the hierarchy formed by the `NamedEntity`, `Person`, `BrazilianCitizen` and `ItalianCitizen` classes of Figure 2, the `NamedEntity` class is not considered in relational schema generation and its attributes are flattened to the `PERSON`, `BRAZILIAN_CITIZEN` and `ITALIAN_CITIZEN` tables, in the same way that `Person` attributes are also propagated to `BRAZILIAN_CITIZEN` and `ITALIAN_CITIZEN` tables, as seen in Figure 5. Since all subclasses have the attributes of superclasses, there is no need for associations between the tables that make up the hierarchy.

It is important to note that the *one table per concrete class* strategy only propagates features from the abstract class to its direct subclass tables, while *one flattened table per concrete*

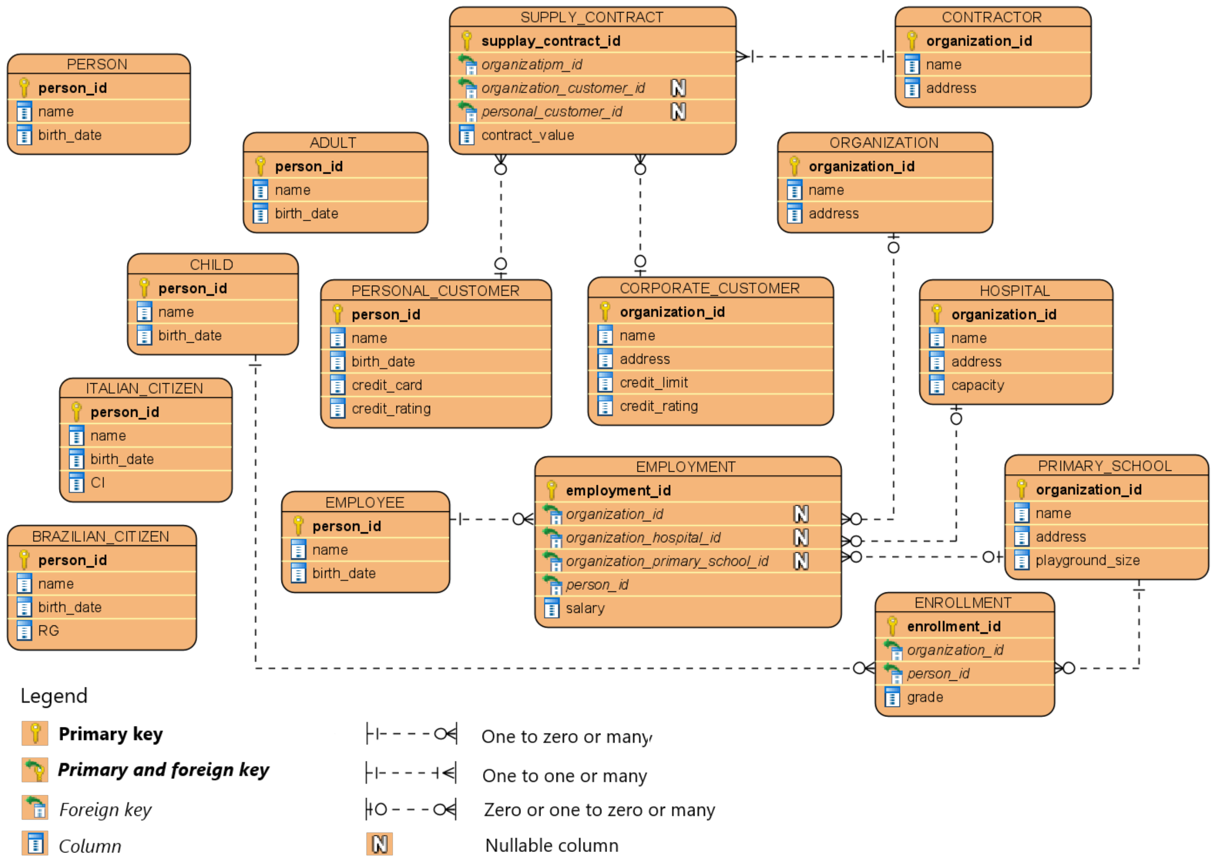


Figure 5 – Running example transformed with the *one flattened table per concrete class* strategy.

class propagates them regardless of whether the superclass is abstract and for all subclass tables of the hierarchy. Another important point is that all attributes belonging to a class are captured in the table that corresponds to it. For example, when storing data for an Italian person, the data is not stored in the `PERSON` table, but only in the `ITALIAN_CITIZEN` table.

One flattened table per concrete class has problems when the generalization set is `overlapping` to identify the instance of the superclass. Consider the "flattening" of `Person` in our example. In case a person has double Brazilian and Italian citizenship, there would be a row in the `BRAZILIAN_CITIZEN` table and another row in the `ITALIAN_CITIZEN` table denoting the same person, but without a correlating identifier. This problem is also present for orthogonal hierarchies of an abstract superclass. (There would be, e.g., a row in the `ADULT` table corresponding to a row in the `BRAZILIAN_CITIZEN` table.) Consequently, it is necessary to maintain a unique way to identify the same person for all subclass tables, otherwise it is impossible to perform polymorphic queries between classes of the hierarchy. Another problem is the duplication of data between tables, aggravated when orthogonal hierarchies are composed of overlapping generalization sets.

This strategy addresses the performance issue discussed for the *one table per class*, at the cost of polymorphic queries. Consider, for example, a query to access the `name` and `birth_date` of all people. In the *one table per class* strategy such a query involves only one table. In this strategy, however, the query requires the union of PERSON, BRAZILIAN_CITIZEN and ITALIAN_CITIZEN (remember, Nationality generalization set is incomplete). The higher the class in the specialization hierarchy, the higher the number of tables involved in a polymorphic query.

Another undesirable feature is the increased number of references to be created in the relational schema when a class has an association with the superclass, generating an association for each class in the hierarchy (if the generalization set is incomplete). As we can see, the association between the Employment and Organization classes (see Figure 2) generates three associations in the relational schema presented in Figure 5: between EMPLOYMENT and ORGANIZATION; between EMPLOYMENT and HOSPITAL; between EMPLOYMENT and PRIMARY_SCHOOL. Referential integrity tends to be problematic if the number of leaf classes is large and when there are orthogonal hierarchies. Consider the case in which there is an association with the Person class. The resulting relational schema would have an association for each subclass table as well as the superclass table, that is, five associations.

One flattened table per concrete class has some similarities when compared to the *one table per concrete class* strategy, since it supports multiple inheritance; has some limitations to implement “disjoint” generalization sets; and has some performance issues when classification is dynamic—as records have to be migrated between tables.

2.1.6 One Flattened Table per Leaf Class

In the *one flattened table per leaf class* strategy, also termed “horizontal inheritance” (TORRES et al., 2017), each of the leaf classes in the hierarchy gives rise to a corresponding table. Attributes of all (non-leaf) superclasses of a leaf class are realized as columns in the “leaf class table”, and all (non-leaf) superclass are not considered in relational schema generation. No foreign keys emulating inheritance are employed in this approach. The strategy can be understood as reiterated application of an operation of “flattening” of superclasses. For example, when applying this strategy under the hierarchy formed by the classes Customer, PersonalCustomer and CorporateCustomer in Figure 2, the credit_rating attribute of the Customer correspond to columns in both the PERSONAL_CUSTOMER and CORPORATE_CUSTOMER tables (which also has columns for attributes of its other superclasses: birth_date and address respectively). Any references to a superclass (e.g., the references realized as foreign keys of a SUPPLY_CONTRACT) now refer to an entry in either of the subclass tables (in this case PERSONAL_CUSTOMER or CORPORATE_CUSTOMER), as seen in Figure 6.

This approach is similar to the *one flattened table per concrete class* strategy, except that the superclass tables are removed. This does not provide an improvement with respect to the conceptual modeling the primitives addressed. This strategy faces an additional problem when the generalizations set is incomplete, because with the deletion of the superclass table, there is no natural way to store just the superclass data in the relational schema. A (rather poor)

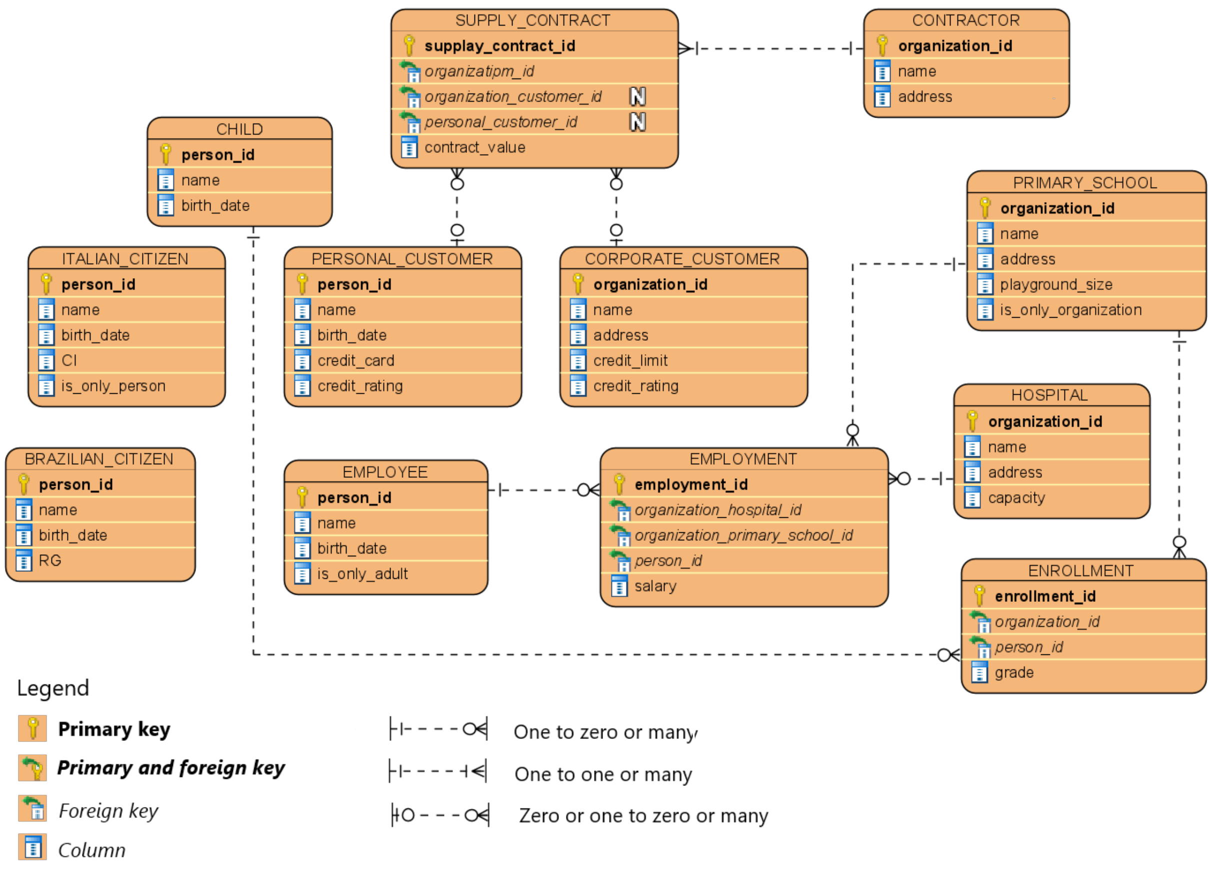


Figure 6 – Running example transformed with the *one flattened table per leaf class* strategy.

workaround is to "choose" a subclass table and provide a means to identify that a given record in this table is an instance of a superclass and not of the subclass (see attribute `is_only_person` in the `ITALIAN_CITIZEN` table to identify a stateless person.)

2.1.7 One Table per Hierarchy

The *one table per hierarchy* strategy, also called "single-table" (FOWLER, 2002) or "one inheritance tree one table" (KELLER, 1997), can be understood as the opposite of *one flattened table per leaf class* strategy, applying a "lifting" operation to subclasses instead of the "flattening" of superclasses. Consider, e.g., the hierarchy formed by `NamedEntity`, `Person`, `BrazilianCitizen` and `ItalianCitizen`, shown in Figure 7(a)[1]. `NamedEntity` is the top-level class in this hierarchy, and will thus give rise to a corresponding `NAMED_ENTITY` table. Attributes of each subclass become columns in the "superclass table", with mandatory attributes corresponding to optional columns. The "lifting"operation is reiterated for each subclass until the top-level class of hierarchy is reached and the subclasses are not considered in relational schema generation. The resulting relational schema is shown in Figure 7(b).

This strategy usually requires the creation of an additional column to distinguish

[1] We only use part of the Running Example because the *one table per hierarchy* strategy does not support multiple inheritance.

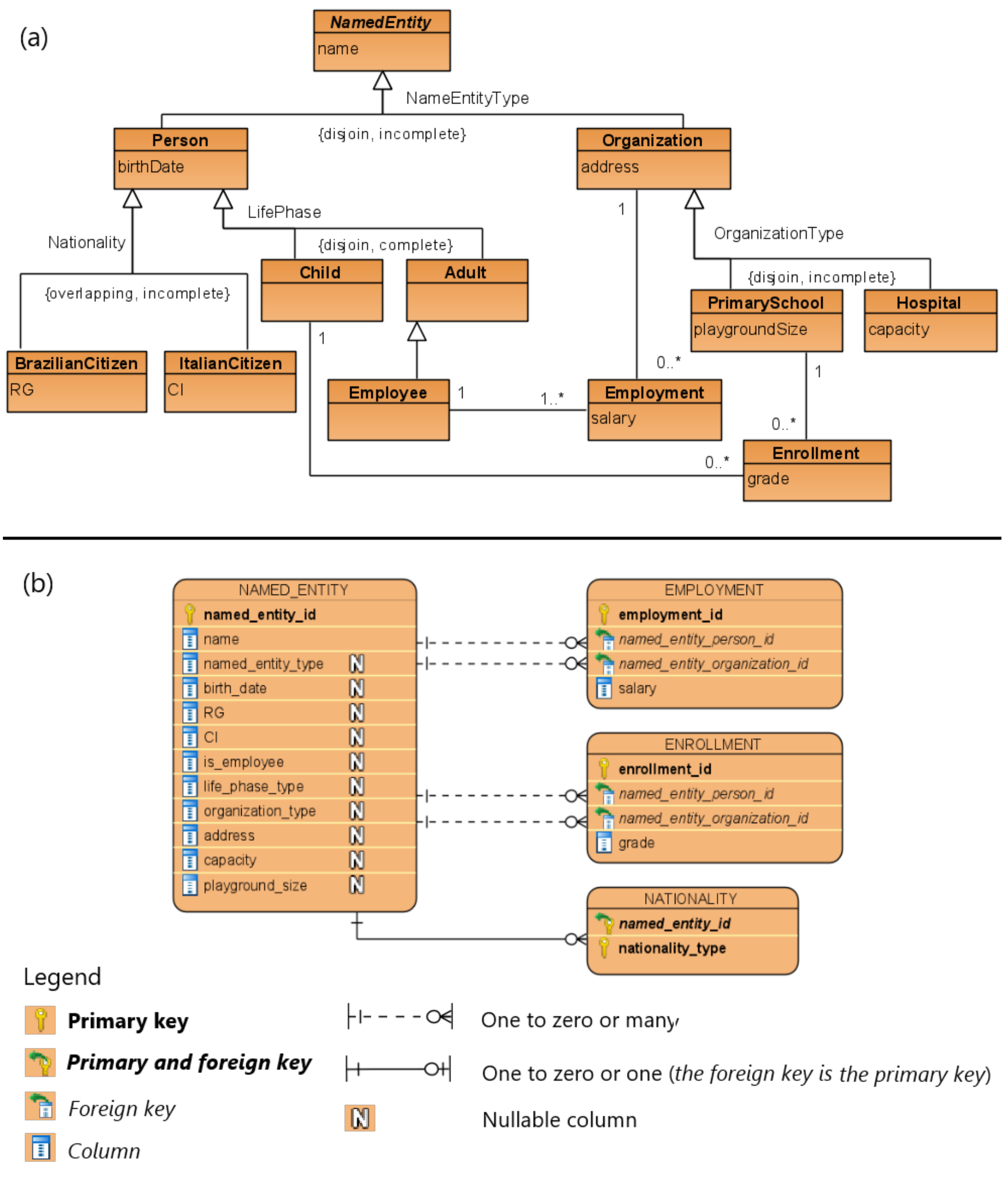


Figure 7 – Running example transformed with the *one table per hierarchy* strategy (excerpt).

which subclass is (or which subclasses are) instantiated by the entity represented in the row (a so-called “discriminator” column). In principle, the performance problems discussed for the other strategies do not appear in this approach. However, as discussed by Torres et al. (2017), standard database integrity mechanisms cannot prevent certain inconsistencies. In our example, the `ENROLLMENT` table would have foreign keys to the top-level class `NAMED_ENTITY`, since `Person` and `PrimarySchool` would be “lifted” to the corresponding top-level class. Thus, the discriminator would have to be checked to make sure that only children are enrolled in primary schools, because the database would admit any named entity enrolled in another named entity, e.g., persons enrolled in hospitals, primary schools enrolled in adults, or even hospitals enrolled in hospitals. In addition, the greater the number of leaf classes in a hierarchy, the greater the number of optional columns that remain unattributed in every row. An additional column to identify the stored subclass may not be a satisfactory solution when there are a large number of subclasses, due to the increased range of accepted values. This problem is exacerbated when the generalization set is overlapping. In our case, we create an identifier for each generalization set; the `NATIONALITY` table was created because the `nationality_type` attribute would be multivalued, since its generalization set is overlapping.

This approach is problematic in the face of multiple inheritance. First of all, one needs to decide to which of the various superclasses a subclass will be "lifted". Consider the case of `Per-sonalCustomer` of Figure 2; should the `credit_card` attribute be lifted to the `NAMED_ENTITY` table or to the `CUSTOMER` table? Second, and more important, if multiple inheritance is admitted, then there may be top-level classes that are not disjoint (e.g., `Customer` and `NamedEntity`). This means that there will be rows in more than one table denoting the same individual, a problem which also appeared in *one flattened table per leaf class*, albeit for different reasons. Dynamic classification at the top of the hierarchy (such as the case of `Customer`) also poses a challenge, not unlike the one faced by *one table per leaf class*.

2.1.8 Generic Structure

This strategy, also called "metadata-driven" (AMBLER, 2003) or "adaptive object model" (YODER; JOHNSON, 2002), is based on storing the model metadata along with the data. For example, consider that classes are stored in a single table called `CLASS` and class attributes are stored in a table called `ATTRIBUTE`. The relationship between these tables identifies which attributes belong to which class, as depicted in Figure 8. Inheritance is modeled as a relationship between classes, just inserting a record in an `INHERITANCE` table relating superclass and subclass. Class attribute values are stored in a single table called `VALUE`, that is, all data manipulated by a domain application are saved only in the `VALUE` table, while the other tables only change when the conceptual model changes. The relational schema shown in Figure 8 is a simplified one; it could be extended to represent associations, aggregations or multivalued attributes, for example.

The advantage of this strategy is that the relational schema is unaffected by any changes in the conceptual model, being recommended when the conceptual model is constantly changing. This approach has some problems: (i) it requires significant effort to write complex queries, as the data is not stored in an intuitive way. For example, access to several rows is required to obtain the data for a single instance; (ii) constraints imposed by the conceptual model, such as nullability of attributes and referential integrity, are lost.

In fact, this strategy cannot be considered a real transformation, because the original

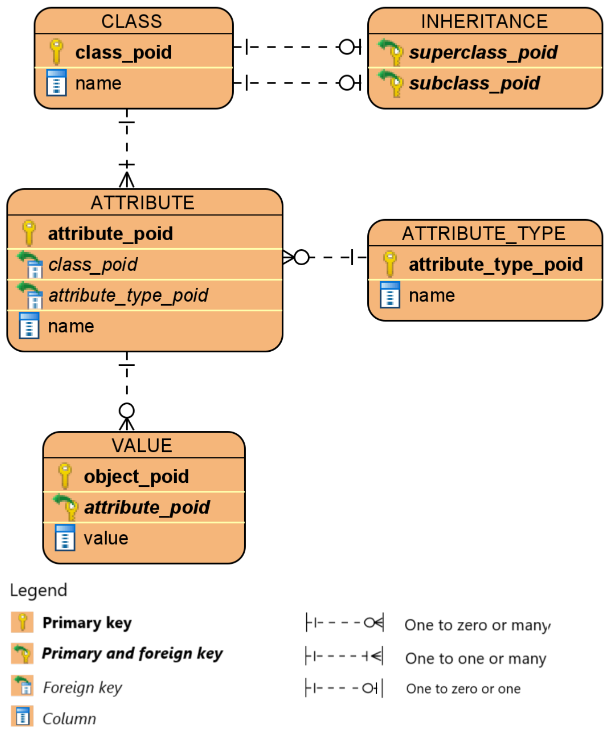


Figure 8 – Relational schema of the *generic structure* strategy.

model does not really need to be modified, that is, it is just another way to represent the model, transformed or not.

2.1.9 Summary

Table 1 summarizes the comparison between the strategies presented. As can be seen, only the *one table per hierarchy* strategy does not support multiple inheritance, since it is the only strategy that 'migrates' features from subclasses to their superclasses. Problems with the implementation of orthogonal hierarchies and overlapping generalization sets occur in the *one flattened table per concrete class* and *one flattened table per leaf class* because the same entity is represented in several tables (unlike the *one table per concrete class* strategy where an entity is obtained by joining several tables). Dynamic classification is better implemented by the *one table per hierarchy* strategy because it does not entail record migration. In the case of *one table per hierarchy*, only a single operation (an update) is required to assign the new entity classification using discriminator columns. All the other strategies require record migration, hence are considered to perform 'poorly' with respect to dynamic classification.

The table also considers some performance characteristics, with parameters used for estimation starting with h to represent the maximum height of the hierarchy (i.e., the maximum path size from a top-level class to a leaf class. $h = 0$ when there is no specialization hierarchy) and n represents the number of subclasses in the hierarchy. The table presents worst-case figures for the retrieval and insertion of an entity (with all its attributes). *One table per class* and *one table per concrete class* fares poorly in this comparison, with h joins required in the worst case, with $h \geq h_c$ (h_c is the maximum height of concrete classes of the hierarchy). The performance of polymorphic queries is considered, with respect to the number of tables involved in a union to read one attribute defined in a superclass. *One flattened table per leaf class* and *one flattened table per concrete class* perform poorly in this respect, where $n_c \geq n_l$ (n_c is the number of concrete subclasses in the hierarchy and n_l is the number of leaf classes in the hierarchy). All others perform equally, requiring only one access to retrieve all attributes of an entity.

The "Number of tables affected in insert operation" for *generic structure* is only one. In fact, this strategy uses only a single table to store all attributes of the conceptual model. However, when analyzing this column from the viewpoint of the "number of operations performed on the relational schema to save an entity", the most suitable result should be "the number of attributes of the entity", since for each attribute one insert operation must be performed. Although this approach seems to fair well when inspecting this summary, as discussed in the previous section, it has problems related to usability, as: (i) high query writing complexity - queries tend to be large, as additional joins must be performed to identify classes and their attributes; (ii) poor query understandability - the joins between the classes are always performed on the same set of tables (differing only by the alias) and by the same attributes, with the understanding worsened due to the size of the query; (iii) poor usability of the result - query result values are displayed in just a single column, that is, data for the same entity end up in different rows; (iv) loss in semantic integrity mechanisms - as the relational schema stores the metadata of the relational model, the referential integrity (which was previously performed by primary and foreign keys) must be guaranteed through additional code; (v) potential problems with indexes - having a single table with a single field to store all attributes of the conceptual model restricts the range of indexes that can be used by the DBMSs to optimize queries. The size of tables can also become a performance hurdle.

Table 1 – Summary of strategies presented.

Realization Strategy	**Multiple inheritance**	**Orthogonal hierarchies**	**Dynamic classification performance**	**Nº of joins to retrieve an entity**	**Nº of tables in union to read one attribute (polymorphic query)**	**Nº of tables affected in insert operation**
One table per class	yes	yes	poor	h	1	$h + 1$
One table per concrete class	yes	yes	poor	h_c	1	$h_c + 1$
One flattened table per concrete class	yes	no	poor	1	n_c	1
One flattened table per leaf class	yes	no	poor	1	n_l	1
One table per hierarchy	no	yes	good	1	1	1
Generic structure	yes	yes	poor	1	1	1

2.2 Ontology-Driven Conceptual Modeling

Thusfar, the source conceptual models that have been considered are regular object-oriented models which can be considered 'ontologically neutral' (GUARINO, 2009). Here, we extend our view on conceptual models to include those that can be characterized as 'Ontology-Driven Conceptual Models' (VERDONCK et al., 2015).

Ontology (written in a capital "O") is a branch of the discipline of Philosophy that deals with the most general features of the "things" (entities) that we want to represent from reality, addressing relations between entities belonging to specific domains of science, such as Physics, Chemistry and Biology and also between entities recognized by common sense (GUIZZARDI, 2007). The term "ontology" has gained popularity since it was introduced in 1967 in the Computer Science discipline (GUIZZARDI, 2005). However, it acquired different meanings in some of its branches (in these cases, ontology is written with a lowercase "o"). The concept of ontology in Conceptual Modeling has the same Philosophy discipline perspective, with the purpose of developing a formal system of categories that can be used in the production of scientific theories or in specific domains of knowledge, regardless of the language used to describe them. In branches of Computer Science, such as Artificial Intelligence, Software Engineering and Semantic Web, the concept of ontology is used in the artifact perspective to formalize and restrict interpretations about a specific problem domain, expressed in a knowledge representation language such as OWL or a conceptual modeling language such as OntoUML. So, in this thesis, we use the term "ontology" in the Computer Science perspective.

The Unified Foundational Ontology (UFO) is a *foundational ontology* developed by Guizzardi (2005) through the combination of Formal Ontology theories of Philosophy, Cognitive Science, Linguistics, and Philosophical Logic. UFO encompasses some microtheories related to wholes and parts, types and instantiation, rigidity, identity, dependency, mereology, among others, with the purpose of conceptualizing the fundamental elements of the conceptual modeling discipline. In order to design a structural modeling language, OntoUML was conceived to reflect the UFO taxonomy through a fragment of the UML 2.0 class diagram. This section discusses the meta-properties of the OntoUML 2.0 taxonomy (GUIZZARDI et al., 2018) that will guide our transformation process.

2.2.1 OntoUML

OntoUML has been used in different domains, including Geology, Biodiversity Management, Organ Donation, Petroleum Reservoir Modeling, Disaster Management, Context Modeling, Datawarehousing, Enterprise Architecture, Data Provenance, Measurement, Logistics, Complex Media Management, Telecommunications, Heart Electrophysiology, among many others (GUIZZARDI et al., 2015). The study carried out by Verdonck (2018) shows that OntoUML can provide additional quality to conceptual models while having a modest learning curve, being among the most used modeling languages in the Ontology-Driven Conceptual Modeling (ODCM) literature (VERDONCK; GAILLY, 2016). In addition to usability and applicability, there are existing studies that transform OntoUML diagrams into other languages, such as an automated transformation to OWL (ZAMBORLINI; GUIZZARDI, 2013), different transformations to Alloy (BENEVIDES et al., 2011; BRAGA et al., 2010) and more recently a transformation to gUFO-based OWL ontologies[2], which is adopted in this work.

In order to design an ODCM language, a set of stereotypes was proposed to be used into UML class diagrams in order to support the UFO taxonomy, which resulted in the OntoUML language. The UML class diagram "profiles" mechanism allows its metaclasses to be extended to give rise to new types of classes. These new types of classes can be used in the user model as stereotypes. OntoUML uses these mechanisms to reflect the UFO taxonomy. Figure 9 shows an overview of the profile, reflecting the UFO taxonomy of universals. The profile is governed by a set of *semantically motivated syntactic constraints* (CARVALHO; ALMEIDA; GUIZZARDI, 2014) that ensure the user's model complies with UFO. The presentation of UFO in this section follows largely (GUIDONI; ALMEIDA; GUIZZARDI, 2020). Further reference and formalization, see (GUIZZARDI, 2005; GUIZZARDI et al., 2021a).

Take a subject domain focused on objects (as opposed to events or occurrences). Central to this domain we will have a number of object *kinds*, i.e., the genuine fundamental types of objects that exist in this domain. The term "kind" is meant here in a strong technical sense, i.e., by a kind, we mean a type capturing essential properties of the things it classifies. In other words, the objects classified by that kind could not possibly exist without being of that specific kind. In Figure 2, we have represented two object kinds, namely, `Person` and `Organization`.

2 <https://purl.org/nemo/doc/gufo>

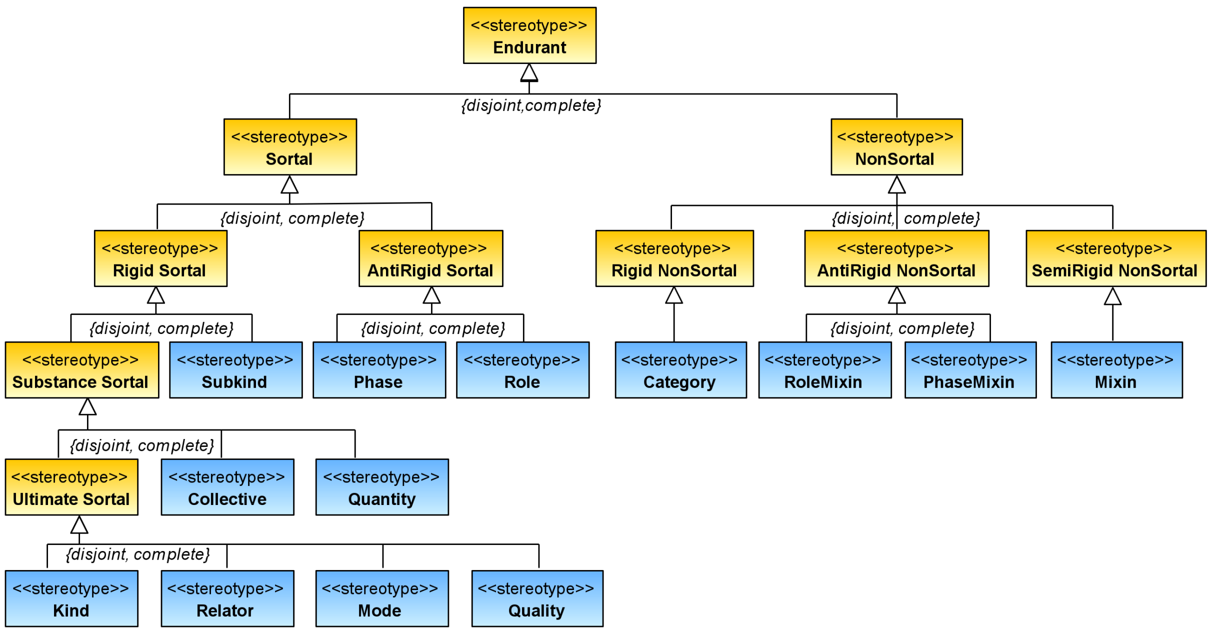


Figure 9 – OntoUML 2.0 profile stereotypes reflecting UFO taxonomy of universals.

These are the fundamental kinds of entities that are deemed to exist in the domain. Kinds tessellate the possible space of objects in that domain, i.e., all objects belong necessarily to exactly one kind.

Static subdivisions (or subtypes) of a kind are naturally termed *subkinds*. In our example, the kind `Organization` is specialized in the subkinds `Primary School` and `Hospital`. Object kinds and subkinds represent essential properties of objects, i.e., properties that these objects instantiate in all possible situations. They are examples of what are termed *rigid* or static types. There are, however, also types that represent contingent or accidental properties of objects (termed *anti-rigid* types). These include *phases* and *roles*. Phases represent properties that are intrinsic to entities; roles, in contrast, represent properties that entities have in a relational context, i.e., contingent relational properties. In our example, we have a *phase partition* including `Child` and `Adult` (as phases in the life of a `Person`). Several other types in the example are *roles*: `Employee`, `Contractor`, `BrazilianCitizen` and `ItalianCitizen` (the last two in the context of a relation with a national state, not represented in the model, for simplicity).

Kinds, subkinds, phases, and roles are all object *sortals*. In the philosophical literature, a sortal is a type that provides a uniform principle of identity, persistence, and individuation for its instances (GRANDY; FREUND, 2021). A sortal is either a kind (e.g., `Person`) or a specialization of a kind (e.g., `Child`, `Employee`, `Hospital`), i.e., it is either a type representing the essence of what things are or a sub-classification of entities that "have that same type of essence". There are also types that apply to entities of multiple kinds, these are called *non-sortals*. An example of non-sortal is `Customer` (which can be played by both people and organizations). We call these role-like types that classify entities of multiple kinds *role mixins*. Another example of non-sortal is `NamedEntity`. However, it is a rigid non-sortal, classifying objects of various kinds statically.

In addition to objects, there are also *existentially dependent* endurants, i.e., endurants that depend on other endurants for their existence. Here, we highlight the so-called *relators*, which reify a relationship *mediating* endurants. In our example, an instance of `Employment` can only exist as long as a particular instance of `Person` (playing the `Employee` role) and a particular instance of `Organization` (playing the corresponding `Employer` role, omitted here) exist. The meta-properties we have discussed for object types also apply to relator types. In our example, `Employment`, `Enrollment` and `SupplyContract` are relator kinds. With the reification of these relationships into relators, the challenge is already addressed at the conceptual model level, with many other benefits (GUARINO; GUIZZARDI, 2015).

Relators are composed of another type of dependent endurant termed a *qua-entity* (or *role instance*) (GUIZZARDI, 2005). Each qua-entity composing a relator inheres in one of the relatum of that relator while being relationally dependent on the other relata. For example, suppose John works for the UN. In this case, there is particular relator instance of `Employment` connecting John and the UN, which is composed of an instance of `Person-qua-Employee` (inhering in John and relationally dependent of the UN) and an instance of `Organization-qua-Employer` (inhering in the UN and relationally dependent on John). As discussed by Guizzardi (2005), qua entities are typically not explicitly represented in type-level conceptual models. There is, however, a formal connection between qua entities and roles, namely, an individual instantiates a role iff it bears a qua entity of a particular type (e.g., John instantiates the role of `Employee` because there is an individual–*John-qua-employee-of-the-UN*–inhering in him).

Figure 10 revisits Figure 2, now including class stereotypes according to the ontological distinctions discussed above, which are part of UFO-based OntoUML profile (GUIZZARDI, 2005). According to the rules that apply to OntoUML (formally characterized in (GUIZZARDI et al., 2018)):

- non-sortals (such as «category» and «roleMixin»), when present, are always superclasses (and never subclasses) of sortals (such as «kind», «subkind», «role», «phase», «relator»);
- non-sortals are abstract and are only instantiated through their sortal subclasses;
- sortals that are not kinds («subkind», «phase» or «role») specialize *exactly one* «kind» (or *relator kind* stereotyped «relator»), from which they inherit their principle of identity. So, there is no multiple inheritance of kinds, since all kinds are mutually disjoint;
- rigid types («kind», «subkind», «category», «relator») never specialize anti-rigid types («role», «roleMixin» or «phase»).

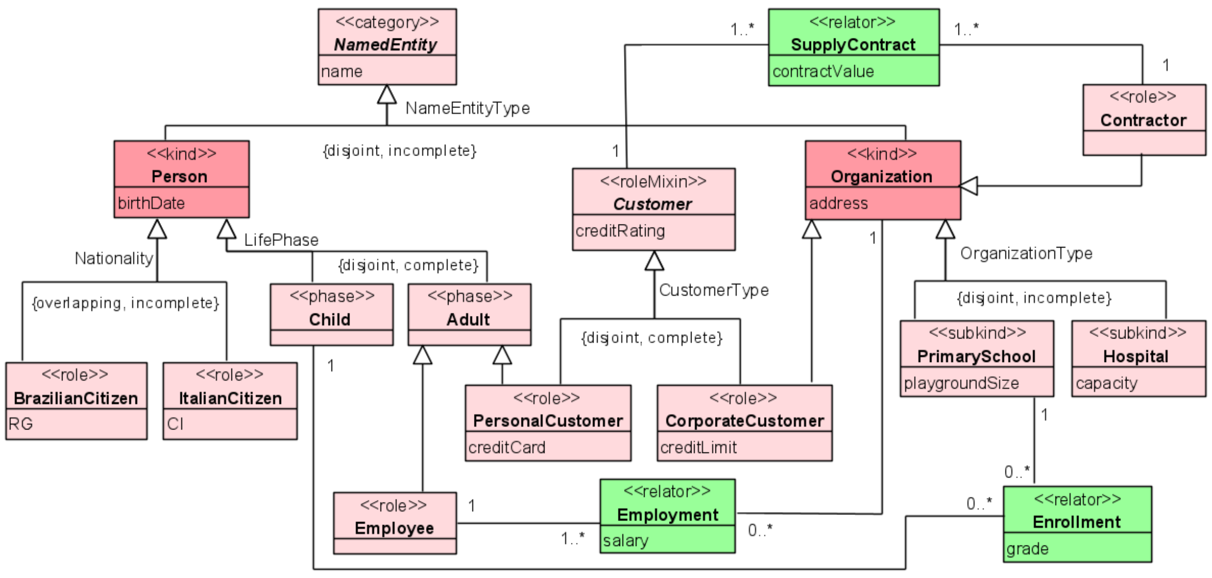


Figure 10 – Running example in OntoUML.

2.3 Final Considerations

In this chapter, we have discussed some *object-relational transformation strategies* widely consolidated in the literature and some principles of *Ontology-Driven Conceptual Modeling*.

Following the transformation strategies of inheritance hierarchies presented by Fowler (2002), Torres et al. (2017), Keller (1997), Ambler (2003), Yoder & Johnson (2002), we have exposed the result of these strategies to the resulting relational schema. By observing the *one table per hierarchy* strategy, we can identify an operation of 'lifting' of classes that is applied throughout the transformation process. By observing the *one (flattened) table per concrete class* and the *one flattened table per leaf class* strategies, we can identify an operation of 'flattening' of classes. We have also observe that, in some cases, there is inappropriate support for important conceptual modeling constructs (especially multiple inheritance, orthogonal classification and dynamic classification). Further, we have argued that a number of challenges remain concerning the usability and performance of some of the schemas that are produced by existing strategies.

We have discussed how *Ontology-Driven Conceptual Modeling* provides additional primitives to enrich a conceptual model. We have discussed some principles of the Unified Foundational Ontology (UFO) and presented the OntoUML language. UFO (and thus OntoUML) provides key distinctions for the creation of sound inheritance hierarchies considering the sortality and rigidity of classes. We argue that these properties can guide the transformation of class hierarchies into relational schemas.

In the next chapter, we use the operations of flattening and lifting along with the ontological semantics of classes to guide the application of these operations in an overall transformation process.

3 A Novel Ontology-Based Transformation Strategy

In Chapter 2, we have observed that there are a number of deficiencies of existing conceptual model transformation approaches, with (i) poor performance in various data manipulation operations, (ii) failure to explore beneficial database mechanisms, and/or (iii) lack of support for various conceptual modeling primitives including orthogonal classification hierarchies, overlapping non-exhaustive generalization sets as well as dynamic classification and multiple inheritance. This chapter addresses this gap, focusing on the realization of taxonomic hierarchies of ontology-based conceptual models. More specifically, we explore sortality and rigidity to propose a new transformation and avoid some problems in existing approaches.

This chapter is further organized as follows: Section 3.1 formalizes the basic operations of flattening and lifting that can be used in the object-relational transformation process to deal with inheritance hierarchies. Section 3.2 discusses how these operations can be applied in a new overall transformation process, which applies them in the light of sortality and rigidity: non-sortals are flattened recursively and sortals are lifted until kinds are reached. The new strategy is called *one table per kind*. Section 3.3 presents some final considerations.

3.1 Basic Transformation Operations

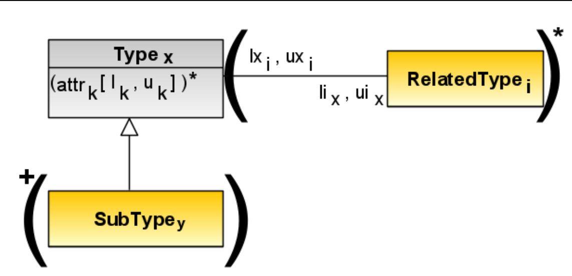
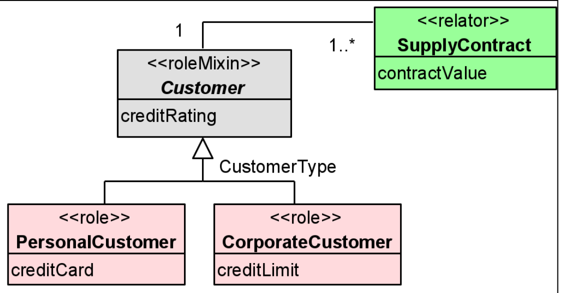
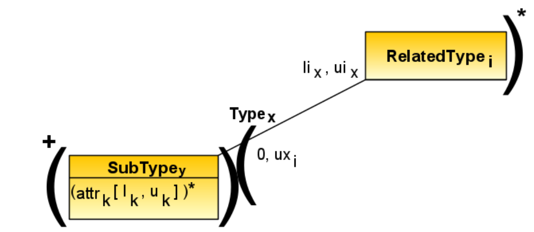
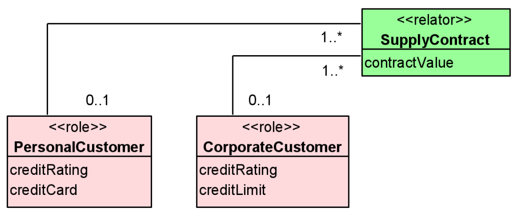
The transformation strategies presented in Chapter 2 have as approach the propagation of features between the classes of the hierarchy, with the progressive removal of classes which have their features propagated. The propagation of features can occur in two directions: (i) from top to bottom, i.e., from top-level classes to leaf classes, and (ii) from the bottom to the top, i.e., from leaf classes to top-level classes. We call the operations that perform these propagations *flattening* and *lifting*, respectively. In the following sections, we present the formalization of these operations and their consequences. These operations basically correspond to the graph transformation *model abstraction rules* proposed in (GUIZZARDI et al., 2019), albeit used here with a different purpose. The *flattening* operation corresponds to the *non-sortal abstraction rule* (R_2), and the *lifting* operation is a combination of the *sortal abstraction rule* (R_3) and the *subkind and phase partition abstraction rule* (R_4).

3.1.1 Flattening

The figures in Table 2 shows the transformation pattern of inheritance hierarchy and their relationships for the *flattening* operation. In these figures, the class $\texttt{Type}_x$ represents a superclass from the domain. The class $\texttt{SubType}_x$ represents a subclass (specialization) of the superclass. The class $\texttt{RelatedType}_i$ represents a different class from the domain related to the superclass through an association. Variables `l` and `u` represent, respectively, the lower and upper bound multiplicities of the features involved (attributes and association ends). When they identify the multiplicities of association ends, they are indexed by the target and source class. For example: $\texttt{lx}_i$ indicates the lower cardinality of participation of the class with index `i` ($\texttt{RelatedType}_i$) in the class with index `x` ($\texttt{Type}_x$). The symbol * indicates zero or more occurrence of an element and the symbol $^+$ indicates one or more occurrence of an element.

In the *flattening* operation, every attribute of the class that is flattened ($Type_x$ in gray of Table 2) is migrated to each of its direct subclasses ($SubType_y$) and the flattened class is removed from the model. Association ends attached to the flattened superclass are also migrated to the subclasses (creating new associations in the process, one for each subclass). The lower bound cardinality of the migrated association end (lx_i) is relaxed to 0, as the original lower bound is not necessarily satisfied for each of the subclasses. The cardinalities of attributes ($attr_k$) as well as association ends attached to classes *other than* the flattened class ($RelatedType_i$) are maintained, as these are invariants that apply to all subclasses in virtue of the semantics of specialization. (For simplicity, the diagram represents only one association between $Type_x$ and a $RelatedType_i$, but in fact, there can be many such associations.)

Table 2 – Flattening operation.

	Flattening Rule	**Example**
Source Graph	 Type x (attr k [l k , u k])* lx i , ux i li x , ui x RelatedType i * + SubType y	 1 1..* <<relator>> SupplyContract contractValue <<roleMixin>> Customer creditRating CustomerType <<role>> PersonalCustomer creditCard <<role>> CorporateCustomer creditLimit
Target Graph	 li x , ui x RelatedType i * Type x 0, ux i + SubType y (attr k [l k , u k])*	 <<relator>> SupplyContract contractValue 1..* 1..* 0..1 0..1 <<role>> PersonalCustomer creditRating creditCard <<role>> CorporateCustomer creditRating creditLimit

For example, in the flattening of the `Customer` class in the source graph row of Table 2, the `creditRating` attribute is migrated to the `PersonalCustomer` and `CorporateCustomer` classes without changing multiplicity, and the `Customer` class is removed from the model. New associations are created from `PersonalCustomer` and `CorporateCustomer` to `SupplyContract`, relaxing the minimum multiplicity with `PersonalCustomer` and `CorporateCustomer`.

Self-Relationships. Self-relationships in a class being flattened need to be handled as special cases. Instead of a single association end migrated to subclasses, we have in this case two association ends that are migrated. This can be approached in two separated steps. First, one association end is migrated to each subclass, following the procedure defined above for associations between different classes. This results in one new association for each subclass. Then, in the second step, the association ends that are attached to the superclass are migrated to each subclass, again following the procedure defined above. In the end, all lower bound cardinalities are relaxed to `0`, and all subclasses became related (and have a resulting self-relationship).

3.1.2 Lifting

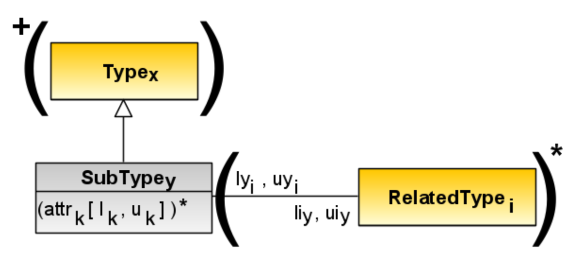
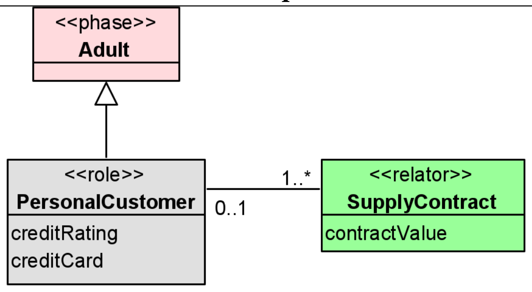
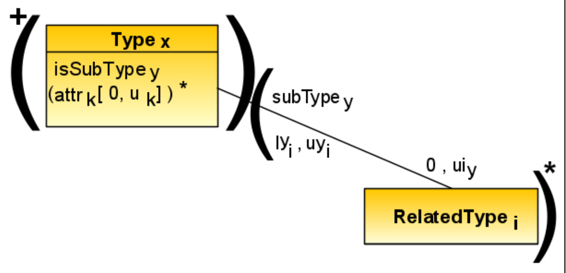
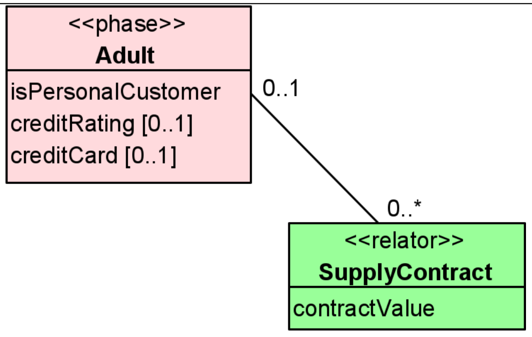
The figures in this subsection show the transformation pattern for an inheritance hierarchy when applying the *lifting* operation. In this operation, every attribute of the class that is lifted ($SubType_y$ in gray of Table 3 and Table 4) is migrated to each direct superclass, with lower bound cardinality (l_k) relaxed to `0` (i.e., mandatory attributes become optional). Upper bound cardinality (u_k) is maintained. 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 ($RelatedType_i$) (if any) are relaxed to `0` in the same way as the attributes of the lifted class.

Simple generalization. When no generalization set is present (or for generalization sets with a single subclass), a Boolean attribute is added to each superclass, to indicate whether the instance of the superclass instantiates the lifted class ($isSubType_y$). Is is prefixed "is" followed by the name of the lifted class. See Table 3.

For example, the attributes of the lifted `PersonalCustomer` class in the source graph row of Table 3 are migrated to the `Adult` class, relaxing their lower bound cardinality to `0`. One new association is created between `Adult` and `SupplyContract`, relaxing the minimum multiplicity of the association end attached to `SupplyContract`.

Generalization sets. In some cases, generalizations are grouped into generalization sets, which are sets of subclasses that specialize the superclass according to a common specialization principle. For example, people can be further classified by their nationality, stage of life, profession, gender, among many others. Each of these principles may give rise to a different generalization set specializing the class `Person`. These grouping principles may even be explicitly identified with *powertypes*, whose instances are subclasses of the base type that is specialized by the set (CARVALHO; ALMEIDA; GUIZZARDI, 2016). Lifting in this case requires a *discriminator* to be added to the superclass, as a means to identify the lifted class that is instantiated. For this purpose, a discriminator enumeration is created (GS_a) in the superclass with labels corresponding to each $SubType_y$ in the generalization set. An attribute with that

Table 3 – Lifting with simple generalization.

	Lifting Rule for Simple Generalization	Example
Source Graph		
Target Graph		

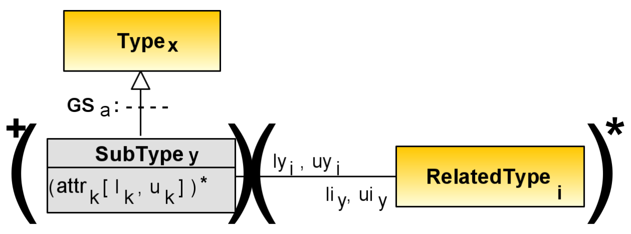
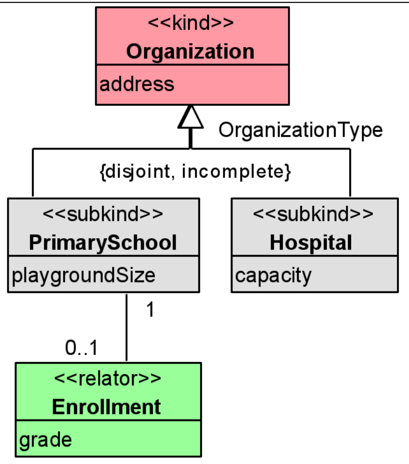
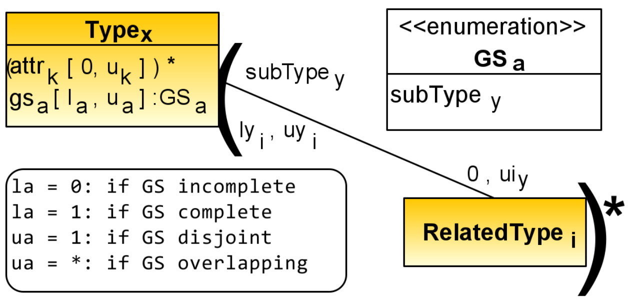
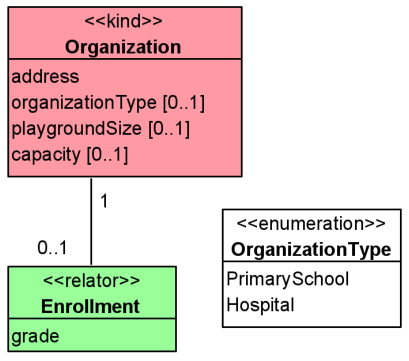
discriminator type is added to each superclass (gs_a). Its cardinality follows the generalization set arrangement:

- *disjoint* - indicates that "at most one" of the subclasses may be instantiated for each superclass instance, so the maximum cardinality of gs_a is set to 1 (one).
- *overlapping* - indicates that "more than one" subclass may be instantiated by the same superclass instance, so the maximum cardinality of gs_a is set to * (many).
- *complete* - indicates that whenever there is an instance of the superclass, it instantiates "at least one" subclass in the set, so the minimum cardinality of gs_a is set to 1 (one).
- *incomplete* - indicates that there are instances of the superclass that are not instances of any of the subclasses, so the minimum cardinality of gs_a is set to 0 (zero).

For example, the attribute of `PrimarySchool` and `Hospital` classes in the Source Graph line of the Table 4 is lifted to `Organization` class changing its multiplicity to be optional. The enumeration `OrganizationType` is created with the names of all subclasses as literals and a new attribute is created in the `Organization` class to identify the "type of organization" instantiated. The multiplicity of this new attribute is optional because the generalization set is *incomplete*. Finally, `PrimarySchool` and`Hospital` classes are removed from the model.

A precondition to the application of lifting is that the class is a leaf of the present hierarchy (i.e., it has no subclass), and, if it is a subclass in a generalization set, its siblings in the generalization set must also be leaves of the present hierarchy. Approaches that rely on lifting (such as *one table per hierarchy*) usually rule out multiple inheritance, since in the presence of

Table 4 – Lifting rule when applied in a generalization set of sortals.

	Lifting Rule for Generalization Set	Example
Source Graph	 Type x GS a : SubType y (attr k [l k , u k])* ly i , uy i li y, ui y RelatedType i + *	 <<kind>> Organization address OrganizationType {disjoint, incomplete} <<subkind>> PrimarySchool playgroundSize <<subkind>> Hospital capacity 1 0..1 <<relator>> Enrollment grade
Target Graph	 Type x (attr k [0, u k]) * gs a [l a , u a] :GS a subType y ly i , uy i <<enumeration>> GS a subType y 0 , ui y RelatedType i * la = 0: if GS incomplete la = 1: if GS complete ua = 1: if GS disjoint ua = *: if GS overlapping	 <<kind>> Organization address organizationType [0..1] playgroundSize [0..1] capacity [0..1] 1 0..1 <<relator>> Enrollment grade <<enumeration>> OrganizationType PrimarySchool Hospital

multiple inheritance, the preservation of the cardinalities (ly_i and uy_i becomes problematic) in the lifting step. Here, we operate under the assumption that multiple inheritance is admissible, but that further lifting steps will end up consolidating the various associations introduced due to lifting to various subclasses into a single one; in other words, we assume that lifted classes are ultimately indirect specializations of a single class[1].

3.2 One Table per Kind

The approach we propose here makes combined and selective use of both “lifting” and “flattening”. The approach leverages the specialized semantics of the OntoUML to precisely determine which classes should be flattened and which should be lifted—given a particular set of conceptualization choices made about the domain. This is possible because OntoUML explicitly represents in a system of stereotypes a number of finer-grained ontological distinctions among types of classes. These stereotypes enable the explicit representation of these choices.

In a nutshell, we propose to group the classes of the inheritance hierarchy into their respective kinds because there is a significant body of evidence in cognitive psychology (MAC-

[1] When applied to the *one table per hierarchy* approach, from which this operation was identified, this means that hierarchies must be disjoint (i.e., there is no class that specializes more than one top-level class).

NAMARA; MACNAMARA; REYES, 1994; XU, 1997; XU; CAREY, 1996) that object kinds are the most relevant category of types in human cognition, being responsible for our most basic operations of individuation and object identity. Therefore, this strategy results in a schema composed of tables corresponding to the *kinds* of entities in the domain. Because of this, it is termed *one table per kind* (GUIDONI; ALMEIDA; GUIZZARDI, 2020). The *one table per kind* strategy is divided into three steps, outlined below.

Steps 1 and 2 of the *one table per kind* transformation strategy are refactoring processes applied in the model in order to remove the inheritance relationships, according to the constructs found in the conceptual model. Thereby, at the end of steps 1 and 2 we will still have the class diagram without the inheritance constructs. The transformation to the relational schema is performed only in step 3.

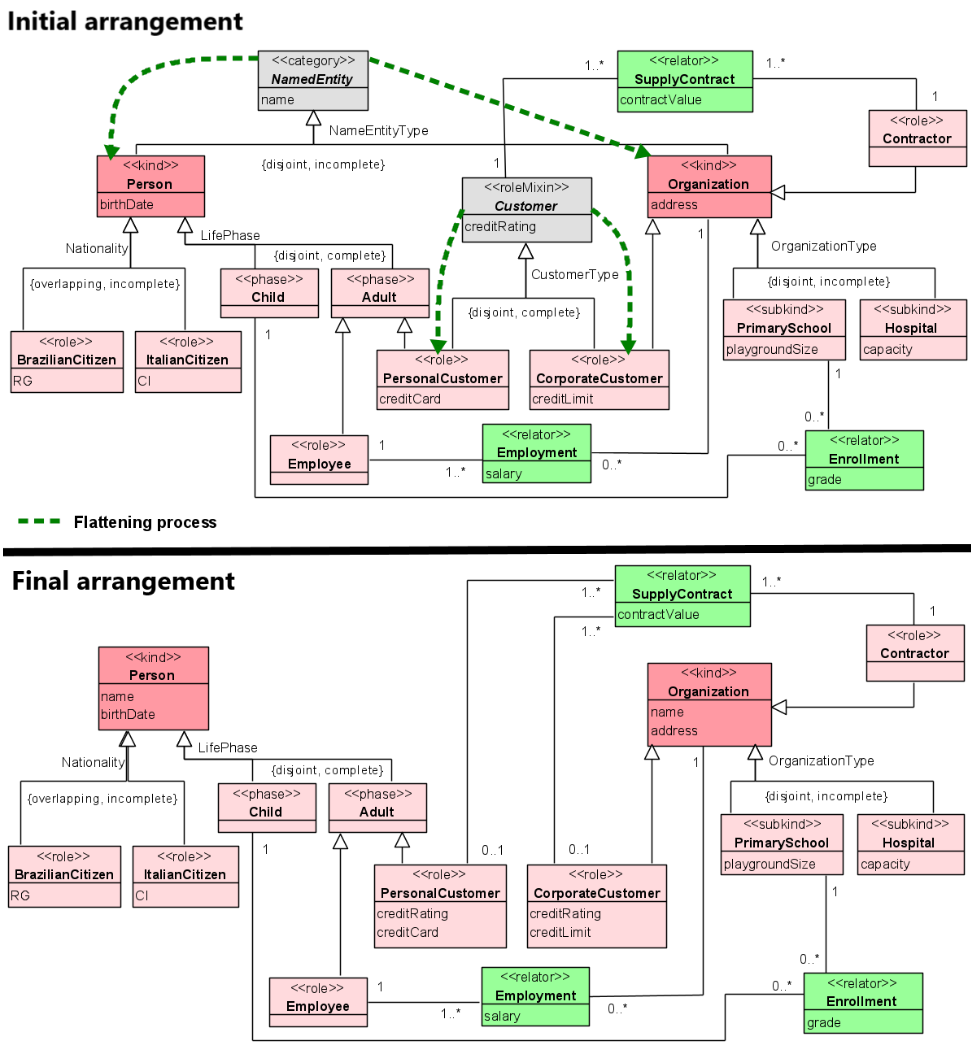

Initial arrangement
<<category>> NamedEntity name
NameEntityType
{disjoint, incomplete}
<<kind>> Person birthDate
<<relator>> SupplyContract contractValue
1..*
1..*
1
<<role>> Contractor
<<kind>> Organization address
<<roleMixin>> Customer creditRating
CustomerType
{disjoint, complete}
Nationality
LifePhase
{overlapping, incomplete}
<<phase>> Child
<<phase>> Adult
<<role>> BrazilianCitizen RG
<<role>> ItalianCitizen CI
<<role>> PersonalCustomer creditCard
<<role>> CorporateCustomer creditLimit
OrganizationType
{disjoint, incomplete}
<<subkind>> PrimarySchool playgroundSize
<<subkind>> Hospital capacity
<<role>> Employee
<<relator>> Employment salary
<<relator>> Enrollment grade
0..*
Flattening process
Final arrangement
<<kind>> Person name birthDate
<<kind>> Organization name address
<<role>> PersonalCustomer creditRating creditCard
<<role>> CorporateCustomer creditRating creditLimit
0..1

Figure 11 – Flattening process example.

Step 1: Non-sortals are flattened towards sortals. Non-sortals are always superclasses (and never subclasses) of sortals. They are also only instantiated through their sortal subclasses (and are represented by abstract classes). The flattening operation is applied repeatedly to the topmost non-sortals of the hierarchy, until no non-sortal class is left in the model.

Exemplifying the flattening operation in our running example: the `Customer` class is a non-sortal, so the `creditRating` attribute is flattened to `PersonalCustomer` and `CorporateCustomer` classes without changing its multiplicity, and the `Customer` class is removed from the model (the same occurs for `NamedEntity` class), as shown in Figure 11. Two new associations are created, one between `SupplyContract` and `PersonalCustomer` and other between `SupplyContract` and `CorporateCustomer`. The minimum multiplicity of new associations with `PersonalCustomer` and `CorporateCustomer` classes are relaxed to accept zero association (0..1), because it is not true that every association with `Customer` necessarily means an association with `PersonalCustomer` and with `CorporateCustomer`.

Step 2: The attributes and associations of the non-kind sortals are lifted (ultimately) to their kinds. The lifting operation is applied from the most specific non-kind sortal until all of them have been lifted to their kinds.

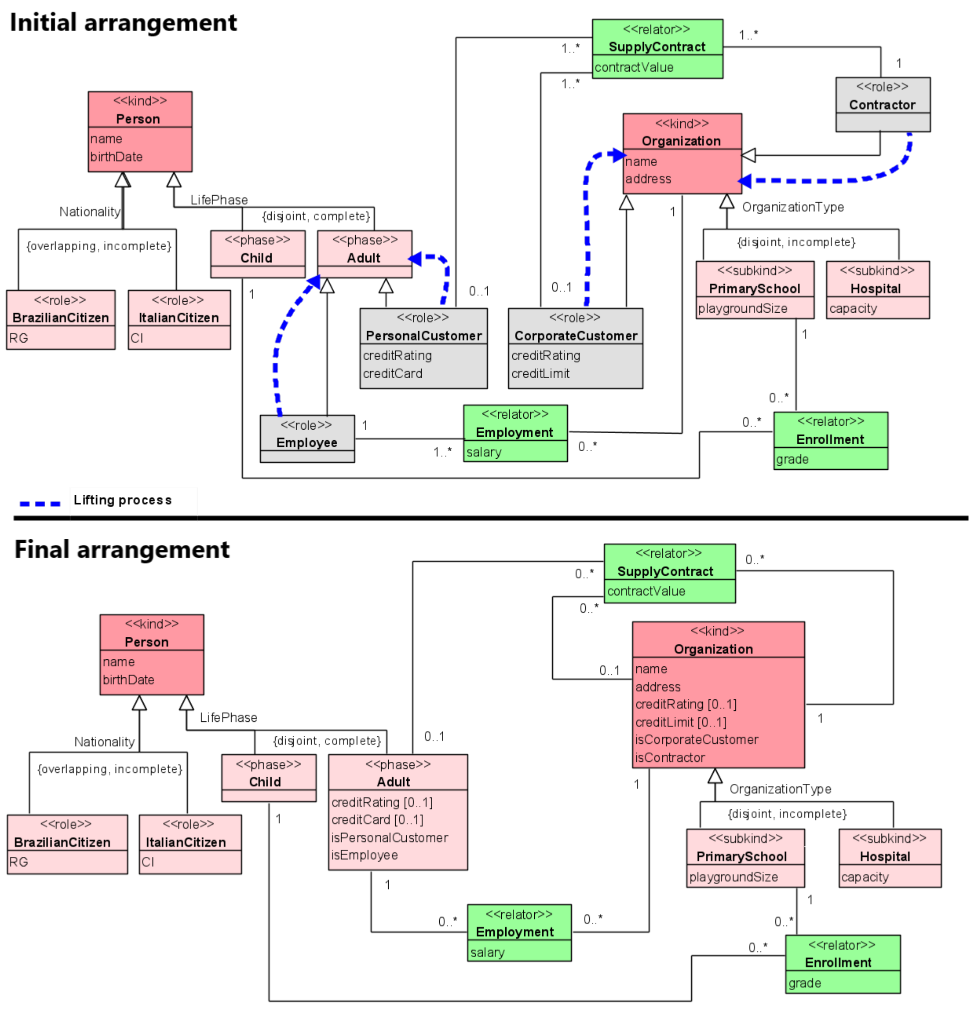

Initial arrangement
<<relator>> SupplyContract contractValue
<<kind>> Person name birthDate
<<role>> Contractor
<<kind>> Organization name address
Nationality
LifePhase
{disjoint, complete}
{overlapping, incomplete}
<<phase>> Child
<<phase>> Adult
OrganizationType
{disjoint, incomplete}
<<subkind>> PrimarySchool playgroundSize
<<subkind>> Hospital capacity
<<role>> BrazilianCitizen RG
<<role>> ItalianCitizen CI
<<role>> PersonalCustomer creditRating creditCard
<<role>> CorporateCustomer creditRating creditLimit
<<role>> Employee
<<relator>> Employment salary
<<relator>> Enrollment grade
Lifting process
Final arrangement
<<kind>> Organization name address creditRating [0..1] creditLimit [0..1] isCorporateCustomer isContractor
<<phase>> Adult creditRating [0..1] creditCard [0..1] isPersonalCustomer isEmployee

Figure 12 – Lifting process example for simple generalization.

Exemplifying the lifting operation in our running example: the `CorporateCustomer` class is a sortal non-kind, so the `creditRating` and `creditLimit` attributes are lifted to `Organization` class changing its multiplicity to optional (0..1). After that, the `isCorporate-Customer` attribute is created in `Organization` and the `CorporateCustomer` class is removed from the model, as shown in Figure 12 (the mandatory multiplicities are not shown). The minimum association with `SupplyContract` is relaxed, because not every `Organization` instance is a `CorporateCustomer` instance, so not every organization will have an association with `SupplyContract`.

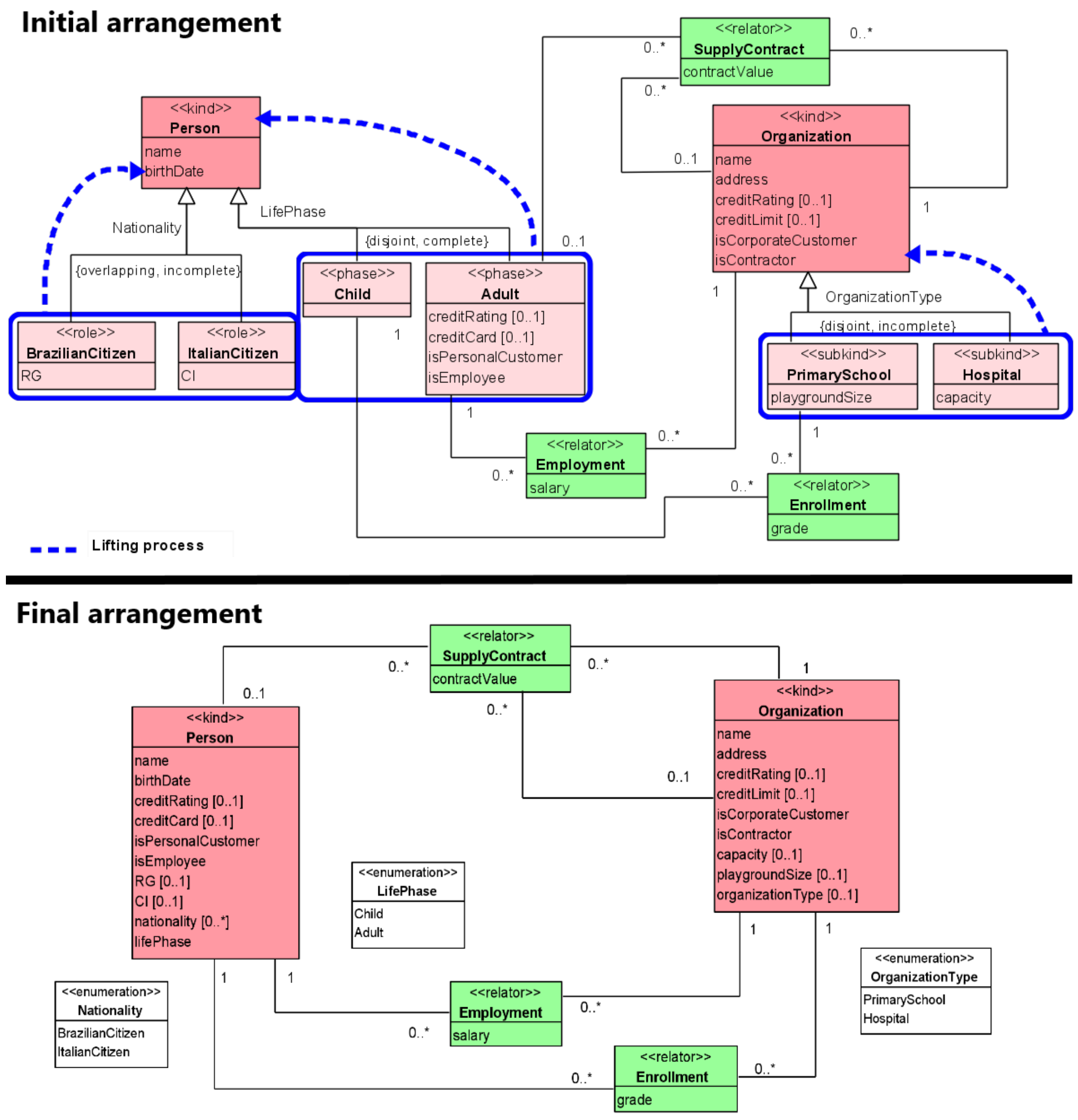


Figure 13 – Lifting process example for generalization set.

Figure 13 shows the lifting of the generalization sets in *step 2*. In this operations, as the `BrazilianCitizen` and `ItalianCitizen` classes are non-kind sortals, making a generalization set with `Person`, so the `RG` and `CI` attributes are lifted to `Person` class changing their multiplicities to optional ([0..1]). The enumeration `Nationality` is created with the names of all subclasses as literals, and the subclasses are removed from the model. The multiplicity of the association between the `Person` and `Nationality` classes is defined as follows: since there is nothing that requires the instantiation of each subclass and at the same time one subclass can be instantiated multiple times, the association multiplicity on the `Person` side is set to [0..∗]; the minimum multiplicity on the `Nationality` side is set to 0 because the generalization set is *incomplete*, that is, there may be people who are do not instantiate any of the subclasses of `Person` in that set, whereas the maximum multiplicity of the Nationality side is ∗ because the generalization set is *overlapping*, that is, a person can have more than one nationality.

Step 3: After these operations have been carried out, tables are produced for each of the classes in the refactored model. The tables corresponding to dependent entities must have foreign keys to the entities on which they depend. This is the case for tables corresponding to relator, kind, and also for the discriminating tables in the case of overlapping generalization sets. In the latter case, each row in a discriminator table represents a qua-entity connecting role players with the corresponding (reified) role class. As previously discussed, qua-entities and relators are existentially dependent entities.

Figure 14 presents the schema that results from the application of these transformation steps in the conceptual model in Figure 10. We obtain the two tables corresponding to object kinds: PERSON, ORGANIZATION, three corresponding to relators: EMPLOYMENT, ENROLLMENT and SUPPLY_CONTRACT. An additional table for the discriminator that results from the overlapping generalization set `Nationality` is introduced (NATIONALITY, representing a qua-entity connecting a person to a particular nationality type).

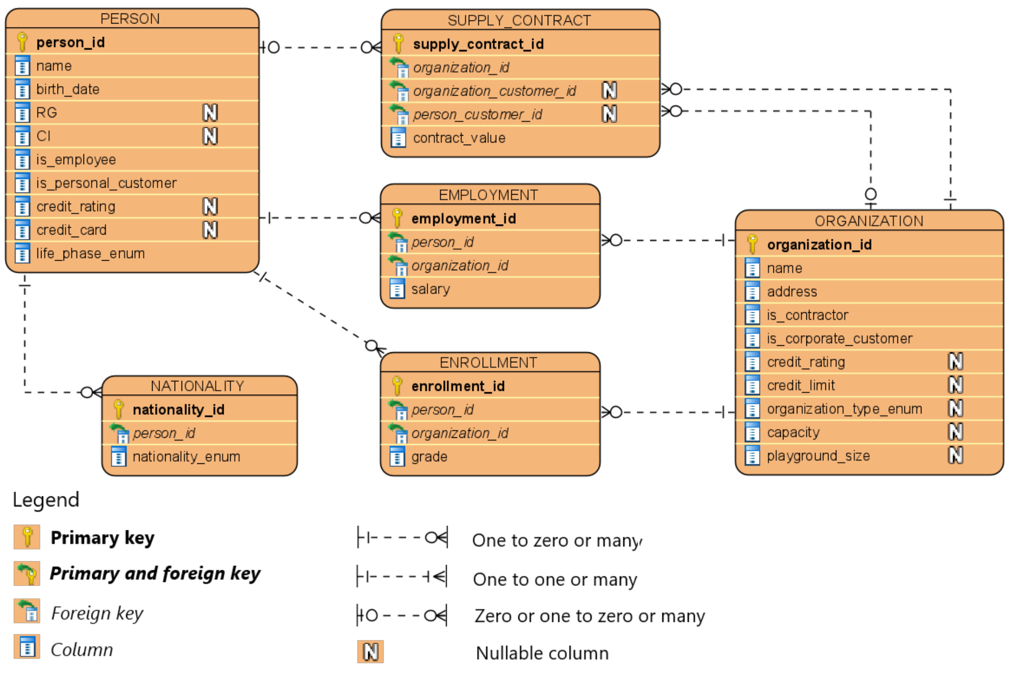


Figure 14 – Running example transformed by *one table per kind* strategy.

3.3 Final Considerations

This chapter has formalized the elementary *flattening* and *lifting* operations that are used to address specialization relations in a class hierarchy. Applying these operations in different orders results in different object-relational transformation strategies. For example, in the *one table per concrete class* strategy (PHILIPPI, 2005) "flattening" is applied for the abstract superclasses repeatedly. The extreme application of "flattening" to remove all non-leaf classes of a taxonomy yields *one table per leaf class* ("horizontal inheritance" (TORRES et al., 2017)). When "lifting" is applied to all subclasses repeatedly we have the opposite extreme: *one table per hierarchy* (also called "single-table" (FOWLER, 2002) or "one inheritance tree one table" (KELLER, 1997)). Differently from these strategies that apply only one operation, the *one table per kind* strategy combines flattening and lifting by employing ontological semantics to determine when they should be applied in the transformation processes. The resulting schema has a table corresponding to each kind in the source conceptual model.

Using the terminology of Lano et al. (2018), Lúcio et al. (2016), the *flattening* and *lifting* operations presented in this chapter can be understood conceptually as *endogenous* model *refactorings*. An endogenous transformation is one whose target and source models are represented in the same language (here they are both class diagrams); a refactoring is a transformation that performs *update-in-place* making local changes to a part of the model. These are usually motivated with the *model editing* intent (LÚCIO et al., 2016), although here, they have been motivated by progressing towards a model that can be more easily *translated* in an ultimate exogenous model transformation (one that produces a relational schema from a class diagram). Because of that, in some cases, our operations have the opposite intent of some refactoring operations in the literature. For example, flattening is the opposite of "extract class" of Fowler (2018). ("Flattening" as used here should not be mistaken with the homonymous model transformation *pattern* of Iacob, Steen & Heerink (2008), which refers to the elimination of hierarchical containment structures in the abstract syntax.) The operations are applied automatically once the classes to be flattened or lifted in a step are identified.

As can be seen, these operations are not linked to the OntoUML categories (represented by UML stereotypes), therefore can be applied to any conceptual model based on the UML class diagram. But here, they are put to use in the *one table per kind* strategy, and hence, their applications is guided by the ontological constructs of OntoUML.

The study of ontological foundations in conceptual modeling has produced a number of advances over the last decades. This chapter extends some of these advances to relational schema design. We have shown that by considering the ontological status of classes in a conceptual model we can propose a new transformation strategy that cannot be articulated with ontologically neutral conceptual modeling primitives (providing thus support for hypothesis 1).

The next chapter compares our strategy with the other strategies presented in Section 2.1, discussing how it provides adequate support to all the constructs mentioned in Section 1.2. We also present some considerations regarding the size, performance and usability of relational schemas produced with the strategy.

4 Evaluation of the Novel Strategy

Unlike the various object-relational transformation strategies in the literature, the proposed *one table per kind* strategy uses principles and constructs provided by an ontological language to guide the object-relational transformation. As we have hypothesized, this leads to improvements in the quality of the resulting relational schemas. This chapter shows support for this hypothesis (hypothesis 2), assessing the various quality characteristics of the proposed strategy when compared to those in the literature, demonstrating our specific objective 2, 3 and part of specific objective 1.

Section 4.1 discusses the support to multiple classification, orthogonal hierarchies and dynamic classification. Section 4.2 compares the size of the resulting schema, drawing some considerations for the performance of data operations taking into account some statistics from previously published OntoUML models. Section 4.3 presents some measurements intended to show the feasibility of the *one table per kind* strategy when considering its query time performance in relation to other strategies. It also considers the performance of update operations. Section 4.4 reports on an empirical study to assess the relative understandability of a schema generated with the *one table per kind* strategy.

4.1 Support for Conceptual Modeling Primitives

Multiple Inheritance. Problems with multiple inheritance that appear specifically in the *one table per hierarchy* strategy do not appear in *one table per kind* because according to UFO (GUIZZARDI, 2005) there is no multiple inheritance of kinds. Hence, an instance is represented naturally in a single table corresponding to the unique kind (ultimate sortal) it instantiates. Multiple inheritance of non-kind sortals (subkinds, phases and roles), while possible, does not pose a problem, since they are lifted to the same common kind. Discriminators then identify the instantiated non-kind sortals. Multiple inheritance of non-sortals creates no problems because they are flattened into kinds, and hence all the inherited features from all superclasses are supported naturally in the table corresponding to the inheriting kind.

Orthogonal hierarchies. Problems with orthogonal hierarchies and overlapping generalization sets that appear in the *one table per leaf class* strategy do not arise in the *one table per kind* strategy. Kind tables are where the entities' primary keys are placed, and hence there is no problem with the same entity being represented in several tables. Flattening of non-sortals poses no problem in this scenario. In the case of the lifting of non-kind sortals, orthogonal hierarchies and overlapping generalization sets are, not unlike multiple inheritance, reflected in discriminators in the kind table.

Dynamic classification. Dynamic classification is supported for all strategies, however always requiring what can be considered a workaround in the *one table per class* and *one table per leaf class* strategies: the migration of rows representing the same instance, i.e., the deletion from a table corresponding to a class that is no longer instantiated by the entity and the insertion in a table corresponding to the newly instantiated class. Differently from those strategies, in *one table per hierarchy* and *one table per kind*, dynamic classification is supported by simply changing the value of a discriminator.

Table 5 summarizes the coverage of primitives in the transformation strategies considered, with the *one table per kind* combining the benefits of the other strategies.

Table 5 – Support for conceptual modeling primitives in different strategies.

Realization Strategy	**Multiple inheritance**	**Orthogonal hierarchies**	**Dynamic classification**
One table per class	yes	yes	yes (but with migration)
One table per leaf class	yes	no	yes (but with migration)
One table per hierarchy	no	yes	yes
One table per kind	yes	yes	yes

4.2 Preliminary Performance Considerations

Table 6 provides a comparison between the proposed *one table per kind* strategy and the other dominant strategies presented in Section 2.1. We consider some variables that may affect query performance: the number of tables representing entities, the number of joins to retrieve one entity (with all its attributes), the number of tables potentially affected in an insert operation, and the number of tables to read an attribute in a polymorphic query. We use the following parameters in this comparison: n is the total number of classes in the source conceptual model, h is the maximum height of the hierarchy (i.e., maximum path size from a top-level class to a leaf class), n_l is the number of leaf classes in the hierarchy, n_t the number of top-level classes, and n_k is the number of kinds. Note that the number of kinds (n_k) is equal to or lower than the number of leaf classes (i.e., $n_k \leq n_l \leq n$), and that they are equal ($n_k = n_l$) only in case there are no subkinds, roles and phases. Thus, the number of tables required to represent entities in the domain in the proposed *one table per kind* strategy is equal to or lower than that required by the other strategies, with the exception of strategy *one table per hierarchy*, as shown in "N^o of tables representing entities" column. The comparison with *one table per hierarchy* requires us to consider the number of top-level classes (n_t). The two approaches result in the same number of tables when there are no non-sortals ($n_k = n_t$).

The table also presents that *one table per kind* is an optimized strategy for the retrieval

Table 6 – Schema size and performance-related metrics in different strategies.

Realization Strategy	**N^o of tables representing entities**	**N^o of joins to retrieve an entity**	**N^o of tables affected in insert operation**	**N^o of tables in union to read one attribute (polymorphic query)**
One table per class	n	h	$h+1$	1
One table per leaf class	n_l	1	1	n_l
One table per hierarchy	n_t	1	1	1
One table per kind	n_k	1	1	n_k (1, if defined in sortal)

and insertion of an entity (with all its attributes), needing only one access to the database to perform the action, as shown in "N^o of joins to retrieve an entity" and "N^o of tables affected in insert operation" columns, respectively. Concerning the performance of polymorphic queries (number of tables involved in a union to read one attribute defined in a superclass), when the attribute is defined in a non-sortal, in which case, n_k unions may be required in the worst case for *one table per kind* (when the non-sortal class in which the attribute is defined classifies entities of all kinds in the model). Even in this case, the approach is equal to or better than *one table per leaf class* (since $n_k \leq n_l$).

Table 7 shows the values for these variables for a number of models in different domains, those also employed in (GUIZZARDI et al., 2019). Our intent is to show actual data from models that have been developed independently (and are part of the OntoUML/UFO catalog[1]). The data reveals that the height of the hierarchy (h) ranges from two to six, and the number of kinds in a model is typically one fourth or one fifth of the total number of classes. The average number of leaf classes (n_l) is 39, contrasted with 15 (n_k) for kinds. Thus, the *one table per kind* strategy leads to fewer tables (except when compared with *one table per hierarchy*, which cannot handle multiple inheritance). When compared with *one table per leaf class* polymorphic queries involve unions with fewer tables.

Table 7 – Variable occurrences by OntoUML model.

Variables	**OntoUML Models**										
	Cloud Vulnerability	**ECG**	**G.805**	**MPOG**	**Normative Acts**	**OpenBio**	**OpenFlow**	**Open Provenance**	**PAS 77**	**Software Requirements**	**Average**
n	30	49	123	15	59	231	20	33	66	17	**64**
h	3	2	6	4	3	5	2	2	3	2	**3**
n_l	12	18	70	7	43	163	8	12	41	11	**39**
n_t	5	4	14	3	5	9	4	8	5	2	**6**
n_k	12	18	18	5	10	37	6	17	19	7	**15**

[1] <https://github.com/OntoUML/ontouml-models>

4.3 Actual Database Performance in two Scenarios

We report here a study on the feasibility of the proposed strategy when compared to existing strategies from the perspective of time performance. We collect data regarding the performance of some relevant queries for different strategies. We use two source OntoUML models: our running example and a model produced by a third-party chosen from the OntoUML repository (BARCELOS et al., 2022). We chose our running example because it covers all primitives that we support including multiple inheritance, dynamic classification, different types of generalization sets, as well as orthogonal hierarchies. The other model belongs to the Object-Oriented Code Ontology project (OOC-O) (AGUIAR, 2021). It was chosen because of its complex inheritance hierarchies (merging in the same hierarchy simple inheritance and generalization sets with several levels, associations of subclasses in different levels with other classes and with subclasses of other hierarchies). Sections 4.3.1 and 4.3.2 are dedicated to each of these source models respectively.

We have chosen for the comparison the following strategies: *one table per concrete class*, *one table per leaf class* and *one table per kind*. The *one table per hierarchy* strategy was not included due to its lack of support for multiple inheritance, which is present in both source models. The relational schemas produced were automatically populated with the same synthetic instances as described in the sequel.

In order to exclude I/O from the response time, which could mask the difference between the strategies used, we have performed a row count for each query effectively "packaging" it into a "select count (1) from (query)". In the performance of update evaluation, each change was enacted separately. The values presented in this section are averages of 100 measurements. As an indication of relative performance, we provide a percentage of difference from *one table per kind* to the other best-performing strategy in the tables that display query response time. The query scripts used can be found in the Appendix C.

Measurements presented in this section were obtained in a Windows 10 notebook with an i3 1.8 GHz processor, 250 GB SSD and 8 GB RAM. Data were obtained through PostgreSQL 5.2, differently from (GUIDONI; ALMEIDA; GUIZZARDI, 2020), where MySql had been employed. We have decided to focus on PostgreSQL because it offered better overall time performance for all queries and exhibited less variance in the response time for the same query when executed multiple times. Note that a comparison of the performance of DBMSs is not part of the scope of this work, and would require in-depth studies on DBMS configurations, impact of the operational system, database size, etc. Further, application-specific analyses are still required, and the purpose of this section is solely to show the feasibility of the approach from the performance perspective, revealing some cases in which the *one table per kind* outperforms other strategies and other cases in which it does not.

4.3.1 Running Example Project

Using the running example as presented in Figure 10, we have created the relational schemas and populated them with:

- 50k organizations (about 42% hospitals, 38% primary schools and 20% without classification);
- 200k persons (about 40% Brazilian citizens, 40% Italian citizens, 15% with double nationality and 5% are stateless, over 160k adults, 40k children, 128k employees);
- about 250k supply contracts; 166k employments (30% of the employees with more than one employment), and
- 56k enrollments (40% of the children with more than one enrollment).

The database size was 110.848 MB for the *one table per concrete class* strategy, 96.645 MB for the *one table per leaf class* and 105.426 MB for the *one table per kind* strategy.

Query Answering Performance

In order to assess relative query answering performance considering our Running Example model, a number of 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 have different (representative) characteristics. Query 1 is a polymorphic query with reference to the non-sortal `Customer` and retrieves an attribute defined at that abstract class. Queries 2 and 3 involve navigation through associations in the conceptual model. 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.

Table 8 shows information regarding the number of tables, unions and filters of our queries considering the relational schemas of the three transformation strategies compared. We can see that queries written for the relational schema generated by the *one table per concrete class* strategy need to access more tables to retrieve the data, resulting in more joins. The queries written for the relational schema generated by the *one table per kind* strategy is the one that uses fewer tables, resulting in fewer joins to retrieve data; however, it is the one that requires more filters. Although Query 4 is polymorphic, it was implemented with "LEFT JOIN" for the *one table per concrete class* strategy, which cannot be done for the *one table per leaf class* strategy.

Table 8 – Number of tables, unions and filters for the queries based on different schemas over our running example model.

Query	**One table per concrete class**			**One table per leaf class**			**One table per kind**		
	Nº of tables	**Nº of unions**	**N° of filters**	**Nº of tables**	**Nº of union**	**N° of filters**	**Nº of tables**	**Nº of unions**	**N° of filters**
1	2	1	0	2	1	0	2	1	2
2	4	0	0	3	0	0	3	0	0
3	8	0	0	5	0	1	7	0	3
4	4	0	0	7	2	0	1	0	0
5	5	0	1	3	0	1	3	0	2

Table 9 shows the results obtained, comparing with the relational schemas proposed. The response times of Query 1 indicate that query performance depends directly on the number of tables accessed to retrieve data (see "Nº of tables" columns of Table 8), with a slight increase in the *one table per kind* strategy due to the filters performed. Query 2 presents an optimal result for the *one table per leaf class* strategy. This result was obtained because this strategy generates a specific table for children which are the specific subjects of this query. The same occurs for the `PRIMARY_SCHOOL` table. This advantage is not realized in the schema generated by the *one table per concrete class* strategy because is necessary to access the `PERSON` and `ORGANIZATION` tables to retrieve the `name` attribute. Despite the benefits for recovering data, the *one table per leaf class* strategy can be problematic for dynamic classification (Table 10 addresses this issue). The benefit of having a table corresponding to the specific entities queried is also present in Query 3 for *one table per leaf class.* However, the response time benefits are not as great as those for Query 2, because Query 3 requires joins with more tables and a filter to discard stateless people.

Query 4 exposes the problem of polymorphic queries for the *one table per leaf class* strategy. As the data are arranged only in the tables corresponding to the leaf classes of the hierarchy, it is necessary to perform two unions to retrieve data from all organizations (one `union` between `HOSPITAL` and `PRIMARY_SCHOOL` and another with `CORPORATE_CUSTOMER`, as the conceptual model allows organizations without defining the organization type). This is the

Table 9 – Queries response times (averages, in seconds) for the running example.

Query	One table per concrete class	One table per leaf class	One table per kind
1	**0.219**	0.220	0.251 (+14.6%)
2	0.272	**0.083**	0.245 (+195%)
3	0.623	**0.481**	0.585 (+21.6%)
4	0.041	0.697	**0.010** (-75.6%)
5	0.192	0.205	**0.182** (-5.5%)

case in which *one table per kind* excels, outperforming the other approaches by a good margin. Query 5 presents a subtle difference between the strategies, with a slight advantage for the *one table per kind* strategy. Although the *one table per concrete class* strategy accessed two more tables than the *one table per kind* strategy, the difference was not significant because the result of this query has just one person.

Performance of Update Operations

Table 10 – Times required for updates of records representing children (in seconds).

Operation	One table per concrete class	One table per leaf class	One table per kind
Insert	0.021	0.011	**0.006** (-45.5%)
Update (name and rg)	0.014	0.014	**0.006** (-57.7%)
Change `life_phase` **(to Adult)**	0.020	0.022	**0.006** (-70.0%)
Delete	0.055	**0.046**	0.115 (+150.0%)

We also evaluated some actions to insert, update and delete records, in the following order: first, we inserted 100 Brazilian children; second, we changed the `name` and `rg` attributes of these children; third, we changed the `life_phase` of this children to `ADULT`; lastly, we delete all the children initially inserted. Table 10 shows the response time for the proposed updates. The *one table per kind* strategy presents better results in general because updates involve only one table in the the resulting schema. The *one table per concrete class* strategy requires three commands to include one child (insert into `PERSON`, `BRAZILIAN_CITIZEN` and `CHILD` tables) and two tables to change the `name` and `rg` (`PERSON` and `BRAZILIAN_CITIZEN` tables), while the *one table per leaf class* strategy needs to change two tables both to insert and to update the children's `name` and `rg` (`BRAZILIAN_CITIZEN` and `CHILD` tables). Changing a person's life phase has worse times in the relational schemas generated by the *one table per concrete class* and *one table per leaf class* strategies because both require one exclusion and one inclusion to complete the action, while this action has the same response time as a simple update over the schema generated by the *one table per kind* strategy. Although changing the life phase to adult involves a DELETE and an INSERT in *one table per concrete class* and *one table per leaf class* strategies, the response time of the operation to remove the initially inserted child is greater because it involves deletes in more tables. The *one table per kind* strategy has the worst times for deleting records, which we attribute to checks required to ensure referential integrity in other large tables that have keys of persons.

4.3.2 Object-Oriented Code Ontology (OOC-O) Project

The OOC-O project (AGUIAR, 2021), whose models are available in the OntoUML repository (BARCELOS et al., 2022) under the name of *aguiar2019ooco*, has the purpose of identifying and representing the semantics of the entities present at compile time in object-oriented source code. Its models show the existential dependence of constructs present in object-oriented code, such as `Language`, `Program`, `Code`, `Module`, `Classes`, `Methods`, `Variables`, `Blocks` (scope in which a variable is valid), etc., and the subcategories of each construct when they exist. For example, the subcategories of `Module` – formed by `Physical Module` and `Abstract Module`; the implementable subcategories of `Class` – formed by `Abstract Class` and `Concrete Class`; the subcategories of `Variable` – formed by `Instance Variable`, `Class Variable`, `Parameter Variable` and `Local Variable` (variables declared in the methods). `Programs` are constituted of `Code` that must be associated with `Physical Modules`, which are a composition of `Classes`. A class can be composed of `Methods` and `Variables` (attributes). The models of the OOC-O project are included in Appendix B.

We create the relational schemas provided by *one table per concrete class*, *one table per leaf class* and *one table per kind* strategy to OOC-O models and populated them with:

- 8k classes (about 20% abstract classes and 80% concrete classes);
- 160k methods (about 20% abstract method, 20% concrete method, over 8k constructor methods, 8k destructor methods, 144k accessor methods), and
- 840k variables (40k declared in classes and 800k declared in methods).

The database size was 255.535 MB for the *one table per concrete class* strategy, 150.309 MB for the *one table per leaf class* and 156.965 MB for the *one table per kind* strategy.

Query Answering Performance

In order to assess relative query answering performance considering the OOC-O models, a number of queries were written to retrieve:

1. the names of all named elements;
2. the names of classes, methods and the types of the variables belonging to concrete methods (parameters or declared in the body of methods);

3. the programs and the names of variables (attributes) belonging to concrete classes, that is, without taking into account parameters or variables declared in methods;
4. all modules, indicating whether they are physical or logical;
5. all programs that use a given variable name.

The queries were designed with a similar purpose to those in Section 4.3.1, i.e., to explore different query characteristics. Due to the nature of the model, which features a deep specialization hierarchy, all queries include some polymorphic access. In addition, there is a large number of leaf classes (see Table 11). Because of this, some of the queries were written with sub-queries. Code for all queries is available in Appendix C.2.

Table 11 – Metrics of OOC-O models.

Nº of classes (n)	**Maximum height of the hierarchy (h)**	**Nº of leaf classes (n_l)**	**Nº of kinds (n_k)**
55	5	25	11

Table 12 shows some elementary information of the queries produced for the transformation strategies used. The greater number of polymorphic queries resulted in queries with a greater number of tables. One table was counted more than once if it is used in more than one sub-query. The queries performed for the *one table per kind* strategy require a smaller number of tables to retrieve the requested information and the one table per leaf class strategy required more unions when compared to the other strategies. The *one table per concrete class* strategy used the greatest number of tables because the requested information is distributed among the tables that represent the inheritance hierarchy.

Table 12 – Number of tables, unions and filters for the queries based on different schemas over OOC-O models.

Query	**One table per concrete class**			**One table per leaf class**			**One table per kind**		
	Nº of tables	**Nº of unions**	**N° of filters**	**Nº of tables**	**Nº of unions**	**N° of filters**	**Nº of tables**	**Nº of unions**	**N° of filters**
1	3	2	0	7	6	0	3	2	0
2	12	1	0	9	4	0	3	0	1
3	25	3	0	24	6	0	6	0	1
4	2	1	0	2	1	0	1	0	0
5	21	2	3	14	7	4	6	0	1

Table 13 shows the response times obtained, for the three strategies employed. Query 1 had very similar response times with *one table per leaf class* marginally better. With the exception of Query 1, in all other queries the *one table per kind* strategy does not use `UNION`s and manipulated a significantly smaller number of tables, achieving better results.

Table 13 – Queries response times (averages, in seconds) on OOC-O models.

Query	One table per concrete class	One table per leaf class	One table per kind
1	0.294	**0.285**	0.296 (+3.7%)
2	0.717	0.430	**0.362** (-18.8%)
3	5.651	2.141	**0.982** (-118.0%)
4	0.003	0.003	**0.002** (-50.0%)
5	0.496	0.493	**0.315** (-56.5%)

Performance of Update Operations

We also evaluated some actions to insert, update and delete records, in the following order: first, we insert 100 constructor methods; second, we change the name and the associated class of these methods; third, we reclassify these methods to accessor method; lastly, we delete all the methods initially inserted. Table 14 shows the response time for the proposed updates.

Table 14 – Times required for updates of records representing methods (in seconds).

Operation	One table per concrete class	One table per leaf class	One table per kind
Insert	0.034	0.011	**0.006** (-83.3%)
Update (name and association)	**0.006**	0.011	**0.006** (-83.3%)
Change to accessor method	0.011	0.084	**0.005** (-120.0%)
Delete	**0.135**	0.191	0.203 (33.5%)

Just like the evaluation of schemas generated from the running example model, the *one table per kind* strategy presents better results in general because updates involve only one table in the the resulting schema. The *one table per concrete class* strategy requires six commands to include one constructor method (insert into `CONSTRUCTOR_METHOD`, `INSTANCE_METHOD`, `CONCRETE_METHOD`, `METHOD_MEMBER_FUNCTION`, `NON_OVERRIDABLE_METHOD` and `GENERIC_-METHOD` tables) and only one command to change the `name` and its associated class (`METHOD_-MEMBER_FUNCTION` table), while the *one table per leaf class* strategy needs to change two tables both to insert and to update the name and the associated class (`CONSTRUCTOR_METHOD` and `GENERIC_METHOD` tables). Changing the method's type to accessor has worse times in the relational schemas generated by the *one table per concrete class* and *one table per leaf class* strategies because both require one exclusion and one inclusion to complete the action, while this action has the same response time as a simple update over the schema generated by the *one table per kind* strategy. The *one table per kind* strategy has the worst times for deleting records, which we attribute to checks required to ensure referential integrity in other large tables that have keys of methods.

4.3.3 Limitations

Our aim in this chapter was to show that the *one table per kind strategy* can address the required modeling primitives, while still providing acceptable performance characteristics in comparison with a number of other strategies. The performance analysis was limited to two models, one designed with the purpose of exploring all the modeling primitives (the running example), and another model produced independently by a third-party and published in a model repository. We have not considered the *one table per hierarchy strategy* as it cannot address multiple inheritance, which is required in both models considered. A clear issue for further investigation is the generalization of these performance results, including a large number of models with different characteristics.

The data presented in this section is not intended to indicate the best transformation strategy. There are a large number of factors that need to be considered in a particular application and that are not analyzed here. The data reveals that the relational schema generated by the *one table per kind* strategy has strengths in certain types of queries and database operations, and not so much in others. A system designer must make the necessary system-specific analyzes to indicate which transformation strategy is most appropriate for the system to be developed. The automation of transformation can support the designer in experimenting with the various resulting schemas under realistic loads.

We did not evaluate the performance of the relational schema regarding concurrency or in relation to different DBMSs. Keeping data concentrated in large tables can impact in the concurrency and, consequently, affect database performance. Further, response time assessment can vary significantly according to the chosen DBMS. In fact, we did preliminary analyzes between PostgreSQL and MySQL and obtained different results regarding the response time of some queries. The data were not presented here because this analysis requires a deeper study in relation to the isolation level used, relational schema size, throughput of the service performed in a concurrency situation, among others. This should be the subject of a dedicated study on this theme.

4.4 Empirical Evaluation of Query Understanding

In order to assess the implications of the *one table per kind* strategy with respect to the usability of the resulting schema, we decided to contrast it with one other strategy only, for a number of reasons. First, adding several schemas would make the experiment long, which could affect negatively the participation in the experiment. Second, the experiment could become overly complex, in particular because we would need explanations for some of the more complex schemas, e.g., those resulting from *one flattened table per leaf class*. Third, learning effects could be compounded if a subject is exposed to several schemas.

Therefore, we focused our efforts on identifying which strategy would best serve in the comparison. First of all, the *one table per hierarchy* strategy was excluded as it cannot be applied to our running example because it has multiple inheritance. We established that the contrasted strategy should still emulate inheritance, differently from *one table per kind*. Hence, *one table per leaf class* was excluded. The *one table per class* strategy was considered, but excluded since it generates an excessive number of tables, and the effect of the number of tables solely could overshadow other factors. Hence, we have opted for the *one table per concrete class* strategy; it still emulates some inheritance, but generates fewer tables.

4.4.1 Materials and Preparation

We use as a starting point the running example with a minor modification, as shown in Figure 15. The sole modification is that the generalization set for nationality is marked `disjoint` instead of `overlapping`, which avoids the creation of an additional `NATIONALITY` table. This table would only have created some noise in the experiment, since the query used does not consider nationality altogether.

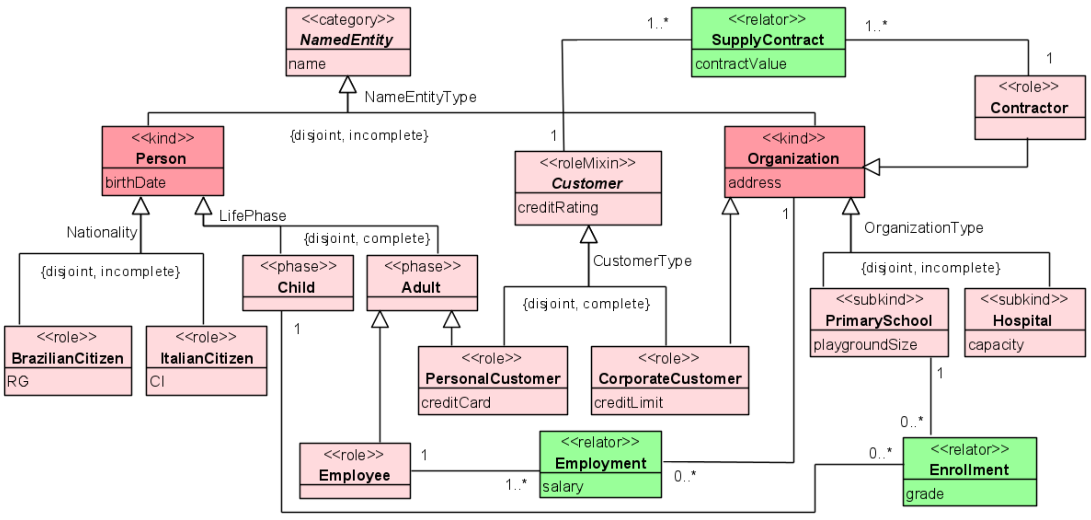


Figure 15 – Running example with disjoint nationality

Figure 16 (a) presents the relational schema generated by the *one table per concrete class* strategy and Figure 16 (b) the relational schema generated by the *one table per kind* strategy.

Several invitations to an online survey, prepared with Google Forms, were sent to online SQL mailing lists (pgsql-sql, SQLite forum, MariaDB discuss and mysql-br). The invitations stated that the purpose of the experiment was to investigate the understanding and usability of different relational schemas, without making any mention of strategies for transformation. It was made clear that participation was voluntary and the responses would be recorded anonymously, not revealing the identity of the participant.

The experiment was organized into two phases: phase I requested demographic data to be used as a control, especially concerning level of expertise; phase II investigated the ability to interpret queries written for the schemas generated with the two strategies and the perceived

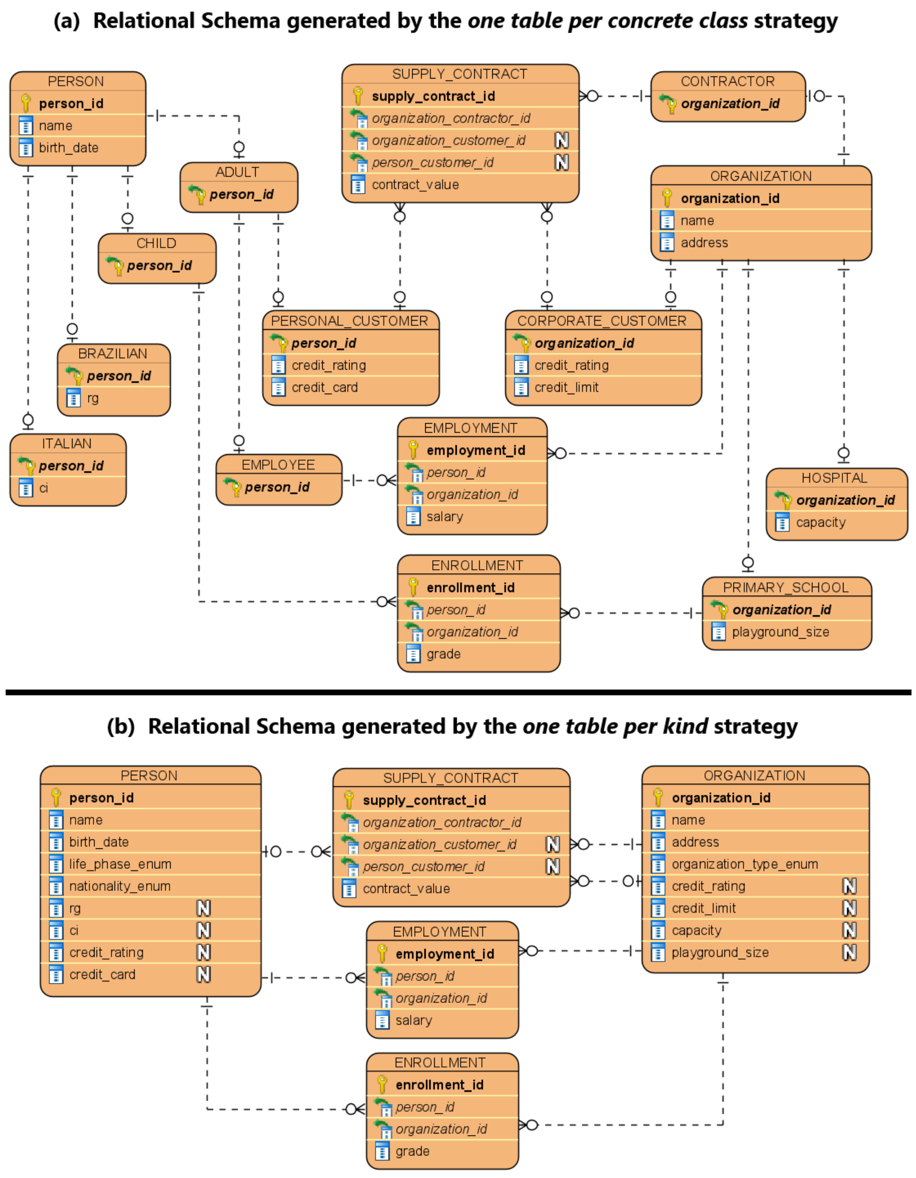


Figure 16 – Schemas used in the empirical evaluation

ease of understanding. Some information was provided to the subjects about the relational schemas, such as the purpose of tables and columns, especially concerning discriminators columns. No information about the conceptual model and the transformation strategy to obtain the relational schema was presented.

Phase II of the experiment consists in three questions about the queries. Question 1 presents the query of the Listing 4.1 and the schema of Figure 16 (a) (*one table per concrete clas*) and asks for the query's purpose. Question 2 presents the query of the Listing 4.2 and the schema of Figure 16 (b) (*one table per kind*) and requires its purpose. The purpose of the queries are the same for both relational schemas. The order of the questions was defined randomly by the tool used, to cancel the effect of learning. So, half of the subjects were asked a question about the query written for the *one table per concrete class* strategy first, and only then for the query written for the *one table per kind* strategy; the other half were asked these questions in reversed order. At the end of this part of the experiment, we asked the participants (Question 3) which query was easier to interpret ("Which of the two queries presented to you before was easier to understand?").

Listing 4.1 – Query on schema generated by the*one table per concrete class* strategy

```sql
SELECT *
FROM   person p
JOIN   child c
       ON  p.person_id                    = c.person_id
JOIN   enrollment e
       ON  c.person_id                    = e.person_id
JOIN   primary_school ps
       ON  e.organization_id              = ps.organization_id
JOIN   organization o
       ON  ps.organization_id             = o.organization_id
JOIN   corporate_customer cc
       ON  o.organization_id              = cc.organization_id
JOIN   supply_contract sc
       ON  cc.organization_id             = sc.organization_customer_id
JOIN   contractor cont
       ON  sc.organization_contractor_id = cont.organization_id
JOIN   organization cont_org
       ON  cont.organization_id           = cont_org.organization_id
JOIN   hospital h
       ON  cont_org.organization_id       = h.organization_id
```

Listing 4.2 – Query on schema generated by the *one table per kind* strategy

```sql
SELECT *
FROM   person p
JOIN   enrollment e
       ON  p.person_id                      = e.person_id
JOIN   organization o
       ON  e.organization_id                = o.organization_id
       AND o.organization_type_enum         = 'PRIMARYSCHOOL'
JOIN   supply_contract sc
       ON  o.organization_id                = sc.organization_customer_id
JOIN   organization cont_org
       ON  sc.organization_contractor_id   =  cont_org.organization_id
       AND cont_org.organization_type_enum = 'HOSPITAL'
WHERE  p.life_phase_enum                    = 'CHILD'
```

4.4.2 Results

In total, 20 people participated in the experiment. 85% reported being IT professionals; 15% professors/researchers; no students participated in the experiment. Only 10% of the participants reported 1 to 3 years of experience with databases, 90% reported over 5 years experience.

A score from 0 to 10 was assigned to the answers to Questions 1 and 2. We have considered 'correct' those answers that received a score equal to or greater than 8. For an answer to be considered correct, the participant must say, at least, that the query "lists children enrolled in primary schools, which have a supply contract with a hospital". Any answer that does not mention somehow “children enrolled in primary schools” and that “school has a supply contract with hospital” was given a score lower than 8.

We obtained the following results (shown in Figure 17): 50% of the participants interpreted both queries correctly; 10% interpreted only one of the queries correctly (5% interpreted only the query with the schema generated by the *one table per kind* strategy correctly and 5% interpreted only the query with the schema generated by the *one table per concrete class* strategy correctly); 40% of the participants scored less than 8 for both queries. Based on these results, we can conclude that the relational schema generated by our transformation strategy did not affect query understanding relative to the relational schema generated by the *one table per concrete class* strategy.

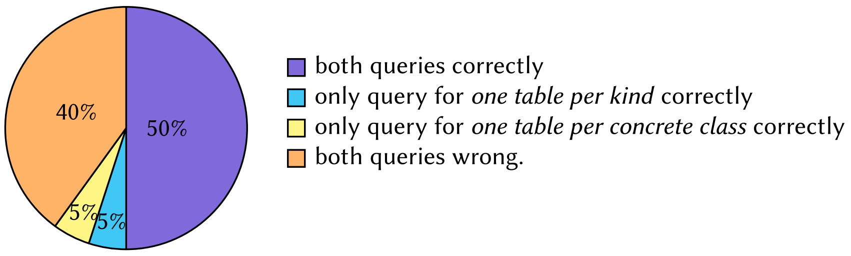


Figure 17 – Query interpretation correctness

Despite the lack of differences concerning query interpretation correctness, we obtained the following result regarding the overall reported query preference, shown in Figure 18: *none* of the participants preferred to perform interpretations on the query on the schema generated by the *one table per concrete class* strategy; 30% were indifferent and 70% preferred to interpret queries on the schema generated by the *one table per kind* strategy. Thus, our experiment shows a striking result in favor of the perceived ease of understanding of the schema generated with *one table per kind*.

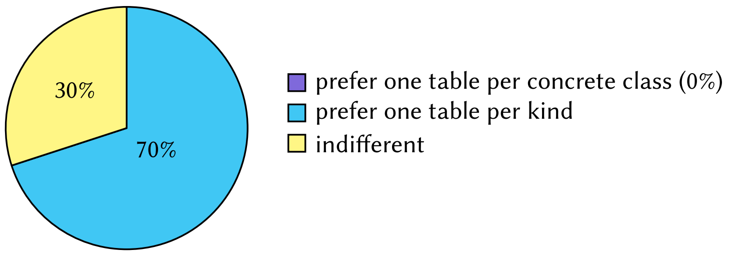
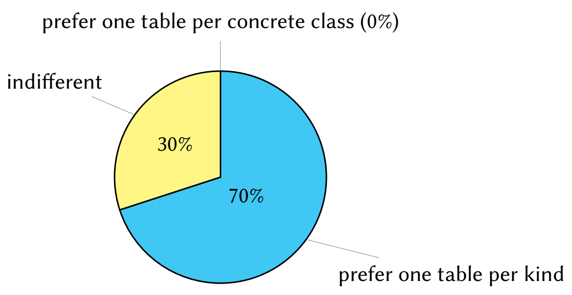
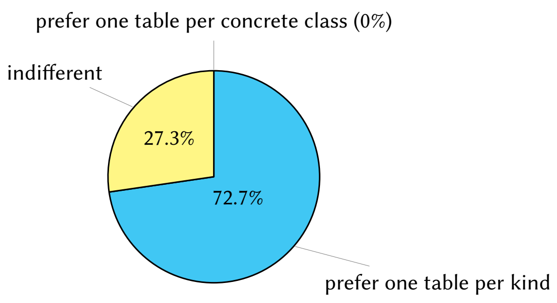


Figure 18 – Query interpretation preference (all respondents)

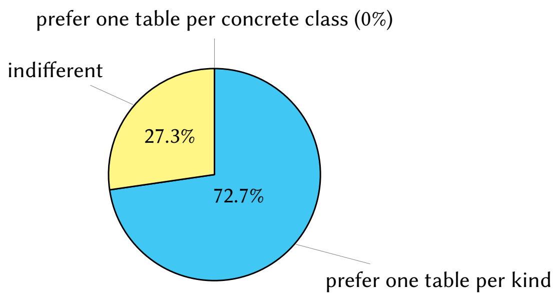
Figure 19 shows the relational schema preference of our participants in relation to the correctness of the query interpretation. The preference is exactly the same when observing only the participants who got both interpretations right (Figure 19 (a)), and equal for those who correctly interpreted just one query right, regardless of whether another query was correct (Figure 19 (b) and (c)). The percentage of participants who preferred to interpret queries for the schema generated by the *one table per concrete class* strategy increases when analyzing only those who answered incorrectly (Figure 19 (d)).



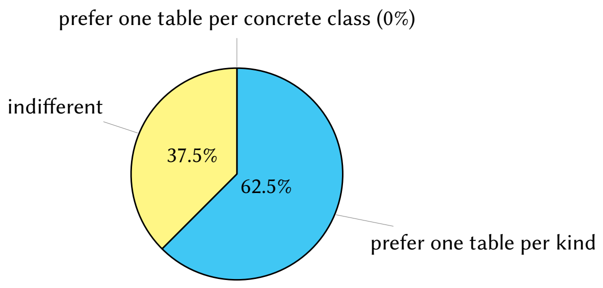
(a) interpreted both queries correctly



(b) interpreted one table per kind correctly



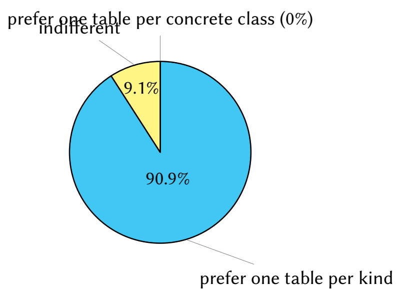
(c) interpreted one table p. concrete class correctly



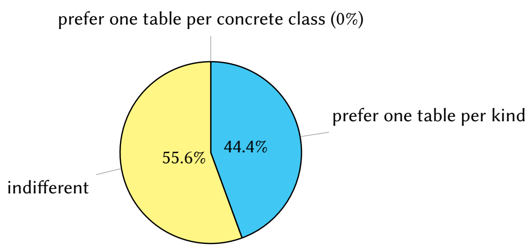
(d) interpreted both queries incorrectly

Figure 19 – Query interpretation preference (breakdown)

Figure 20 shows a break-down of the query preference considering the order in which the schemas were presented to participants. Figure 20 (a) shows that a vast majority of participants that were exposed to the *one table per kind* schema first preferred it. Figure 20 (b) reveals that more participants became indifferent when shown the *one table per concrete class* first. A more in-depth experiment would be required to explain this effect.



(a) *one table per kind* first



(b) *one table per concrete class* first

Figure 20 – Query interpretation preference regarding the schema displayed first

4.4.3 Limitations

We defined a query for the experiment involving only simple SQL constructs, to maximize the target audience, therefore we did not work with queries with a higher degree of difficulty, such as polymorphic queries or queries that required the use of LEFT JOIN or subqueries. We hypothesize that these results obtained for simple queries would transfer to more complex queries, but that remains to be confirmed. We are aware that the design space for SQL queries is enormous, and, hence, a comprehensive assessment of the various aspects affecting query and schema design and consequent ease of use is an important theme for future work.

For the reasons discussed in the beginning of Section 4.4, we have limited the experiment to a comparison between *one table per kind* and one other strategy (*one table per concrete class*). Comparing more approaches is a topic for further studies.

In the experiment, due to the low number of participants we were not able to cover all levels of participants experience in this experiment. Most participants can be considered non-beginners, and most can be considered experienced SQL users. Therefore, it was not possible to assess the understanding of queries in relation to the experience of the participants. It is plausible that beginners could have revealed more differences with respect to query interpretation correctness, and this is an issue for further study.

As the data were collected remotely through Google Forms, it was not possible to collect information about the time used for the participants answer the proposed questions. The average time spent on each question could provide further insight into relative ease of use.

4.5 Related Work

Rybola and Pergl wrote several papers about transforming OntoUML models into relational models (RYBOLA; PERGL, 2016b; RYBOLA; PERGL, 2016c; RYBOLA; PERGL, 2016a; RYBOLA; PERGL, 2017), which are summarized in a Ph.D. thesis (RYBOLA, 2017). The main feature of their work is an object-relational transformation process anchored in Model-Driven Development (MDD) principles. Their approach performs the transformation in three steps: (1) from OntoUML to UML; (2) from UML to Relational Database Model; (3) from Relational Database Model to the relational schema. In his Ph.D. thesis, Rybola (2017) proposes the transformation of non-sortals with a pattern that introduces a table for each class. In this sense, his approach approximates *one table per class*, but produces even more tables due to the patterns employed to address the relation between the non-sortal and sortal hierarchies. This exacerbates the performance issues when accessing or inserting an object.

Some ontology-based transformation processes focus on the relational realization of computational OWL ontologies. We have transformations such as those of Afzal et al. (2016) and Vyšniauskas et al. (2011), which implement the *one table per class* strategy, thus, transforming each OWL class to a relational table; without considering any additional ontology constructs in the transformation process.

There are also a number of programming-level object-relational mapping tools. A popular one is Hibernate[2] (BAUER; KING, 2005), which targets the Java programming language. It uses explicit annotations in Java code to guide the creation of the relational schema and the corresponding object access through SQL queries, making the database largely transparent to the developer. Regarding the inheritance hierarchy, Hibernate is capable of performing the following transformation strategies, named as: *Mapped Superclass*, *Single Table*, *Joined table*, and *Table per Class*. These strategies are referred to in this work respectively as: *one flattened table per leaf class*, *one table per hierarchy*, *one table per concrete class*, and *one flattened table per concrete class*. No support for multiple inheritance or dynamic classification is provided, as these are not supported in the language.

Guizzardi et al. (2019) present OntoUML model transformation rules with the purpose of reducing the number of classes visualized in the model. The purpose of that work is to produce a more abstract version of the model for communication and understanding purposes, not for realization of the conceptual model. Some rules presented in our work are similar to the rules presented by Guizzardi et al. (2019) because we act on the same UFO categorization principles, but with different effects on the target model. For example, the names of the classes that are lifted are used to name associations ends. Here, in contrast, classes lifted are identified by discriminator attributes, as the information on whether a particular entity instantiates the lifted subclass cannot be lost in the database.

Over the years, a number of authors have compared the various object-relational transformation strategies in terms of system infrastructure (performance and storage) and relational schema design (understanding and maintainability). Among these, Keller (1997) points out the infrastructure and design "forces" that govern the development of a relational schema, as well as some characteristics used in the application design, such as polymorphism. The author also exposes the consequences of using the strategies, which also is done by Fowler (2002) when identifying the strengths and weaknesses of each strategy. Ambler (1997) performs a brief comparison between the strategies and is concerned with the practical differences between the relational and the object-oriented paradigms. Philippi (2005) establishes the consequences of transformation strategies in relation to the infrastructure and design aspects of the relational schema when the inheritance hierarchy is associated with other classes of the model along with association cardinality. In turn, Torres et al. (2017) does not perform a systematic comparison between the strategies, but identifies their adoption in the various object-relational transformation tools. Considerations of the authors in favor of their solutions are based on their (subjective) experiences, while we present some quantitative data to support the comparison between strategies.

[2] <https://hibernate.org/>

4.6 Final Considerations

In addition to the dominant strategies we have discussed, there are approaches which use the distinction between abstract and concrete classes, with impact on performance characteristics. For example, *one flattened table per concrete class* is a variant of *one table per concrete class* in which abstract classes are flattened out. Since flattening of abstract classes reduces the height of the hierarchy, this strategy has the potential of improve the performance of retrieving and inserting an entity. Nevertheless, that performance is still much dependent on the size of the concrete class hierarchy. Further, dynamic classification performance remains a challenge in this approach. By identifying the ontological meta-properties of classes in the source conceptual model, we are able to better navigate performance tradeoffs, beyond what can be achieved with the abstract–concrete distinction. For example, strategies such as *one table per rigid sortal* become possible, approximating *one flattened table per concrete class* in terms of performance but circumventing its difficulty with dynamic classification. We have shown that the *one table per kind* strategy has characteristics that differ from the dominant approaches in the literature. Further, it supports source conceptual models with multiple inheritance, orthogonal and overlapping hierarchies and dynamic classification.

Problems with multiple inheritance in *one table per hierarchy* do not appear in *one table per kind* because according to UFO definitions (GUIZZARDI, 2005) there is no multiple inheritance of kinds. Multiple inheritance of non-kind sortals (subkinds, phases and roles) does not pose a problem, as discriminators identify the instantiated classes. Multiple inheritance of non-sortals creates no problems because they are flattened into kinds. Problems with orthogonal hierarchies and overlapping generalization sets in *one table per leaf class* also do not arise as a consequence of the transformation strategy. Kinds tables are where the entities primary keys are placed, and hence there is no problem with the same entity being represented in several tables. Flattening of non-sortals poses no problem in this scenario. In the lifting of non-kind sortals, orthogonal hierarchies and overlapping generalization sets are, not unlike multiple inheritance, reflected in discriminators in the kind table. Dynamic classification is supported naturally as reclassification is simply change in discriminators. This is not the case with *one table per class*, *one table per concrete class*, *one flatted table per concrete class* and *one flatted table per leaf class*, which require deletion and insertion, posing a problem for referential integrity.

We have shown that the *one table per kind* approach has acceptable performance characteristics when contrasted with other approaches. Although there are a number of important limitations to the performance study conducted here (as discussed in Section 4.3.3), it is indicative of the feasibility of the *one table per kind* strategy from the perspective of performance. We emphasize that automating the generation of schemas for the various strategies can be used for targeted performance evaluation considering specific applications (with specific queries and with a specific DBMS solution in mind), in the presence of strict performance requirements.

5 High-Level Data Access

Despite the various benefits of object-relational 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. This chapter addresses these challenges. We instrument the transformation process so that as it advances, at each application of an operation, a set of traces from source to target model is maintained, 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 (providing thus support for hypothesis 3).

This chapter is further organized as follows: Section 5.1 presents the motivation to access data in terms of the conceptual model (domain ontology) instead of the relational schema. Section 5.2 presents the basic concepts of the Ontology-Based Data Access (OBDA) technology employed. Section 5.4 shows the adaptation of our transformation process to maintain traces between the conceptual model and the relational schema and synthethize the required OBDA mapping. Section 5.5 presents a comparison of the performance time of queries generated by a tool that implements OBDA and manually on different relational schemas. Section 5.6 presents some related work and finally the Section 5.7 presents some final considerations.

5.1 Motivation

In order to exemplify the differences in data access consider the user is interested in a report with "the Brazilian citizens who work in hospitals with Italian customers". This need would require two different queries depending on the transformation strategy. Listing 5.1 shows the query for the *one table per concrete class* strategy and Listing 5.2 for *one table per kind* strategy.

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

Listing 5.1 – Query on the schema of Figure 4, 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 5.2 – Query on the schema of Figure 14, 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'
```

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 (as discussed in the previous chapter). 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. It should be independent of the particular transformation strategy and allow the expression of data access in terms of the conceptual model, instead of a particular schema.

5.2 Ontology-Based Data Access (OBDA)

We can leverage Ontology-Based Data Access (OBDA) to offer data access at a higher-level of abstraction. OBDA (POGGI et al., 2008) provides integration and access to the relational schema through queries that are specified taking into account the ontological conceptual model rather then the relational schema. Figure 21 depicts an overview of the OBDA workflow (XIAO et al., 2019). The approach performs all required data processing to translate data between technological spaces: at the lowest level, data storage using relational databases (accessed through SQL) and at the highest level, information representation through a computational ontology (using SPARQL). Each SPARQL query is automatically rewritten to SQL queries that are executed at the database. Results of the query are then automatically mapped back to triples and consumed by the user using the vocabulary established at the ontology. The integration layer is structured through graphs, kept virtual, in order to incorporate ontology knowledge. For this reason, the OBDA paradigm is also known in the literature as the *Virtual Knowledge Graph* (VKG) (XIAO et al., 2019) paradigm.

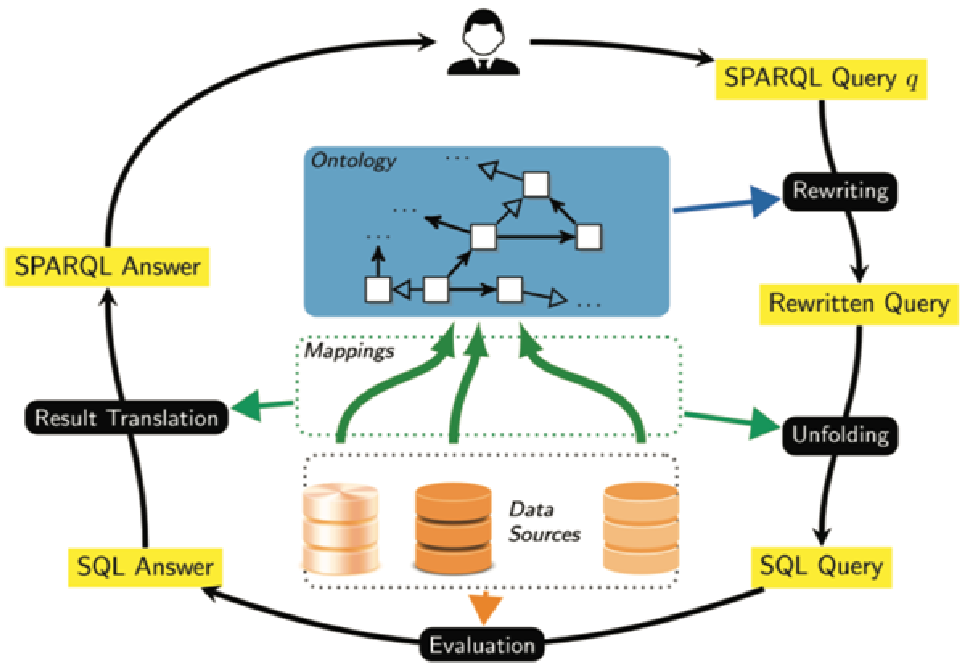


Figure 21 – OBDA workflow (XIAO et al., 2019).

To retrieve the data as a conceptual representation of the problem domain and not as they are structured in the relational database, OBDA requires the specification of mapping assertions between the ontology and the data source (represented by the green arrows in Figure 21). Each mapping assertion consists of an association of an ontology concept or property in an SQL query. The overall OBDA specification is defined in (XIAO et al., 2018) as $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$, where:

- $\mathcal{O}$ - is the computational ontology, and must be written in the OWL language. This layer can be seen as a set of axioms expressing the formal description of the domain of interest.
- $\mathcal{S}$ - is the schema of the data source. In our case, it is a relational database.
- $\mathcal{M}$ - is the set of assertions for mapping the data between its physical data source and the elements of the ontology, i.e., specifies how A-Box assertions (or RDF triples) can be created in accordance with the data of the relational schema.

Let x be a list of arguments to be returned by a query, then mapping $\mathcal{M}$ can be understood as a set of assertions in the form $\varphi(x) \rightarrow \psi(x)$, where $\varphi(x)$ is a query over $\mathcal{S}$, and $\psi(x)$ the same query over the alphabet of $\mathcal{O}$. The tuples returned by $\varphi(x)$ are filled in $\psi(x)$ respecting rules in ontology. Thus, the mapping creates a bridge between $\mathcal{O}$ and $\mathcal{S}$, and the OBDA solution can work as a middleware to connect disparate structures.

The standard for representing the mapping $\mathcal{M}$ is defined by the W3C as RDB to RDF Mapping Language (R2RML)[1]. Generating R2RML mappings is not a trivial task, as a consequence, many systems that implement OBDA technology have their own approach to carry out the mapping between the ontology and the relational schema, which is the case of Ontop (CALVANESE et al., 2017), the OBDA solution adopted here. Ontop is an open-source OBDA framework developed at the Free University of Bozen-Bolzano, released under the Apache 2 License. Ontop has good performance in query answering, supports almost all the features of SPARQL 1.1, R2RML and OWL 2 QL and SPARQL entailment (XIAO et al., 2020).

5.3 Tracing Flattening and Lifting

As discussed in Section 3.2, 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 22, creates one trace from this class to `Person` and another to `Organization`.

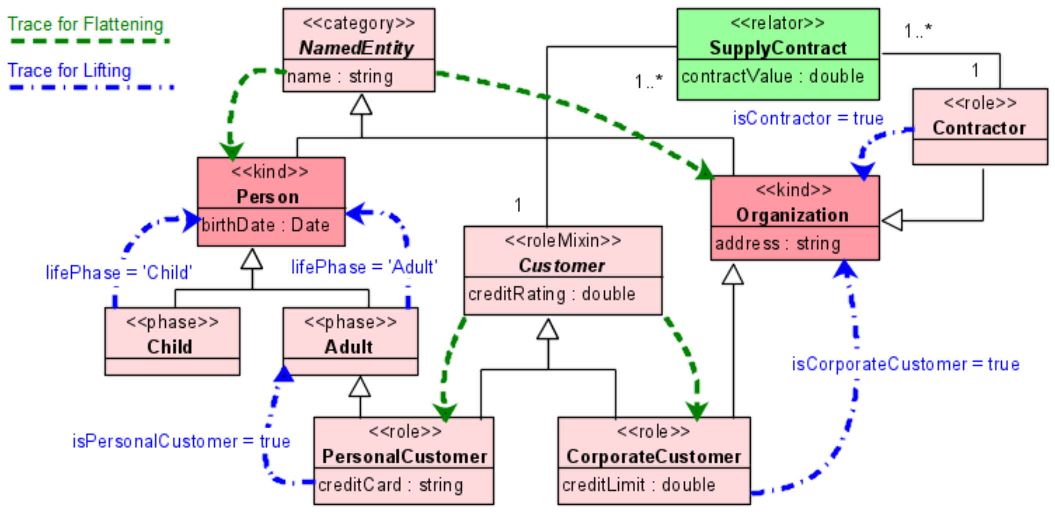


Figure 22 – Tracing example for each execution of flattening and lifting.

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

[1] https://www.w3.org/TR/r2rml/

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` (traces as a result of lifting are represented with blue arrows in Figure 22). However, not every `Person` instance is an instance of `Child`. The added filter thus requires the discriminator `lifePhase='Child'`.

We have employed a trace table following the model shown in Figure 23. A `TraceTable` is produced in each application of the transformation and consists of a collection of `TraceSets` for each source class in the model. Each `TraceSet` identifies the set of `Traces` to target classes. The transformation begins by initializing the `TraceTable` with one `TraceSet` for each source class in the model each of which containing a single `Trace` to the same class. As the transformation progresses, traces in a trace set are updated. Required `Filters` are added in case of lifting. We detail below the changes in `TraceTable` on the conceptual model of Figure 22 when the flattening and lifting operations are applied.

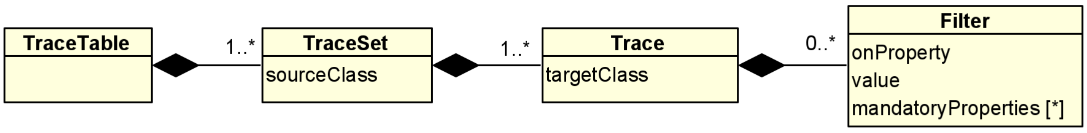


Figure 23 – Trace table structure.

Figure 24 shows the trace sets contained in the initial `TraceTable` as an object diagram. For brevity, the traces for `Person`, `Organization` and `SupplyContract` classes will be omitted as they remain unchanged throughout the process to apply the *one table per kind* approach, which we adopt in this example, as it applies both flattening (with an intermediate result that reflects *one table per concrete class*) and lifting.

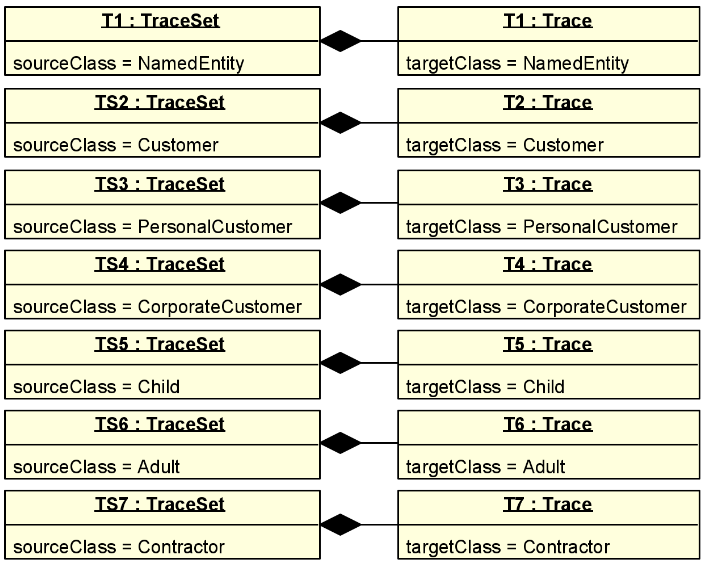


Figure 24 – Initial trace table.

In the conceptual model shown at Figure 22, NamedEntity will be flattened to the Person and Organization classes. This means that NamedEntity will be identified by the union of all instances of Person and Organization. Thereby, is necessary to replace the traces that currently refer to the flattened class with those referring to the subclasses in the flattening operation. The same is true for the Customer class and its subclasses. The result of this updates are shown in gray in Figure 25. (If we are following the *one table per concrete class* approach, this would be the final state of the trace table.)

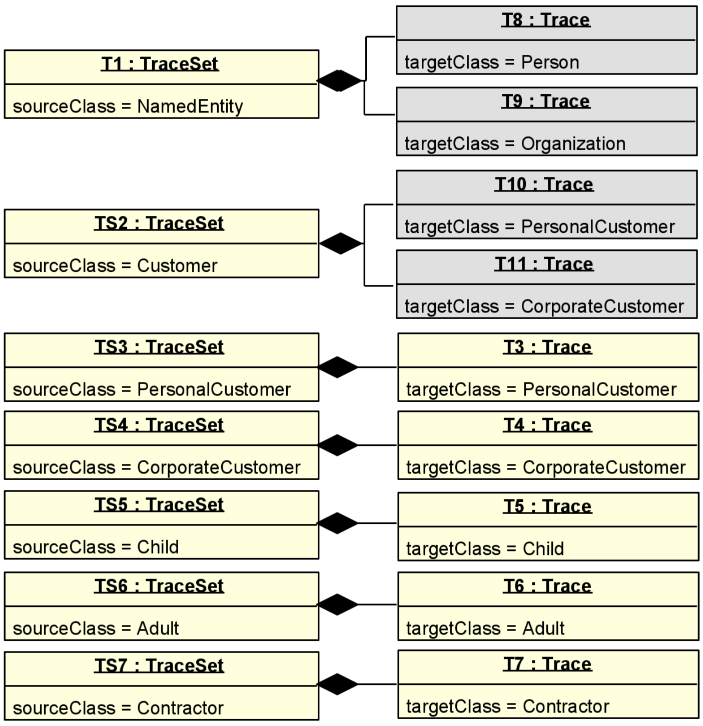


Figure 25 – Trace table after the flattening of NamedEntity and Customer.

When lifting is performed on a subclass (to carry out *one table per kind* strategy at this point in the transformation), the traces currently referring to the lifted subclass are updated with traces to the superclasses. Filters are added according to the discriminators required as discussed in Section 3.1.2. The set of mandatory properties that were lifted are added to the filter. For example, the lifting of the PersonalCustomer class indicates that it becomes identified in the Adult class when the isPersonalCustomer attribute is equal to true, requiring the filling of the creditRating and creditCard attributes. As the PersonalCustomer class no longer exists in the resulting model, every reference to it in the trace table is updated to the target class of the lifting process (in this case, Adult). Thereby, the PersonalCustomer and Customer source classes that traced PersonalCustomer now trace Adult (including filters). Figure 26 shows the result of the lifting of PersonalCustomer, CorporateCustomer and Contractor, whose lifting preconditions are established. After lifting these classes, the trace for the source class PersonalCustomer is updated to Adult. The trace for source class CorporateCustomer is updated to Organization. The traces of source class Customer is also updated, as it targeted the lifted classes. It now includes a trace to Adult and a trace to Organization, since the

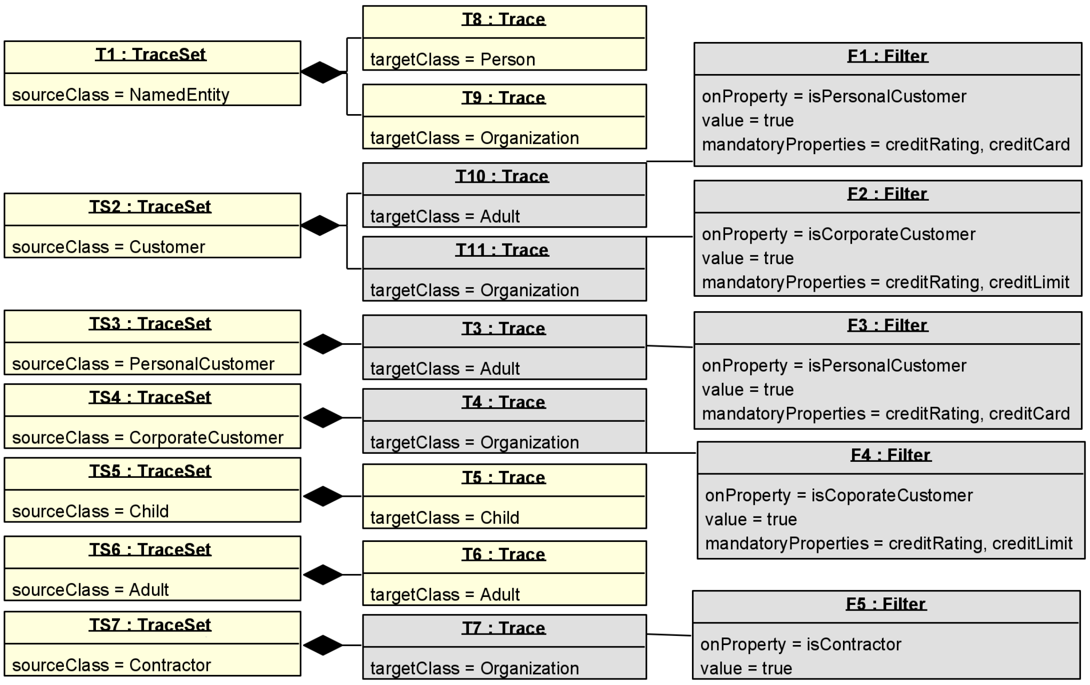

T1 : TraceSet
sourceClass = NamedEntity
T8 : Trace
targetClass = Person
T9 : Trace
targetClass = Organization
TS2 : TraceSet
sourceClass = Customer
T10 : Trace
targetClass = Adult
T11 : Trace
targetClass = Organization
TS3 : TraceSet
sourceClass = PersonalCustomer
T3 : Trace
targetClass = Adult
TS4 : TraceSet
sourceClass = CorporateCustomer
T4 : Trace
targetClass = Organization
TS5 : TraceSet
sourceClass = Child
T5 : Trace
targetClass = Child
TS6 : TraceSet
sourceClass = Adult
T6 : Trace
targetClass = Adult
TS7 : TraceSet
sourceClass = Contractor
T7 : Trace
targetClass = Organization
F1 : Filter
onProperty = isPersonalCustomer
value = true
mandatoryProperties = creditRating, creditCard
F2 : Filter
onProperty = isCorporateCustomer
value = true
mandatoryProperties = creditRating, creditLimit
F3 : Filter
onProperty = isPersonalCustomer
value = true
mandatoryProperties = creditRating, creditCard
F4 : Filter
onProperty = isCoporateCustomer
value = true
mandatoryProperties = creditRating, creditLimit
F5 : Filter
onProperty = isContractor
value = true

Figure 26 – Trace table after the lifting of `PersonalCustomer`, `CorporateCustomer` and `Contractor`.

previously traced `PersonalCustomer` and `CorporateCustomer` have been lifted to those classes. The trace for `Contractor` is updated to `Organization`. In all cases filters are added to identify the subclass in the superclass.

Finally, Figure 27 shows the lifting of `Child` and `Adult`. Note that since `Adult` already had a filter (in the trace set of `PersonalCustomer`), this filter is preserved, and the new one is added (`lifePhase=ADULT`). This is because the final conditions that must be imposed on the target class are the conjunction of these filters. At this point, no further operations are applicable in the *one table per kind* approach and we have the final state of the trace table for translation to the relational schema. Each class in the set of target classes referred to in the trace table corresponds to a table in the resulting relational schema.

5.4 Synthesizing High-Level Data Access

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. Having $\mathcal{O}$ as the user-provided ontology and the transformation process of $\mathcal{O}$ to produce $\mathcal{S}$, $\mathcal{M}$ can be obtained by inspecting the trace tables discussed in the previous section. With the final set of traces, we are able to synthesize the OBDA mapping for Ontop (CALVANESE et al., 2017).

Although the source models we consider are specified here with UML (or OntoUML),

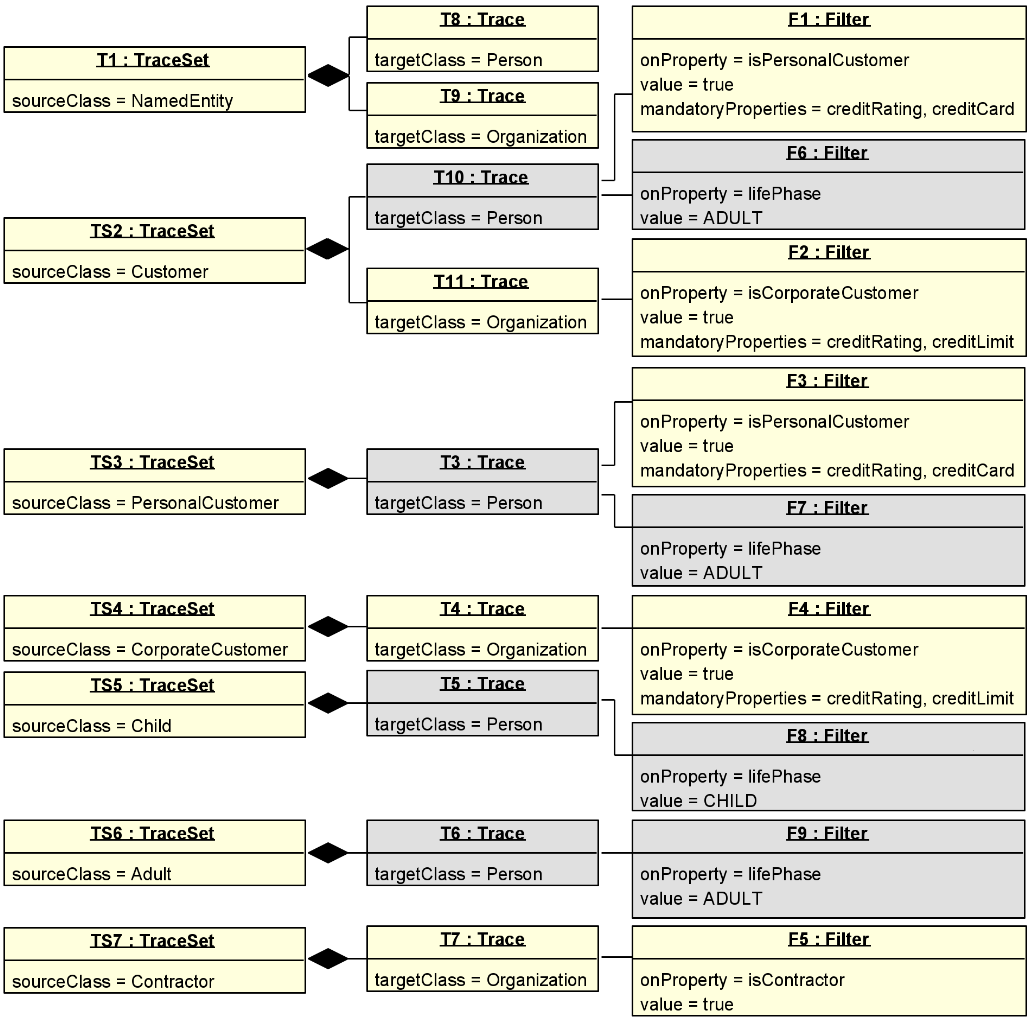


Figure 27 – Trace table after the lifting operation on the generalization set with `Child` and `Adult`.

a transformation to OWL is used as part of the overall solution as required for Ontop; 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[2] (also implementing the *one table per kind* cf. Chapter 3.)

For each tracing path from `TraceTable` (shown in Figure 27) 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 the 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`). (The complete mapping is in Appendix F).

Listing 5.3 shows the Ontop mapping generated for the kind `Person`. Given the absence of flattening and lifting, the mapping establishes the straightforward correspondence of a

[2] <http://purl.org/guidoni/ontouml2db>

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 5.3 – 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
```

Listing 5.4 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 5.4 – 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 5.5 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 is 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.

Listing 5.5 – OBDA mapping for the lifted ItalianCitizen class 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'
```

5.5 Performance of Data Access

In order to evaluate the performance of data access in our approach, we create and randomly populate the databases generated by the *one table per concrete class* and *one table per kind* strategies (corresponding to figures 4 and 14, respectively). 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.

We used the same queries and database settings of Section 4.3. We use the Ontop 2.0.3 plugin for Protégé 5.5.0. The SPARQL queries can be found in Appendix D. Table 15 shows the results obtained, including the overhead for the queries that were generated automatically.

Table 15 – 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	**0.219**	0.232 (+5.9%)	0.251	0.275 (+9.6%)
2	0.272	0.290 (+6.6%)	**0.245**	0.281 (+14.7%)
3	0.623	0.641 (+2.9%)	**0.585**	0.932 (+59.3%)
4	0.041	0.043 (+4.9%)	**0.010**	0.022 (+120.0%)
5	0.192	0.207 (+7.8%)	**0.182**	0.193 (+6.0%)

In the *one table per concrete class* strategy, the performance of the automatically transformed queries is similar to the performance of the manually written queries. In the *one table per kind* strategy, significant overhead is only imposed for queries 3 and 4. 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. Ontop adds that check to the query, in addition to several `IS NOT NULL` checks. As a result, the Ontop queries are 'safer' than the manual ones. Removing the additional checks from the rewritten queries reduces the overhead as shown in Table 16. Regardless of the overhead, the values in bold (for queries 2, 4 and 5) show cases in which the (optimized) Ontop queries for *one table per kind* still outperform the manually written queries for *one table per concrete class.*

Table 16 – Performance comparison with Ontop query optimizations (in seconds).

Query	**One Table per Kind**	
	Manual Query	**Ontop Query**
1	0.251	0.257 (+2.4%)
2	0.245	**0.248** (+1.2%)
3	0.585	0.910 (+55.6%)
4	0.010	**0.020** (+100.0%)
5	0.182	**0.187** (+2.8%)

5.6 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 (SEQUEDA; MIRANKER, 2013) and D2RQ (BIZER; SEABORNE, 2004), facilitate the *reverse engineering* of a high-level representation model from relational schemas. Ultrawrap (SEQUEDA; MIRANKER, 2013) 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 (BIZER; SEABORNE, 2004) also allows access to relational databases through SPARQL queries. It supports automatic and user-assisted modes of operation for the ontology production. Virtuoso (ERLING; MIKHAILOV, 2007) 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 (JIMÉNEZ-RUIZ et al., 2015; MEDEIROS et al., 2015) 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. (2018) propose 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 increase data access abstraction to the reference ontology level.

The Hibernate Query Language[3] (HQL) is an object-oriented query language used to access data stored in relational databases using Hibernate. Unlike the SQL language that

[3] <https://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html>

operates on tables, HQL operates on objects and their properties, with the ability to support inheritance and aggregation. Although this is some form of higher-level data access, as stated in Section 4.5, Hibernate is limited to the object-oriented programming language it operates on (which among other things does not support multiple inheritance, orthogonal hierarchies and dynamic classification).

5.7 Final Considerations

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[4]; 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 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.

[4] <http://purl.org/guidoni/ontouml2db>

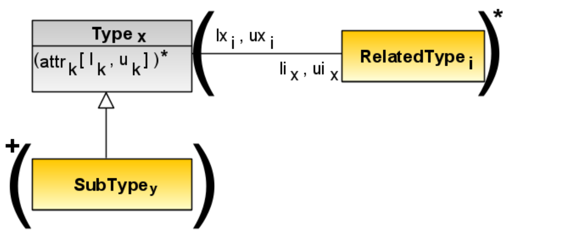
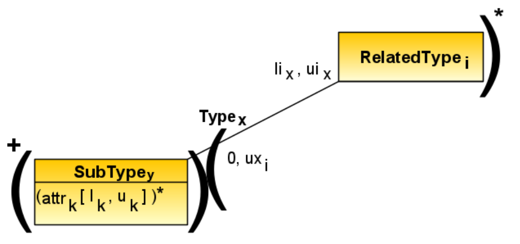
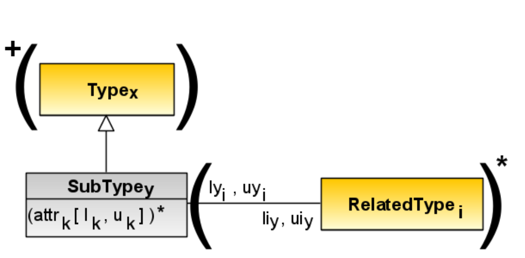
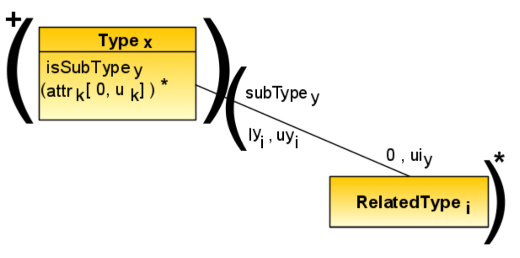
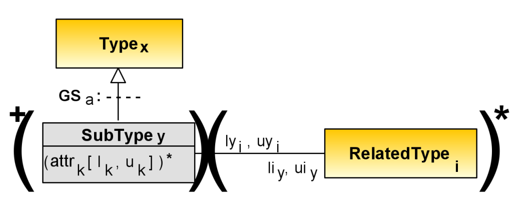
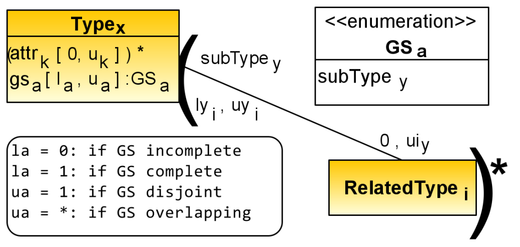
6 Preserving Conceptual Model Semantics

These chapter identifies the constraints that are lost in the transformation of structural conceptual models to relational schemas, independently of the class-to-table transformation strategy applied. Our approach is based on the primitive refactoring operations of *flattening* and *lifting* of classes that can account for the transformation of a specialization hierarchy into tables according to the class-to-table transformation strategies seen in Section 2.1. We show how to incorporate the generation of the required constraints along a transformation of a conceptual model into a relational schema. 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 invariants that are applicable to the resulting schema using the set of traces. These invariants are implemented as triggers, detecting when data that violates the conceptual model is introduced in the database. A fully automated implementation of the transformation is provided.

6.1 Missing Constraints due to Flattening

As can be seen in the "Flattening" line of Table 17 , the flattening of the superclass to all its subclasses does not generate integrity issues regarding superclass attributes, because they are propagated in the same way as they are defined in the superclass. However, associations with other classes (which are not subclasses) are redefined to suit the new classes arrangement, generating distortions of what is allowed in the resulting model in relation to the original model. The missing constraint are: (**MC1**), in the presence of an association in the source model with an association end attached to the flattened superclass and lower bound $lx_i = 1$, we must ensure that at least one instance of the target subclasses is related to an instance of the $RelatedType_i$ through the associations that were introduced in the flattening operation, in order to enforce the original lower bound cardinality. In this case, for a particular `SupplyContract`, there is a related `PersonalCustomer` or a `CorporateCustomer`; and (**MC2**), in the presence of an association in the source model with an association end attached to the flattened superclass and upper bound $ux_i = 1$, we must ensure that an instance of the superclass is related through the associations that were introduced in the flattening operation *to at most one instance of the union of all subclasses*. In this case, considering both conditions, an instance of `SupplyContract` is related to *exactly one instance in the union of* `PersonalCustomer` and `CorporateCustomer`.

Table 17 – Revisiting the flattening and lifting operations.

		Source Graph	Target Graph
Flattening			
Lifting	Single Generalization		
	Generalization Set		

6.2 Missing Constraints due to Lifting

Lifting of Attributes. As seen in Table 17, the lifting of attributes has the following consequences: (**MC3**) mandatory attributes are now optional, a situation which, although necessary—since not all instances of the superclass are instances of the lifted subclass—is inadmissible for instances of the lifted subclass according to the original model. The example model after lifting of the generalization set admits the possibility that a personal customer represented as an instance of `Person` is assigned no value for `creditRating`. Further, (**MC4**) instances of the superclass (even those that do not represent instances of a lifted subclass) may indiscriminately be given values to the lifted attributes. In the example model, it is possible for a child represented as an instance of `Person` to be assigned a value for `creditCard`, even though this was inadmissible in the original model. The creation of a discriminator attribute in the lifting process provides us the means to formulate missing constraints to enforce the semantics of the original model: assignment of a value to a lifted attribute must be admitted conditionally on the value of the discriminator. In the model shown in Figure 13 (Final arrangement), due to the lifting of `CorporateCustomer` to `Organization`, we need to condition the assignment of `creditRating` and `creditLimit` to those instances of `Organization` for which `isCorporateCustomer=true`. Since we have two applications of lifting of `PersonalCustomer` to `Adult` and then from `Adult` to `Person`, we have the condition that persons may be assigned a `creditRating` and a `creditCard` only when `isPersonalCustomer=true` (i) and that is only admissible when `lifePhase=ADULT` (ii). Note that discriminator attributes may also be subject to lifting in further applications of the operation, and will also result in additional occurrences of MC4 in those cases,that is, the discriminator attribute may depend on the filling of some other discriminator attribute.

Migration of association ends to superclasses. The lifting operation results in the migration of association ends from the lifted subclasses to their superclasses. As a consequence, analogously to mandatory attributes, the association ends opposite to the superclass must have their lower bound cardinality (li_y) relaxed. Again, the discriminator attribute ($isSubType_y$ or gs_a) gives us the means to enforce the semantics of the association is in the source model: (**MC5**), in case $li_y = 1$, we need to introduce a constraint that, for each instance of the superclass, if $isSubType_y =$ `true` (or gs_a is equal to or includes $subType_y$), then that instance of the superclass must be associated to one instance of the related class ($RelatedType_i$). (**MC6**), only those instances with $isSubType_y =$ `true` (or gs_a is equal to or includes $subType_y$) may be associated to an instance of the related class ($RelatedType_i$), to ensure the association end typing is respected. In the model shown in Figure 13 (Final arrangement), due to the lifting of `CorporateCustomer` to `Organization`, we need to condition the association of an organization to a supply contract through `hasCorporateCustomer` to those instances of `Organization` for which `isCorporateCustomer=true`. Likewise, due to the lifting of `Contractor` to `Organization`, we need to condition the association of an organization to a supply contract through `hasContractor` to those instances of `Organization` for which `isContractor=true`. Since we have two applications of lifting of `PersonalCustomer` to `Adult` and then from `Adult` to `Person`, we have the condition that persons must be associated with one `SupplyContract` when `isPersonalCustomer=true` (i) and that that is only admissible when `lifePhase=ADULT` (ii).

6.3 Missing Constraints due to Hierarchy Emulation

The literature on EER to relational schema transformation—such as the seminal work of (TEOREY; YANG; FRY, 1986)—has early on identified missing integrity constraints when translating EER diagrams with generalization and subset hierarchies into relational schemas. They have concentrated their efforts in the *one table per entity* strategy, which does not apply any operations such as flattening and lifting. In any case, the missing constraints they have identified must be taken into account in any transformation that still preserves generalizations in the last translation step (e.g., *one table per class*, *one table per concrete class*).

In these transformations, there is an emulation of generalization in the relational schema. A table corresponding to a subclass has as primary key a foreign key that refers to entries in the table corresponding to the topmost superclass. (In the case of multiple inheritance of classes in the top of the hierarchy, this becomes a composite key.) Let us consider for instance the model in Figure 11 (Final arrangement), which is the result of the flattening of abstract classes, and hence is the class model to be translated into a relational schema in the *one table per concrete class approach*. In the translation of this model, the table `ADULT` corresponding to the concrete class `Adult` has a foreign key `person_id` which is the primary key of `PERSON`, and likewise for `CUSTOMER` and `PERSONAL_CUSTOMER`. A similar solution applies to the other subclasses in the model.

The emulation of generalization provides some guarantees of referential integrity through the use of keys: an entry representing a subclass instance will always be properly related to its superclasses. However, the following missing integrity constraint is identified by Teorey, Yang & Fry (1986, Section 3.1.4, with terminology adjusted here): when a generalization set is disjoint, it must be inadmissible for two tables corresponding to disjoint subclasses to have entries that refer to the same 'superclass key' (**MC7**). This is required to preserve the semantics of disjointness and applies in our example, to the tables `CHILD` and `ADULT` in the *one table per concrete class* approach. Note that this constraint is not required in approaches that remove all generalizations by progressively applying flattening and lifting (e.g., *one table per leaf class*, *one table per hierarchy* and *one table per kind*.)

6.4 Augmenting the Transformation

As we have observed in the previous section, the operations produce cumulative effects throughout the transformation process. We discuss here how this process is operationalized in our implementation by maintaining a set of traces, which are updated when each of the two operations are applied. We also show the last step of the overall conceptual model to relational schema transformation, which entails the production of triggers to enforce the accumulated constraints from the final set of traces and the refactored model.

6.4.1 Transformation Process

Three class-to-table approaches have been implemented in a plugin to Visual Paradigm[1]: *one table per class*, *one table per concrete class*, and *one table per kind*. There are no manual steps in our approach, and the implementation applies the three transformation strategies fully automatically. The three approaches are accommodated by applying flattening and lifting operations in different orders and under different conditions. In the *one table per kind* and in *one table per concrete class*, first, all abstract classes are flattened, from the top of the hierarchy until concrete classes are reached. In the *one table per kind*, leaf concrete classes are lifted until all only kinds remain. In all strategies, the resulting refactored class model is then translated into a relational schema. (In the case of the *one table per class* approach, no operations of flattening and lifting are performed, and hence translation happens in the original model.)

[1] see <https://github.com/nemo-ufes/forward-db-vp-plugin> for the full implementation

6.4.2 Generation of Triggers

We focus here on the generation of triggers to detect violations of the missing constraints that were identified in Section 6.1 and 6.2. The trigger code is generated by using the final state of the trace table and the source model.

Consequences of lifting. The process goes through the trace table to identify the target classes that have filters that were the result of the lifting of mandatory attributes; for example, `isPersonalCustomer` for the target class `Person` and `isCorporateCustomer` for `Organization`. A trigger specification must then detect violations: if (i) the discriminator attribute in the filter matches the filter value and at the same time any of the columns corresponding to mandatory attributes are not filled in (line 7 in Listing 6.1 for the `PERSON` table and the filter on `isPersonCustomer`), addressing the missing constraint **MC3**; or, if (ii) the discriminator attribute in the filter does not match the filter value and any of the columns corresponding to mandatory attributes are filled in (line 10 in Listing 6.1), addressing the missing constraint **MC4**. (While the trigger shown in the listing applies to inserts on `PERSON`, a trigger with the same body is included for any subsequent updates.)

Listing 6.1 – Trigger for the `PERSON` table

```
1   CREATE OR REPLACE FUNCTION check_person_i()
2     RETURNS TRIGGER
3     LANGUAGE PLPGSQL
4     AS
5   $$
6   BEGIN
7       if (
8           (NEW.is_personal_customer = true AND (NEW.credit_card is null OR NEW.
                credit_rating is null))
9           OR
10          (NEW.is_personal_customer <> true AND (NEW.credit_card is not null OR NEW.
                credit_rating is not null))
11      )
12      then
13          RAISE EXCEPTION 'ERROR 1: ...[check_person_i].';
14      end if;
15      if(
16          ( NEW.life_phase_enum <> 'ADULT' AND (
17              (NEW.is_personal_customer is not null  AND  NEW.is_personal_customer = TRUE )
                     OR
18               NEW.credit_rating is not null  OR NEW.credit_card is not null  )
19          )
20      )
21      then
22          RAISE EXCEPTION 'ERROR 2: ...[check_person_i].';
23      end if;
24    RETURN NULL;
25  END
26  $$;
27
28  CREATE TRIGGER tg_person_i
29  AFTER INSERT ON person
30  FOR EACH ROW
31  EXECUTE PROCEDURE check_person_i();
```

The process also identifies, for each target class (here `SupplyContract`, `Person` and `Organization`), whether they were originally associated with other classes in the source model. When the associated source classes have filters in their respective trace sets (here `Contractor` and `Customer` for `SupplyContract`), this means the associated source classes were subject to lifting, and possible violations of the original semantics need to be detected according to the missing constraints identified in Section 6.2 (under 'Migration of association ends to superclasses'). The following violations must be detected in the trigger: (i) if an entry in the table corresponding to the target class (such as `SUPPLY_CONTRACT`) is associated to a lifted source class (such as `Contractor`), but the required filters associated to that source class are not satisfied (lines 7–18 of Listing 6.2 for `SUPPLY_CONTRACT` in its original association with a `Contractor`; lines 20–31 for the original association with a `Customer` concerning its trace to `Organization`, and lines 33–45 for the original association with a `Customer` concerning its trace to `Person`). This addresses **MC6**.

Consequences of flattening. The process also goes through the trace table to identify those trace sets of originally associated classes with more than one trace, which is a consequence of flattening. In our example, this occurs for the association between `SupplyContract` and `Customer`. As discussed in Section 6.1 (under 'Missing constraints'), we must ensure that a supply contract is associated with exactly one instance in the union of `PersonalCustomer` and `CorporateCustomer` in lines 47–52 of Listing 6.2. This addresses the missing constraints **MC1** and **MC2**.

Listing 6.2 – Trigger for the `SUPPLY_CONTRACT` table

```
1  CREATE OR REPLACE FUNCTION check_supply_contract_i()
2    RETURNS TRIGGER
3    LANGUAGE PLPGSQL
4    AS
5  $$
6  BEGIN
7      if NEW.organization_contractor_id is not null
8      then
9          if not exists (
10                 SELECT 1
11                 FROM organization
12                 WHERE is_contractor = TRUE
13                 AND   organization.organization_id = NEW.organization_contractor_id
14               )
15         then
16             RAISE EXCEPTION 'ERROR 1: ...[check_supply_contract_i]';
17         end if;
18     end if;
19
20     if NEW.organization_customer_id is not null
21     then
22         if not exists (
23                 SELECT 1
24                 FROM organization
25                 WHERE is_corporate_customer = TRUE
26                 AND   organization.organization_id = NEW.organization_customer_id
27               )
```

```
        then
            RAISE EXCEPTION 'ERROR 2: ...[check_supply_contract_i]';
        end if;
    end if;

    if NEW.person_id is not null
    then
        if not exists (
                SELECT 1
                FROM person
                WHERE is_personal_customer = TRUE
                AND   life_phase_enum = 'ADULT'
                AND   person.person_id = NEW.person_id
              )
        then
            RAISE EXCEPTION 'ERROR 3: ...[check_supply_contract_i]';
        end if;
    end if;

    if( SELECT  CASE WHEN NEW.organization_customer_id is null THEN 0 ELSE 1 END +
                CASE WHEN NEW.person_id is null is null THEN 0 ELSE 1 END
      )  <> 1
    then
        RAISE EXCEPTION 'ERROR 4: ...[check_supply_contract_i]';
    end if;
END;
$$;

CREATE TRIGGER tg_supply_contract_i
AFTER INSERT ON supply_contract
FOR EACH ROW
EXECUTE PROCEDURE check_supply_contract_i();
```

6.4.3 Implementation Restrictions

Whenever associations have lower bound cardinality 1 in both association ends, there is a mutual dependency between the instances of the associated classes. This is the case in the example of the associations between `SupplyContract` and `Customer` and between `SupplyContract` and `Contractor`. In this case, enforcing both **MC5** and **MC6** after *each* insert becomes problematic: e.g., inserting a row in the SUPPLY_CONTRACT table would require an insertion in the CUSTOMER table and vice-versa. Unfortunately, this would require related inserts to be part of a single transaction, and a database implementation with transaction-level triggers, which is not the case here. Hence, we have opted not to enforce **MC5** and **MC6** in tandem in the implemented trigger generation. Violations of the missing constraint (**MC5** in this case) could be detected by a regular query that can easily be generated from the trace table following the same procedure to obtain the expressions concerning the triggering of **MC6**.

We have not yet implemented support for detecting violations of **MC7**, which applies to the emulation of specialization with keys in disjoint hierarchies. As discussed in Section 6.3, the constraint is not required in *one table per kind*, which has been the focus of our efforts. To support this constraint in general, further instrumentation of the transformation process is required, in particular due to flattening involving disjoint generalization sets. This is not necessary for the particular case of OntoUML models, since the required conditions for the relational schema can be derived directly from the source model by inspecting the classes stereotyped «kind» and «relator» along with generalization set constraints.

Finally, although we have discussed the support for self-relationships in Chapter 3, these are not supported in the implementation.

6.5 Implementation and Tests

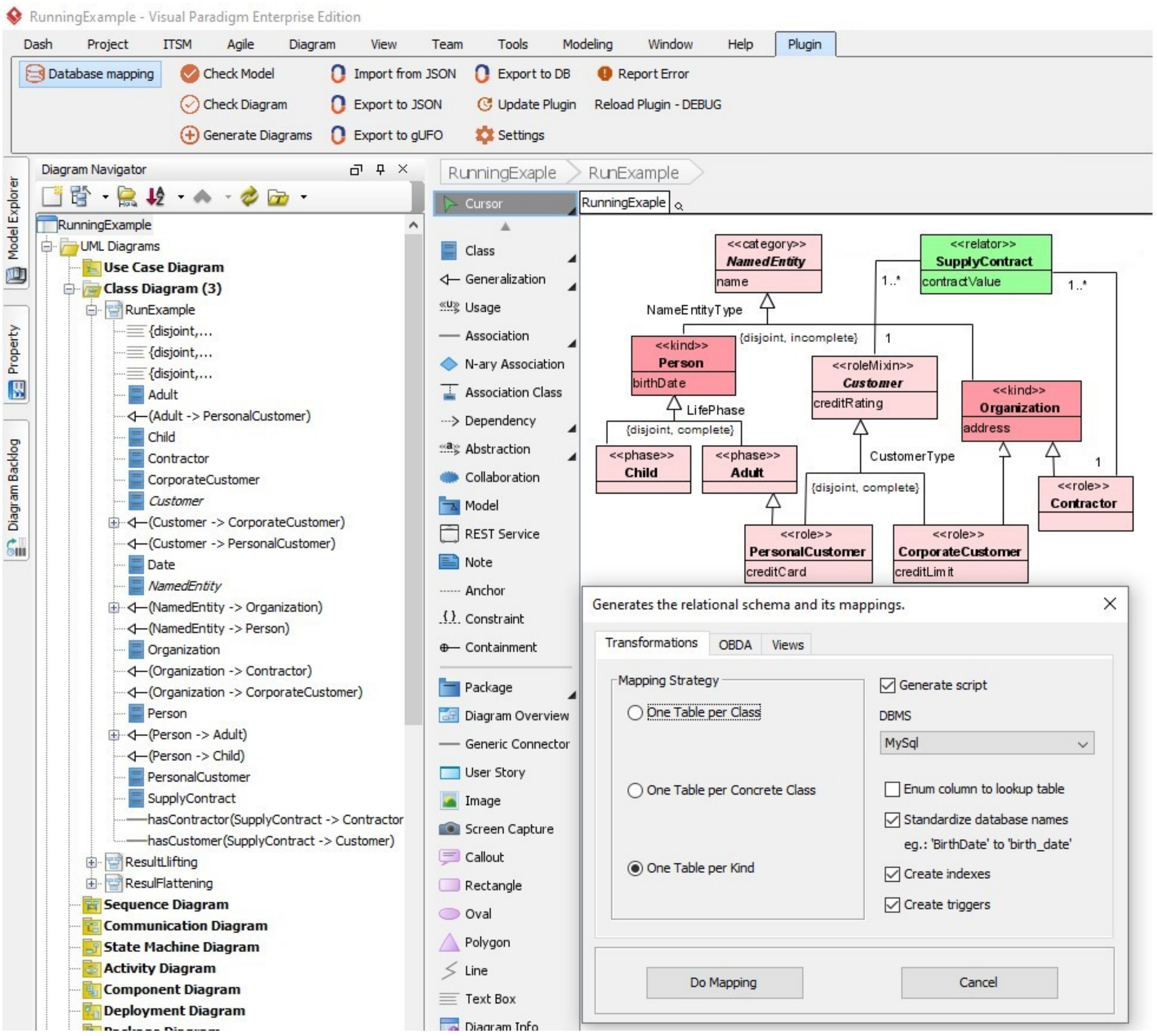
Figure 28 contains a screenshot of the implemented object-relational plugin for Visual Paradigm. The menu at the top includes the "Database mapping" button to access the functionalities of the plugin. (The other buttons displayed are part of the OntoUML plugin[2]) The implementation supports three transformation approaches (*one table per class*, *one table per concrete class*, and *one table per kind*); different target DBMSs for the generated scripts (MySql, Postgres and standard SQL 1999); and some optimization and customization parameters.

In addition to the formal specification and automated proofs for the flattening and lifting operations, we have performed a number of tests on the implemented transformation. As the verification and validation of the constraints produced is manual and requires the understanding of the conceptual model, we employed in the tests the first 30 projects published in the OntoUML repository (BARCELOS et al., 2022), which are produced by third parties. The models evaluated in our tests are in ".vpp" format, forcing us to open the model through the Visual Paradigm to carry out the tests, making it difficult to automatically evaluate a large volume of models. The main objective of these test is to prove that the developed tool works in different modeling situations and not just in controlled scenarios. We used the *one table per kind* approach, as it exercises both *lifting* and *flattening*. Unlike paper (GUIDONI; ALMEIDA; GUIZZARDI,), the data presented in this work was obtained through PostgreSQL 5.2.

Table 18 provides a quantitative overview of the results, reporting the number of generated tables (excluding those used to represent `N:N` associations and multivalued attributes, which are created regardless of the chosen object-relational approach) as well as the number of constraints enforced by the resulting artifacts. Constraints `MC1` and `MC2` are reported together because the same generated command validates both cases (see. e.g., lines 50–56 of Listing 6.2). The same occurs for `MC3` and `MC4`. We excluded from the tests 8 of the first 30 projects due to (i) syntactic errors in the source models, (ii) presence of reserved keywords in labels, or (iii) complex handling of datatypes (addressing complex datatype support and treating reserved keywords is not yet implemented in the prototype).

As expected, the transformation of some projects did not generate flattening-related

[2] OntoUML plugin was developed to facilitate the development of OntoUML models and verifies its compliance with OntoUML's syntactical rules (see (GUIZZARDI et al., 2021b) for more details), among other features. This plugin can be found at <https://github.com/OntoUML/ontouml-vp-plugin>.



Generates the relational schema and its mappings.

Mapping Strategy
- ( ) One Table per Class
- ( ) One Table per Concrete Class
- (●) One Table per Kind

- [✓] Generate script
- DBMS: MySql
- [ ] Enum column to lookup table
- [✓] Standardize database names
  eg.: 'BirthDate' to 'birth_date'
- [✓] Create indexes
- [✓] Create triggers

Figure 28 – A screenshot of the object-relational plugin interface.

MC1 and MC2 constraints because they do not have associations involving abstract classes. These constraints are quite numerous however in projects that include classes defined at a high level of generality (typical of "core ontologies") such as "aguiar2019oco", "aires2022valuenetworks-geo", "amaral2019rot" and "amaral2020rome". Further, some of the projects do not include MC3 and MC4 because they have no attributes in lifted classes or have no concrete class hierarchies of depth greater than two (in which discriminators from a previous lifting round are themselves lifted). MC6 occurrences are the most common as they relate to the typing of associations in lifted classes (and thus are required for every association involving lifted classes independently of the cardinalities). The only projects with no occurrences of MC6 are "barros2020programming" (as it is a pure taxonomy with no associations) and "chartered-service" (that has associations, but does not employ specialization.)

We have selected one project for exhaustive inspection, namely "aguiar2019ooco", which has the largest number of classes in the sample and for which all types of constraints were generated. We do not evaluate the selected project against modeling quality attributes

Table 18 – Transformation process results

Project	**Number of Classes**	**Number of Tables**	**MC1 and MC2 Constraints**	**MC3 and MC4 Constraints**	**MC6 Constraint**
aguiar2019ooco	55	19	12	16	28
ahmad2018aviation	21	5	0	10	16
aires2022valuenetworks-geo	24	19	18	0	6
amaral2019rot	48	26	48	2	18
amaral2020game-theory	22	11	0	2	16
amaral2020rome	48	35	38	0	16
amaral2022ethical-requirements	20	15	2	0	8
andersson2018value-ascription	13	11	0	0	2
bank-model	23	8	2	2	24
barcelos2013normative-acts	42	16	0	10	24
barros2020programming	12	1	0	6	0
brazilian-governmental-organizational-structures	15	5	4	2	4
buchtela2020connection	19	6	0	0	2
buridan-ontology2021	53	20	0	10	82
carolla2014campus-management	17	12	0	0	18
castro2012cloudvulnerability	32	14	0	8	8
cgts2021sebim	29	10	0	2	2
chartered-service	11	11	0	0	0
clergy-ontology	29	13	0	14	58
cmpo2017	55	18	0	12	98
construction-model	13	7	0	4	6
debbech2019gosmo	22	10	0	2	34

such as correctness. We have manually inspected the source model to derive test cases. The test cases cover all of the constraints required to preserve the semantics of the source model in the present of lifting and flattening. 60 test cases consist in attempting to insert or update data that would offend the original model semantics, and hence a pass in these test cases is an insert or update error raised by the generated triggers (the number of test cases does not correspond to the total number of missing constraints because some test cases check inserts only, some other test cases check updates only, and a number of those concern more than one missing constraint.) The test cases jointly cover 100% of the conditions (IFs) in the generated triggers for the model. Another 36 test cases were added to insert or update data that does not offend the original model semantics, to make sure that they would not result in unintended insert or update errors.

An example of test case (`TC_AGUIAR_MC3and4_001`) is shown in Listing 6.3, that exercises the database corresponding to the fragment shown in Figure 29 (from "aguiar2019ooco"). As a result of the lifting process, two discriminator columns are introduced in the table corresponding to the `Language` kind. The `is_object_oriented_programming_language` discriminator column is introduced as a result of the first application of lifting, and the `is_programming_language` is introduced in the second application of lifting. As a consequence, the column `is_object_oriented_programming_language` can only be given a value when `is_programming_language='1'` (i.e., corresponds to `true`). The test case attempts to insert a new language, with identifier 3 (`language_id=3`), marked as an object-oriented programming language (`is_object_oriented_programming_language='1'`), but not marking it as a programming language (`is_programming_language='0'`). This should result in a violation of the insert trigger for the `language` table.

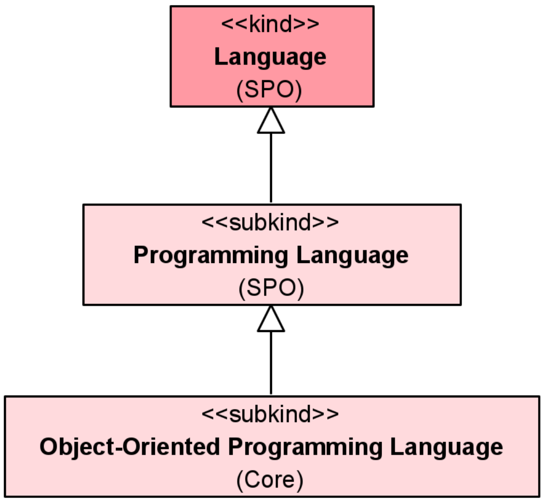


Figure 29 – Fragment of the "aguiar2019ooco" project that results in iterated lifting.

Listing 6.3 – Example test case TC_AGUIAR_MC3and4_001.

```
Test case name: TC_AGUIAR_MC3and4_001
Project name: aguiar2019ooco
Missing constraints evaluated: MC3and4          Number: 1          Action:
    Insert
Objective: Check whether MC3 and MC4 are correctly enforced in the case of
    lifting of "Object-Oriented Programming Language" and "Programming Language
    ".
Test: Insert a row in the table 'language' that is marked as an "Object-Oriented
     Programming Language" but not as "Programming Language".
Expected result: Error.
Script:

INSERT INTO language
       (language_id, is_programming_language,
           is_object_oriented_programming_language)
VALUES (3, false, true);

Return message: ERROR: Violating conceptual model rules[check_language_i].
Test result: PASS
```

We have performed some simple tests to assess the performance overhead imposed with the introduction of the validation triggers. We selected the table from our test set that is accompanied by the largest number of constraints in triggers, namely the `METHOD_MEMBER_FUNCTION` table in "aguiar2019ooco". This table's triggers enforce seven constraints that resulted from lifting and flattening. We have contrasted the performance of 10 individual record inserts in the table with and without the triggers. It is important to mention that there is only one trigger enabled per insert or update operation and there are no indexes on the affected tables. We have found the following results: 10 milliseconds on average for an insert in the table when the triggers are enabled; and 9 milliseconds on average for an insert when the triggers are disabled. This indicates the overhead is not prohibitive. Further performance analysis in actual applications is straightforward, since triggers can be disabled and regular operation (that does not violate constraints) assessed directly.

Table 19 – Transformation performance (in seconds)

Project	**Schema generation time (seconds)**	**Triggers generation time (seconds)**
aguiar2019ooco	0.019	0.026
buridan-ontology2021	0.056	0.063
cmpo2017	0.168	0.237

Finally, in order to assess the performance of the transformation itself (design time performance), we have measured the time required to generate all relational schemas and triggers in the largest models in our test set. The results are presented in Table 19, showing the response times in seconds for the generation of the schema and of the scripts for the three largest projects in the test set (the data were obtained in the same computational environment presented in Section 4.3).

6.6 Related Work

Banerjee et al. (1987) propose well-defined rules that cover many aspects of object-oriented database schema manipulations, including "dropping an existing class". Differently from our work, their overall objective is to perform evolutionary manipulations of object-oriented schemas, so there is no concern for preservation of all aspects of the original model. The same can be said of several other works, like those of Penney & Stein (1987), Lerner & Habermann (1990). They also rely on the solution space of object-oriented database systems.

Some work on refactoring strategies on UML class diagrams aim to preserve the syntactic correctness and/or semantics of the original model. For example, Markovic & Baar (2005) detail some refactoring rules along with their impacts on OCL constraints. Baar & Markovic (2006) focus on the preservation of semantics in the face of a 'MoveAttribute' operation. As discussed in Section 6.1 and 6.2 concerning the refactoring presented by Fowler (2018), the intent here is the transformation between models, and so the supported operations do not match those we require for relational schema realization.

There are also different approaches that aim to obtain SQL implementations of explicitly formulated OCL constraints. Some of these approaches, e.g., OCL2SQL (DEMUTH; HUSSMANN, 1999; DEMUTH; HUSSMANN; LOECHER, 2001), Incremental OCL constraints checking (ORIOL; TENIENTE, 2014) and OCL_{FO} (FRANCONI et al., 2014) are focused on transformations OCL Boolean expressions only, while others such as SQL-PL4OCL (EGEA; DANIA, 2017) and MySQL4OCL (EGEA; DANIA; CLAVEL, 2010) consider OCL expressions in general. One other approach, namely, GeoUML (PELAGATTI et al., 2009) considers the SQL implementation of special-purpose OCL constraints for expressing geo-spatial relations.

There is a variety of studies focusing on the transformation of structural conceptual models, such as EER diagrams or UML class diagrams, into relational schemas, such as those performed by Shah & Slaughter (2003), Hull & King (1987), Teorey, Yang & Fry (1986), Pergl, Sales & Rybola (2013). Shah & Slaughter (2003) discuss the various class-to-table strategies but do not provide detailed model transformation rules for realizing these strategies, and do not consider the consequences of the strategies in terms of preservation of semantics. The seminal study presented by Teorey, Yang & Fry (1986) proposes transformation rules to bridge the constructs of EER diagrams with those of relational schemas, identifying missing constraints for disjoint generalizations as we discussed in Section 6.3. More recently, Pergl, Sales & Rybola (2013) have discussed how to transform OntoUML models into relational schemas. In Rybola (2017) Ph.D. thesis, proposes the transformation process, presenting quite sophisticated ways of preserving integrity constraints. Validation triggers were proposed to preserve the semantics of the original constructs from the source OntoUML model (*modal aspects* of the stereotypes which we do not address here). A common trait of these approaches is that they consider solely the *one table per class* strategy for transforming generalization hierarchies. While this yields a straightforward relation between the conceptual model and the relational schema, using the resulting relational schema may be cumbersome depending on the source conceptual model, in particular in the presence of generalization hierarchies, which are emulated as we discussed in Section 6.3, with the consequences discussed in Chapters 2 and 4. Because of this, these tools are usually employed by providing a UML class diagram that is in fact a visual representation of the relational schema: the model is produced manually at a lower level of abstraction, with automation restricted to straightforward translation. In some cases, stereotypes for primary and foreign keys are introduced in the notation (SHAH; SLAUGHTER, 2003).

Alloy is commonly used in the literature to validate semantic changes in UML class models. For example, the work of Cunha, Garis & Riesco (2015) proposes to identify semantic losses through bidirectional transformations using Alloy. Their job consists of performing transformations for Alloy through the UML and OCL specifications, and to translate them back to UML and OCL, thus allowing to verify and validate the results of a transformation in Alloy. Gheyi, Massoni & Borba (2007) present a catalog of primitive transformations to predict whether a change in the source model will cause semantic loss. To do so, the authors formalize a static semantics for Alloy to match the source model semantics. With this, it is possible to predict whether any change in the source model will cause semantic losses. These approaches provide useful general frameworks which could be employed to formalize the operations we have discussed here.

Finally, there are a number of model transformation specification languages in the literature, including most prominently ATL (JOUAULT; KURTEV, 2005), the Epsilon Transformation Language (ETL) (KOLOVOS; PAIGE; POLACK, 2008) and those solutions implementing the MOF QVT specification (KURTEV, 2007). These languages focus on the *specification* of a model transformation in terms of the constructs present in source and target metamodels. Model transformation specifications written in these languages are usually interpreted with a corresponding transformation engine to execute the transformation of a specific source model. Model transformation languages operate at the abstract syntax level and are neutral with respect to the semantics of the source and target languages. We focus here instead on the *content* of the transformation, i.e., we are concerned with the design decisions that can be generalized into a specific model transformation and with the semantics of the source model and its correspondence with the resulting relational schema. In principle, any of these transformation languages could have been used for the implementation of the transformation we present here.

6.7 Final Considerations

In this chapter, we have focused on the consequences of various class-to-table transformation strategies, such that the relational schema can be complemented with integrity constraints that reflect the source conceptual model constraints. We have formulated the approach in terms of the consequences of flattening and lifting thus account for various transformation strategies in the literature, including *one table per concrete class*, *one table per leaf class*, *one table per kind*, and *one table per hierarchy*.

We has focused only on the challenges concerning the transformation of inheritance hierarchies into relational schemas; other aspects of object-relational mapping such as whole-part associations, N-to-N associations, or the creation and propagation of primary keys were not part of our studies. However, we used the extensive knowledge available in the literature on object-relational mapping to make our transformation tool functional.

An important limitation of the approach is that we do not deal with existing (OCL) constraints that need to be rewritten in the transformation process due to lifting and flattening. For example, consider the constraint that a contractor should not be a customer of itself in the scope of a supply contract. We expect we can profit from query rewriting strategies in the literature—such as those used in Ontology-Based Data Acccess (CALVANESE et al., 2017)—to address this in the future. We also aim to investigate to what extent we can leverage the aforementioned views such that OCL invariants can be directly enforced at the database level with references to those views instead of rewriting. We also do not address behavioral and dynamic aspects, and provide no special treatment for whole-part relations.

7 Conclusions and Future Work

In this work, we have explored principles of Ontology-Driven Conceptual Modeling to propose a new object-relational transformation strategy. By studying the literature on object-relational transformations, we have identified the flattening and lifting operations performed on inheritance hierarchies. We have then explored sortality and rigidity of types to guide flattening and lifting selectively according to these metaproperties, producing the *one table per kind* strategy. Hence, we have demonstrated that ontological metaproperties can be used to automate relational schema design. Applying these operations by following different criteria, other strategies may arise, such as *one table per sortal*, *one table per rigid sortal*, etc.

We demonstrated that our strategy can cope effectively with certain aspects of conceptual modeling that have been neglected by some other transformation strategies (multiple inheritance, orthogonal generalizations, dynamic classification). We have also shown that, in terms of performance, the proposed approach presents similar or better results in some situations in relation to other strategies. We have also shown empirical support for the claim that the proposed approach does not jeopardize the correctness of relational schema understanding (as judged by query correctness interpretation). The *one table per kind* strategy resulted in a schema that was preferred by a significant majority of the participants in an empirical experiment when contrasted with *one table per concrete class*.

A common characteristic of many approaches in the literature (SHAH; SLAUGHTER, 2003; HULL; KING, 1987; TEOREY; YANG; FRY, 1986; PERGL; SALES; RYBOLA, 2013) is that they consider solely the *one table per class* strategy for transforming generalization hierarchies, which is also the approach adopted in the various commercial tools to perform this kind of transformation. A result is that the conceptual model is in fact produced manually by the modeler at a lower level of abstraction, with automation restricted to straightforward translation. That is detrimental to the objectives of model-driven development, including abstraction and platform-independence. In this light, our work contributes to a fuller application of model-driven transformation of conceptual models, by effectively decoupling conceptual models from their implementation.

We evolved our transformation process to incorporate a tracing structure of the conceptual model classes into the relational schema tables, adhering to any object-relational mapping strategy that uses flattening and lifting operations to transform the inheritance hierarchy. From this evolution, we were able to generate the corresponding mappings for the Ontop OBDA platform independently of the selection of a transformation strategy. We have demonstrated the effectiveness of our solution through the Ontop plugin for Protégé, where the same SPARQL query can be executed on different relational schemas generated by our solution. We also present a simple study of the response time of queries generated by Ontop with queries produced manually in different relational schemas. Encapsulating transformation decisions and OBDA mapping generation alleviate designers from an otherwise manual and error-prone design process.

Finally, we identify and conceptualize some constraints lost by the transformation process in light of flattening and lifting operations. As a result, we were able to generate database triggers to restrict inserts and updates that do not respect the conceptual model constraints. The amount of validations generated for different conceptual models, presented in Section 6.4.3, corroborates the need to generate restrictions to ensure that the conceptual model constraints are respected in the relational database realization.

7.1 Research Contributions

We can summarize the research contributions in this thesis as follows:

- **We have proposed the *one table per kind* strategy of object-relational transformation.** After contextualizing the concepts that govern how hierarchies can be arranged according to OntoUML 2.0, we proposed the *one table per kind* strategy of object-relational transformation based on UFO principles (addressing the specific objective 1). The strategy presented is based on *flattening* and *lifting* operations, which can be applied to other class-based models, as long as it is possible to identify the *kinds* in these models. The *flattening* and *lifting* operations can also be customized to generate other strategies, whether ontology-driven—such as *one table per sortal*—or not. This work demonstrates that it is possible to propose a new approach for transforming inheritance hierarchies by taking into account the ontological categories of classes, providing evidence to support our first hypothesis.

- **We automate the relational schema generation process.** As a result of this work, we developed a Visual Paradigm plugin capable of generating the relational schema script from the class model for a number of Database Management Systems (PostgreSQL, MySQL, SqlServer, Oracle, and H2) (addressing specific objective 2). The supported transformation strategies (initially, *one table per class*, *one table per concrete class*, and *one table per kind*) were implemented through successive applications of *flattening* and *lifting* operations.

- **We evaluate the understanding of the relational schemas generated by the *one table per kind* strategies.** We performed an empirical study to assess the understanding of the relational schema produced by our strategy (addressing the specific objective 3). The majority of the participants indicated understanding was easier with the relational schema generated by the *one table per kind* strategy, providing evidence to support our second hypothesis.

- **We have proposed a forward engineering approach to enable access to data in the relational schema through the conceptual model.** We encapsulate all the manual (error-prone) work of relational schema generation and OBDA mapping production (addressing specific objective 4), confirming our third hypothesis. To make this possible, we reused a plugin to generate the OWL file from the OntoUML model and another plugin to transform SPARQL queries into SQL queries (a gUFO plugin for Visual Paradigm and the Ontop plugin for Protégé, respectively).

- **We provide means to preserve model constraints in the relational realization.** We identify the constraints of the conceptual model that are affected as a consequence of the *flattening* and *lifting* operations. With this, we were able to identify, store and reconstruct the constraints in terms of triggers that prevent the invalid insertions or updates (addressing specific objective 5).

The aforementioned research contributions were published in three peer-reviewed papers:

- GUIDONI, Gustavo Ludovico; ALMEIDA, João Paulo A.; GUIZZARDI, Giancarlo. **Transformation of ontology-based conceptual models into relational schemas.** In: Conceptual Modeling - 39th International Conference, ER 2020. Lecture Notes in Computer Science, vol. 12400, Springer, 2020. DOI: 10.1007/978-3-030-62522-1_23.

- GUIDONI, Gustavo Ludovico; ALMEIDA, João Paulo A.; GUIZZARDI, Giancarlo. **Forward Engineering Relational Schemas and High-Level Data Access.** In: Conceptual Modeling - 40th International Conference, ER 2021. Lecture Notes in Computer Science, vol. 13011, Springer, 2021. DOI: 10.1007/978-3-030-89022-3_12.

- GUIDONI, Gustavo Ludovico; ALMEIDA, João Paulo A.; GUIZZARDI, Giancarlo. **Preserving conceptual model semantics in the forward engineering of relational schemas**, Frontiers in Computer Science, vol. 4, Frontiers Media, 2022. DOI: 10.3389/fcomp.2022.1020168.

7.2 Limitations

The performance analysis was limited to two models, one designed with the purpose of exploring all the modeling primitives (the running example), and another model produced independently by a third-party. Restricting the evaluation to these two models, we were only able to show the feasibility of the proposed strategy in these scenarios. To indicate which transformation strategy is better suited for a set of relational schema usage requirements, it is necessary to carry out deeper analyses, which would involve various of the strategies presented in this work. We argue that this can be done in a much easier manner when the various transformations are automated as proposed here. Applying several automated transformations can enable one later to measure and compare response times, perform further analyses of the space used, the effect of concurrency on the various generated schemas, etc. Similarly, one can consider the effect of techniques employed by different Database Management Systems (DBMS) such as query optimization strategies, suppression of "null" values, etc.

The empirical evaluation was carried out with a relatively small number of experienced database users for a single model. We believe it is necessary to evaluate the approach with a larger group of participants, categorizing them by experience in database projects to identify the consequences of using the relational schemas produced by the strategies presented in this work. A variety of realistic models could be employed in this process.

The verification of the correctness of the relational schema generated by the proposed tool was performed manually for the test models presented in Section 6. Although these testing steps were undertaken carefully, it is relevant that this evaluation be carried out in an automated way or by people not involved in the design and implementation of the transformation tool.

7.3 Future Work

We identify the following points for further investigation and improvement:

- The transformation approach can be extended to facilitate access to the data through database views that mirror the conceptual model classes. In the same way that OBDA mappings are generated for Ontop, we anticipate that it is also possible to generate database views of each conceptual model class, regardless of the chosen transformation strategy. In these cases, in addition to providing database abstraction, it would also be possible to work with applications that are written directly in terms of the conceptual model classes. Special attention will be required to data change operations.

- The transformation approach can be extended to propagate changes in the conceptual model to the relational schema. This will require the definition of a new transformation process to identify changes in the conceptual model and to synthesize scripts for automated database schema migration.

- The transformation approach can be extended to enable bidirectional tracing. The tracing table presented in Chapter 5 provides tracing from the classes of the conceptual model to the tables of the relational schema. Identifying the classes that correspond to the resulting tables is not supported easily with the trace tables that were defined, because the resulting tables can be found several times on the target side of the `TraceTable`.

Having bidirectional traces may support a strategy to propagate changes in the relational schema back to the corresponding classes in the conceptual model.

- The approach can be extended to support the automatic evaluation of database performance for a specific source conceptual model. This extension would be possible by generating different relational schemas through different transformation strategies, populating the resulting database and evaluating the performance of each relational schema. This solution could also be applicable to evaluate the most suitable DBMS for a given source model and strategy. Unlike benchmarks, which characterize the general performance of a database, this solution would provide indication of performance for the specific application at hand.

- The transformation approach could be extended to support the rewriting of OCL constraints in the source model, which are affected due to flattening and lifting operations. OCL constraints can refer to classes or features that are altered in the transformation processes, and that should be taken into account.

- Further investigation could focus on the usability of resulting queries. The design space for SQL queries is enormous, and ease of use varies significantly according to the developer's experience. Therefore, a comprehensive assessment of the various aspects that affect query and schema design and consequent ease of use could be useful to influence the various transformation strategies.

- Ontology-based transformation strategies similar to the one employed for here relational databases could be investigated concerning their suitability to guide the realization of non-relational databases, such as "NoSQL" databases (STONEBRAKER, 2010), which are gaining popularity and have not had the same attention as relational databases when considered as targets for conceptual model transformation.

Bibliography

AFZAL, H. et al. OWLMap: fully automatic mapping of ontology into relational database schema. *Intl. journal of advanced computer science and applications*, v. 7, n. 11, p. 7–15, 2016. Cited in page 70.

AGUIAR, C. Z. de. *Interoperabilidade Semântica entre Códigos-Fonte baseada em Ontologia.* Vitória, ES, Brasil, 2021. Cited 2 times in pages 57 and 61.

ALBANO, A. et al. An object data model with roles. In: *Proc. 19th International Conf. on Very Large Data Bases.* Morgan Kaufmann, 1993. p. 39–51. Disponível em: <http://www.vldb.org/conf/1993/P039.PDF>. Cited in page 26.

ALMEIDA, J.; IACOB, M.; ECK, P. v. Requirements Traceability in Model-Driven Development: Applying Model and Transformation Conformance. *Information Systems Frontiers*, v. 9, p. 327–342, 2007. ISSN 13873326. Cited in page 19.

AMBLER, S. W. Mapping objects to relational databases. White Paper, AmbySoft Inc. 1997. Cited 3 times in pages 20, 25, and 71.

AMBLER, S. W. *Agile database techniques: effective strategies for the agile software developer.* USA: Wiley, 2003. Cited 4 times in pages 18, 20, 35, and 42.

BAAR, T.; MARKOVIC, S. A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In: *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI.* Novosibirsk, Russia: Springer, 2006. (Lecture Notes in Computer Science, v. 4378), p. 70–83. Disponível em: <https://doi.org/10.1007/978-3-540-70881-0_9>. Cited in page 96.

BAGOSI, T. et al. The Ontop framework for ontology based data access. In: ZHAO, D. et al. (Ed.). *The Semantic Web and Web Science - 8th Chinese Conference, CSWS.* Wuhan, China: Springer, 2014. (Communications in Computer and Information Science, v. 480), p. 67–77. Cited in page 20.

BANERJEE, J. et al. Semantics and implementation of schema evolution in object-oriented databases. In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference.* San Francisco, CA, USA: ACM Press, 1987. p. 311–322. Disponível em: <https://doi.org/10.1145/38713.38748>. Cited in page 96.

BARCELOS, P. P. F. et al. A FAIR Model Catalog for Ontology-Driven Conceptual Modeling Research. In: *41st International Conference Proceedings (ER).* Hyderabad, India: Springer-Verlag, 2022. p. 17–20. <https://purl.org/ontouml-models/>. Cited 3 times in pages 57, 61, and 92.

BAUER, C.; KING, G. *Hibernate in action.* Greenwich, CT: Manning, 2005. v. 1. Cited in page 71.

BENEVIDES, A. B. et al. Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. *J. Univers. Comput. Sci.*, v. 16, n. 20, p. 2904–2933, 2011. Cited in page 39.

BIZER, C.; SEABORNE, A. D2RQ-Treating non-RDF databases as virtual RDF graphs. In: *Proceedings of the 3rd international semantic web conference, ISWC*. Springer, 2004. v. 2004. Disponível em: <https://files.ifi.uzh.ch/ddis/iswc_archive/iswc/ab/2004/iswc2004.semanticweb.org/posters/PID-SMCVRKBT-1089637165.pdf>. Cited in page 83.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. 1st. ed. Boston, MA, USA: Addison-Wesley, 1999. ISBN 978-0131489066. Cited in page 17.

BRAGA, B. F. B. et al. Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. *Innov. Syst. Softw. Eng.*, v. 6, n. 1-2, p. 55–63, 2010. Cited in page 39.

BRAMBILLA, M.; CABOT, J.; WIMMER, M. *Model-Driven Software Engineering in Practice*. San Rafael, CA, USA: Morgan & Claypool, 2012. v. 1. (Synthesis Lectures on Software Engineering, v. 1). ISBN 978-1-60845-882-0. Cited in page 17.

CABIBBO, L. Objects meet relations: On the transparent management of persistent objects. In: *Proc. CAiSE*. Springer, 2004. (Lecture Notes in Computer Science, v. 3084), p. 429–445. Disponível em: <https://doi.org/10.1007/978-3-540-25975-6_31>. Cited in page 21.

CALVANESE, D. et al. Conceptual schema transformation in ontology-based data access. In: *European Knowledge Acquisition Workshop*. Cham: Springer, 2018. (Lecture Notes in Computer Science, v. 11313), p. 50–67. Cited in page 83.

CALVANESE, D. et al. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, v. 8, n. 3, p. 471–487, 2017. Cited 4 times in pages 24, 76, 79, and 98.

CARDELLI, L. A semantics of multiple inheritance. In: KAHN, G.; MACQUEEN, D. B.; PLOTKIN, G. D. (Ed.). *Semantics of Data Types*. Sophia-Antipolis, France: Springer, 1984. (Lecture Notes in Computer Science, v. 173), p. 51–67. ISBN 3-540-13346-1. Disponível em: <https://doi.org/10.1007/3-540-13346-1_2>. Cited in page 26.

CARRÉ, B.; GEIB, J. The point of view notion for multiple inheritance. In: YONEZAWA, A. (Ed.). *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming, OOPSLA/ECOOP, Proceedings*. Ottawa, Canada: ACM, 1990. p. 312–321. Disponível em: <https://doi.org/10.1145/97945.97983>. Cited in page 26.

CARVALHO, V. A. *Foundations for Ontology-based Multi-level Conceptual Modeling*. Tese (Doutorado) — Universidade Federal do Espírito Santo, 2016. Cited in page 18.

CARVALHO, V. A.; ALMEIDA, J. P. A.; GUIZZARDI, G. Using reference domain ontologies to define the real-world semantics of domain-specific languages. In: JARKE, M. et al. (Ed.). *Advanced Information Systems Engineering - 26th International Conference, CAiSE. Proceedings*. Thessaloniki, Greece: Springer, 2014. (Lecture Notes in Computer Science, v. 8484), p. 488–502. Cited in page 39.

CARVALHO, V. A.; ALMEIDA, J. P. A.; GUIZZARDI, G. Using a well-founded multi-level theory to support the analysis and representation of the powertype pattern in conceptual modeling. In: *International Conference on Advanced Information Systems Engineering*. Ljubljana, Slovenia,: Springer, 2016. p. 309–324. Cited in page 45.

CHEN, P. P. The entity-relationship model - toward a unified view of data. *ACM Trans.*

Database Syst., v. 1, n. 1, p. 9–36, 1976. Disponível em: <https://doi.org/10.1145/320434.320440>. Cited 2 times in pages 17 and 18.

CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 13, n. 6, p. 377–387, jun. 1970. Disponível em: <https://doi.org/10.1145/362384.362685>. Cited in page 18.

CORRADINI, A. et al. Handbook of graph grammars and computing by graph transformation: Volume i. foundations. World Scientific Publishing Co., p. 163–245, 1997. Cited in page 17.

CUNHA, A.; GARIS, A. G.; RIESCO, D. Translating between Alloy specifications and UML class diagrams annotated with OCL. *Softw. Syst. Model.*, v. 14, n. 1, p. 5–25, 2015. Disponível em: <https://doi.org/10.1007/s10270-013-0353-5>. Cited in page 97.

CZARNECKI, K.; EISENECKER, U. W. *Generative Programming. Methods, Tools and Applications.* USA: ACM Press/Addison-Wesley Publishing Co., 2000. Cited in page 17.

DEMUTH, B.; HUSSMANN, H. Using UML/OCL constraints for relational database design. In: *Proc. UML'99.* Berlin, Heidelberg: Springer, 1999. (Lecture Notes in Computer Science, v. 1723), p. 598–613. Cited in page 96.

DEMUTH, B.; HUSSMANN, H.; LOECHER, S. OCL as a specification language for business rules in database applications. In: GOGOLLA, M.; KOBRYN, C. (Ed.). *«UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Proceedings.* Toronto, Canada: Springer, 2001. (Lecture Notes in Computer Science, v. 2185), p. 104–117. Disponível em: <https://doi.org/10.1007/3-540-45441-1_9>. Cited in page 96.

DING, L. et al. Using ontologies in the semantic web: A survey. In: SHARMAN, R.; KISHORE, R.; RAMESH, R. (Ed.). *Ontologies: A Handbook of Principles, Concepts and Applications in Information Systems.* Springer, 2007, (Integrated Series in Information Systems, v. 14). p. 79–113. Disponível em: <https://doi.org/10.1007/978-0-387-37022-4_4>. Cited in page 20.

EGEA, M.; DANIA, C. SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language. *Software & Systems Modeling*, Springer, v. 18, n. 1, p. 769–791, maio 2017. Disponível em: <https://doi.org/10.1007/s10270-017-0597-6>. Cited in page 97.

EGEA, M.; DANIA, C.; CLAVEL, M. MySQL4OCL: A stored procedure-based mysql code generator for OCL. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, v. 36, 2010. Cited in page 97.

ERLING, O.; MIKHAILOV, I. RDF support in the virtuoso DBMS. In: AUER, S. et al. (Ed.). *The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW).* Leipzig, Germany: GI, 2007. (LNI, P-113). Cited in page 83.

FOWLER, M. *Patterns of enterprise application architecture.* Boston, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321127420. Cited 7 times in pages 18, 28, 30, 33, 42, 52, and 71.

FOWLER, M. *Refactoring: Improving the Design of Existing Code.* USA: Pearson Education, 2018. (Addison-Wesley Signature Series (Fowler)). ISBN 9780134757704. Cited 2 times in pages 52 and 96.

FRANCONI, E. et al. Logic foundations of the OCL modelling language. In: *Logics in Artificial*

Intelligence - 14th European Conference, JELIA. Proceedings. Funchal, Madeira, Portugal: Springer, 2014. (Lecture Notes in Computer Science, v. 8761), p. 657–664. Cited in page 97.

GHEYI, R.; MASSONI, T.; BORBA, P. A static semantics for Alloy and its impact in refactorings. *Electron. Notes Theor. Comput. Sci.*, v. 184, p. 209–233, 2007. Disponível em: <https://doi.org/10.1016/j.entcs.2007.03.023>. Cited in page 97.

GONÇALVES, B.; GUIZZARDI, G.; FILHO, J. G. P. An electrocardiogram (ECG) domain ontology. In: *Workshop on Ontologies and Metamodels for Software and Data Engineering, 2nd.* João Pessoa, Brazil: [s.n.], 2007. p. 68–81. Cited in page 18.

GOTTLOB, G.; SCHREFL, M.; RÖCK, B. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, ACM, New York, NY, USA, v. 14, n. 3, p. 268–296, 1996. ISSN 1046-8188. Cited in page 26.

GRANDY, R. E.; FREUND, M. A. Sortals. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy.* Summer 2021. Metaphysics Research Lab, Stanford University, 2021. Disponível em: <https://plato.stanford.edu/archives/sum2021/entries/sortals/>. Cited in page 40.

GRECA, I. M.; MOREIRA, M. A. Mental models, conceptual models, and modelling. *International journal of science education*, Taylor & Francis, v. 22, n. 1, p. 1–11, 2000. Cited in page 17.

GUARINO, N. The ontological level: Revisiting 30 years of knowledge representation. In: BORGIDA, A. et al. (Ed.). *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos.* Springer, 2009. (Lecture Notes in Computer Science, v. 5600), p. 52–67. Disponível em: <https://doi.org/10.1007/978-3-642-02463-4_4>. Cited in page 38.

GUARINO, N.; GUIZZARDI, G. "We Need to Discuss the Relationship": Revisiting Relationships as Modeling Constructs. In: *Advanced Information Systems Engineering - 27th International Conference, CAiSE, Proceedings.* Stockholm, Sweden: Springer, 2015. (Lecture Notes in Computer Science, v. 9097), p. 279–294. Disponível em: <https://doi.org/10.1007/978-3-319-19069-3_18>. Cited in page 40.

GUARINO, N.; WELTY, C. A. An overview of OntoClean. In: STAAB, S.; STUDER, R. (Ed.). *Handbook on ontologies.* Heidelberg: Springer, 2004. p. 151–171. Cited in page 20.

GUIDONI, G. L.; ALMEIDA, J. P. A.; GUIZZARDI, G. Preserving conceptual model semantics in the forward engineering of relational schemas. *Frontiers in Computer Science*, Frontiers, p. 155. Cited in page 92.

GUIDONI, G. L.; ALMEIDA, J. P. A.; GUIZZARDI, G. Transformation of ontology-based conceptual models into relational schemas. In: *Conceptual Modeling - 39th International Conference, ER, Proceedings.* Vienna, Austria: Springer, 2020. (Lecture Notes in Computer Science, v. 12400), p. 315–330. Cited 3 times in pages 39, 48, and 57.

GUIZZARDI, G. *Ontological foundations for structural conceptual models.* Tese (Doutorado) — University of Twente, 10 2005. Cited 11 times in pages 18, 20, 21, 22, 25, 38, 39, 40, 41, 54, and 72.

GUIZZARDI, G. On ontology, ontologies, conceptualizations, modeling languages, and (meta)models. In: VASILECAS, O.; EDER, J.; CAPLINSKAS, A. (Ed.). *Databases and Information Systems IV - Selected Papers from the Seventh International Baltic Conference, DB&IS.* Vilnius, Lithuania: IOS Press, 2007. (Frontiers in Artificial Intelligence and Applications, v. 155), p. 18–39. Cited in page 38.

GUIZZARDI, G. et al. Towards an ontological analysis of powertypes. In: PAPINI, O. et al. (Ed.). *Proceedings of the Joint Ontology Workshops 2015 Episode 1: The Argentine Winter of Ontology co-located with the 24th International Joint Conference on Artificial Intelligence (IJCAI)*. Buenos Aires, Argentina: CEUR-WS.org, 2015. (CEUR Workshop Proceedings, v. 1517). Cited in page 20.

GUIZZARDI, G. et al. Ontology-based model abstraction. In: *13th International Conference on Research Challenges in Information Science, RCIS*. Brussels, Belgium: IEEE Press, 2019. p. 1–13. Cited 3 times in pages 43, 56, and 71.

GUIZZARDI, G. et al. Types and taxonomic structures in conceptual modeling: A novel ontological theory and engineering support. *Data & Knowledge Engineering*, Elsevier, v. 134, p. 101891, 2021. Cited 2 times in pages 22 and 39.

GUIZZARDI, G. et al. Types and taxonomic structures in conceptual modeling: A novel ontological theory and engineering support. *Data Knowl. Eng.*, v. 134, p. 101891, 2021. Disponível em: <https://doi.org/10.1016/j.datak.2021.101891>. Cited in page 92.

GUIZZARDI, G. et al. Endurant types in ontology-driven conceptual modeling: Towards OntoUML 2.0. In: *Proc. ER*. Xi'an, China: Springer, 2018. (Lecture Notes in Computer Science, v. 11157), p. 136–150. Cited 3 times in pages 21, 38, and 41.

GUIZZARDI, G. et al. Towards ontological foundations for conceptual modeling: The unified foundational ontology (UFO) story. *Appl. Ontology*, v. 10, n. 3-4, p. 259–271, 2015. Disponível em: <https://doi.org/10.3233/AO-150157>. Cited 4 times in pages 18, 20, 22, and 38.

GUIZZARDI, G. et al. An ontologically well-founded profile for UML conceptual models. In: PERSSON, A.; STIRNA, J. (Ed.). *Advanced Information Systems Engineering, 16th International Conference, CAiSE, Proceedings*. Riga, Latvia: Springer, 2004. (Lecture Notes in Computer Science, v. 3084), p. 112–126. Disponível em: <https://doi.org/10.1007/978-3-540-25975-6_10>. Cited in page 18.

HULL, R.; KING, R. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, ACM, v. 19, n. 3, p. 201–260, 1987. Disponível em: <https://doi.org/10.1145/45072.45073>. Cited 2 times in pages 97 and 99.

IACOB, M.; STEEN, M. W.; HEERINK, L. Reusable model transformation patterns. In: *12th Enterprise Distributed Object Computing Conference Workshops*. Munich, Germany: IEEE, 2008. p. 1–10. ISBN 978-0-7695-3720-7. Cited in page 52.

IRELAND, C.; BOWERS, D. Exposing the myth: object-relational impedance mismatch is a wicked problem. In: *Proc. 7th DBKDA*. IARIA XPS Press, 2015. p. 21–26. Disponível em: <https://oro.open.ac.uk/43318/>. Cited 2 times in pages 17 and 25.

IRELAND, C. et al. A classification of object-relational impedance mismatch. In: CHEN, Q. et al. (Ed.). *The First International Conference on Advances in Databases, Knowledge, and Data Applications*. Gosier, Guadeloupe, France: IEEE Computer Society, 2009. p. 36–43. Disponível em: <https://doi.org/10.1109/DBKDA.2009.11>. Cited in page 22.

IRELAND, C. et al. Understanding object-relational mapping: A framework based approach. *Int J Adv Softw*, v. 2, 2009. Cited in page 25.

JIMÉNEZ-RUIZ, E. et al. BootOX: Practical mapping of RDBs to OWL 2. In: *Proc. 14th Int.*

Semantic Web Conf. ISWC - Part II. Bethlehem, USA: Springer, 2015. v. 9367, p. 113–132. Cited in page 83.

JOUAULT, F.; KURTEV, I. Transforming models with ATL. In: *International Conference on Model Driven Engineering Languages and Systems*. Montego Bay, Jamaica: Springer, 2005. p. 128–138. Cited in page 98.

KELLER, W. Mapping objects to tables. In: *Proc. of European Conference on Pattern Languages of Programming and Computing*. Kloster Irsee, Germany: Citeseer, 1997. v. 206, p. 207. Cited 9 times in pages 18, 19, 20, 28, 30, 33, 42, 52, and 71.

KENT, W.; HOBERMAN, S. *Data and Reality: A Timeless Perspective on Perceiving and Managing Information in Our Imprecise World*. Technics Publications, LLC, 2012. (Classic Series). ISBN 9781935504214. Disponível em: <https://books.google.com.br/books?id=7z57tgAACAAJ>. Cited in page 18.

KOLOVOS, D. S.; PAIGE, R. F.; POLACK, F. A. C. The epsilon transformation language. In: VALLECILLO, A.; GRAY, J.; PIERANTONIO, A. (Ed.). *Theory and Practice of Model Transformations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 46–60. ISBN 978-3-540-69927-9. Cited in page 98.

KURTEV, I. State of the art of QVT: A model transformation language standard. In: SCHÜRR, A.; NAGL, M.; ZÜNDORF, A. (Ed.). *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE*. Kassel, Germany: Springer, 2007. (Lecture Notes in Computer Science, v. 5088), p. 377–393. Cited in page 98.

LANO, K. et al. A survey of model transformation design patterns in practice. *Journal of Systems and Software*, v. 140, p. 48–73, 2018. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121218300438>. Cited in page 52.

LERNER, B. S.; HABERMANN, A. N. Beyond schema evolution to database reorganization. In: YONEZAWA, A. (Ed.). *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming, OOPSLA/ECOOP, Proceedings*. Ottawa, Canada: ACM, 1990. p. 67–76. Disponível em: <https://doi.org/10.1145/97945.97956>. Cited in page 96.

LÚCIO, L. et al. Model transformation intents and their properties. *Software & Systems Modeling*, v. 15, n. 3, p. 647–684, 2016. Disponível em: <https://doi.org/10.1007/s10270-014-0429-x>. Cited in page 52.

MACNAMARA, J. T.; MACNAMARA, J.; REYES, G. E. *The logical foundations of cognition*. New York, NY: Oxford University Press on Demand, 1994. (Vancouver Studies in Cognitive Science, 4). Cited in page 48.

MARKOVIC, S.; BAAR, T. Refactoring OCL annotated UML class diagrams. In: *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS, Proceedings*. Montego Bay, Jamaica: Springer, 2005. (Lecture Notes in Computer Science, v. 3713), p. 280–294. Disponível em: <https://doi.org/10.1007/11557432_21>. Cited in page 96.

MARTIN, J.; ODELL, J. J. *Object-oriented methods*. New Jersey, USA: Prentice hall PTR, 1994. Cited in page 18.

MEDEIROS, L. de et al. MIRROR: automatic R2RML mapping generation from relational databases. In: *Proc. 15th ICWE*. Rotterdam, The Netherlands: Springer, 2015. v. 9114, p. 326–343. Cited in page 83.

MOREIRA, J. L. R. et al. Ontowarehousing - multidimensional design supported by a foundational ontology: A temporal perspective. In: BELLATRECHE, L.; MOHANIA, M. K. (Ed.). *Data Warehousing and Knowledge Discovery - 16th International Conference, DaWaK. Proceedings*. Munich, Germany: Springer, 2014. (Lecture Notes in Computer Science, v. 8646), p. 35–44. Disponível em: <https://doi.org/10.1007/978-3-319-10160-6_4>. Cited in page 18.

OMG. *Model Driven Architecture (MDA) MDA Guide rev. 2.0*. 2014. OMG Document ormsc/2014-06-01. Disponível em: <https://www.omg.org/cgi-bin/doc?ormsc/14-06-01>. Cited in page 19.

ORIOL, X.; TENIENTE, E. Incremental checking of OCL constraints through SQL queries. In: *Proc. MODELS)*. Valencia, Spain: CEUR-WS.org, 2014. (CEUR Workshop Proceedings, v. 1285), p. 23–32. Cited in page 97.

PELAGATTI, G. et al. From the conceptual design of spatial constraints to their implementation in real systems. In: *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, NY, USA: Association for Computing Machinery, 2009. (GIS '09), p. 448–451. ISBN 9781605586496. Disponível em: <https://doi.org/10.1145/1653771.1653841>. Cited in page 97.

PENNEY, D. J.; STEIN, J. Class modification in the gemstone object-oriented DBMS. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, Proceedings*. Orlando, Florida, USA: ACM, 1987. p. 111–117. Disponível em: <https://doi.org/10.1145/38765.38817>. Cited in page 96.

PÉREZ, J.; ARENAS, M.; GUTIERREZ, C. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, v. 34, n. 3, p. 16:1–16:45, 2009. Disponível em: <https://doi.org/10.1145/1567274.1567278>. Cited in page 20.

PERGL, R.; SALES, T. P.; RYBOLA, Z. Towards OntoUML for software engineering: from domain ontology to implementation model. In: *Model and Data Engineering - Third International Conference, MEDI. Proceedings*. Amantea, Italy: Springer, 2013. (Lecture Notes in Computer Science, v. 8216), p. 249–263. Cited 2 times in pages 97 and 99.

PHILIPPI, S. Model driven generation and testing of object-relational mappings. *Journal of Systems and Software*, v. 77, p. 193–207, 2005. Cited 5 times in pages 19, 20, 21, 52, and 71.

POGGI, A. et al. Linking data to ontologies. *J. Data Semantics*, v. 10, p. 133–173, 2008. Cited 2 times in pages 20 and 74.

RYBOLA, Z. *Towards OntoUML for Software Engineering: Transformation of OntoUML into Relational Databases*. Prague, Czech Republic: PhD thesis, Czech Technical University in Prague, 2017. Cited 2 times in pages 70 and 97.

RYBOLA, Z.; PERGL, R. Towards OntoUML for software engineering: introduction to the transformation of OntoUML into relational databases. In: *Workshop on Enterprise and Organizational Modeling and Simulation*. Ljubljana, Slovenia: Springer, 2016. p. 67–83. Cited in page 70.

RYBOLA, Z.; PERGL, R. Towards OntoUML for software engineering: transformation of anti-rigid sortal types into relational databases. In: *International Conference on Model and Data Engineering*. Almería, Spain: Springer, 2016. p. 1–15. Cited in page 70.

RYBOLA, Z.; PERGL, R. Towards OntoUML for software engineering: transformation of rigid sortal types into relational databases. In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. Gdańsk, Poland: IEEE, 2016. p. 1581–1591. Cited in page 70.

RYBOLA, Z.; PERGL, R. Towards OntoUML for software engineering: Transformation of kinds and subkinds into relational databases. *Comput. Sci. Inf. Syst.*, v. 14, n. 3, p. 913–937, 2017. Cited in page 70.

SENDALL, S.; KOZACZYNSKI, W. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, v. 20, n. 5, p. 42–45, 2003. Cited in page 17.

SEQUEDA, J. F.; MIRANKER, D. P. Ultrawrap: SPARQL execution on relational data. *J. Web Semant.*, v. 22, p. 19–39, 2013. Cited in page 83.

SHAH, D.; SLAUGHTER, S. Transforming UML class diagrams into relational data models. In: *UML and the Unified Process*. USA: IGI Global, 2003. p. 217–236. Cited 2 times in pages 97 and 99.

SMITH, K. E.; ZDONIK, S. B. Intermedia: A case study of the differences between relational and object-oriented database systems. In: MEYROWITZ, N. K. (Ed.). *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Proceedings*. Orlando, Florida, USA: ACM, 1987. p. 452–465. Cited in page 25.

STEIMANN, F. On the representation of roles in object-oriented and conceptual modelling. *Data Knowledge Engineering*, v. 35, n. 1, p. 83–106, 2000. Cited in page 26.

STEIMANN, F. The role data model revisited. *Applied Ontology*, v. 2, n. 2, p. 89–103, 2007. Cited in page 26.

STONEBRAKER, M. Sql databases v. nosql databases. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 53, n. 4, p. 10–11, apr 2010. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/1721654.1721659>. Cited in page 103.

TEOREY, T. J.; YANG, D.; FRY, J. P. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, Association for Computing Machinery (ACM), v. 18, n. 2, p. 197–222, jun. 1986. Disponível em: <https://doi.org/10.1145/7474.7475>. Cited 4 times in pages 87, 88, 97, and 99.

TORRES, A. et al. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information & Software Technology*, v. 82, 2017. Cited 10 times in pages 17, 18, 22, 28, 30, 32, 34, 42, 52, and 71.

VERDONCK, M. *Ontology-driven conceptual modeling: Model comprehension, ontology selection, and method complexity*. Tese (Doutorado) — Ghent University, 2018. Cited 2 times in pages 22 and 38.

VERDONCK, M.; GAILLY, F. Insights on the use and application of ontology and conceptual modeling languages in ontology-driven conceptual modeling. In: COMYN-WATTIAU, I. et al. (Ed.). *Conceptual Modeling - 35th International Conference, ER, Proceedings*. Gifu, Japan: Springer, 2016. (Lecture Notes in Computer Science, v. 9974), p. 83–97. Cited in page 39.

VERDONCK, M. et al. Ontology-driven conceptual modeling: A systematic literature mapping and review. *Appl. Ontology*, v. 10, n. 3-4, p. 197–227, 2015. Cited in page 38.

VYŠNIAUSKAS, E. et al. Reversible lossless transformation from owl 2 ontologies into relational databases. *Information technology and control*, v. 40, n. 4, p. 293–306, 2011. Cited in page 70.

WIERINGA, R. J.; JONGE, W. de; SPRUIT, P. Using dynamic classes and role classes to model object migration. *TAPOS*, v. 1, n. 1, p. 61–83, 1995. Cited in page 26.

XIAO, G. et al. Ontology-based data access: A survey. In: LANG, J. (Ed.). *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence IJCAI*. Stockholm, Sweden: ijcai.org, 2018. p. 5511–5519. Cited in page 75.

XIAO, G. et al. Virtual knowledge graphs: An overview of systems and use cases. *Data Intell.*, v. 1, n. 3, p. 201–223, 2019. Cited 2 times in pages 10 and 75.

XIAO, G. et al. The virtual knowledge graph system Ontop. In: *The Semantic Web–ISWC: 19th International Semantic Web Conference, Proceedings, Part II*. Athens, Greece: Springer, 2020. p. 259–277. Cited in page 76.

XU, F. From lot's wife to a pillar of salt: Evidence that physical object is a sortal concept. *Mind & Language*, Wiley Online Library, v. 12, n. 3-4, p. 365–392, 1997. Cited in page 48.

XU, F.; CAREY, S. Infants metaphysics: The case of numerical identity. *Cognitive psychology*, Academic Press, v. 30, n. 2, p. 111–153, 1996. Cited in page 48.

YODER, J. W.; JOHNSON, R. E. The adaptive object-model architectural style. In: BOSCH, J. et al. (Ed.). *Software Architecture: System Design, Development and Maintenance, IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture (WICSA3)*. Montréal, Québec, Canada: Kluwer, 2002. (IFIP Conference Proceedings, v. 224), p. 3–27. Cited 3 times in pages 18, 35, and 42.

ZAMBORLINI, V.; GUIZZARDI, G. An ontologically-founded reification approach for representing temporally changing information in OWL. In: *11th International Symposium on Logical Formalizations of Commonsense Reasoning*. Agia Napa, Chipre: Elsevier, 2013. Cited in page 39.

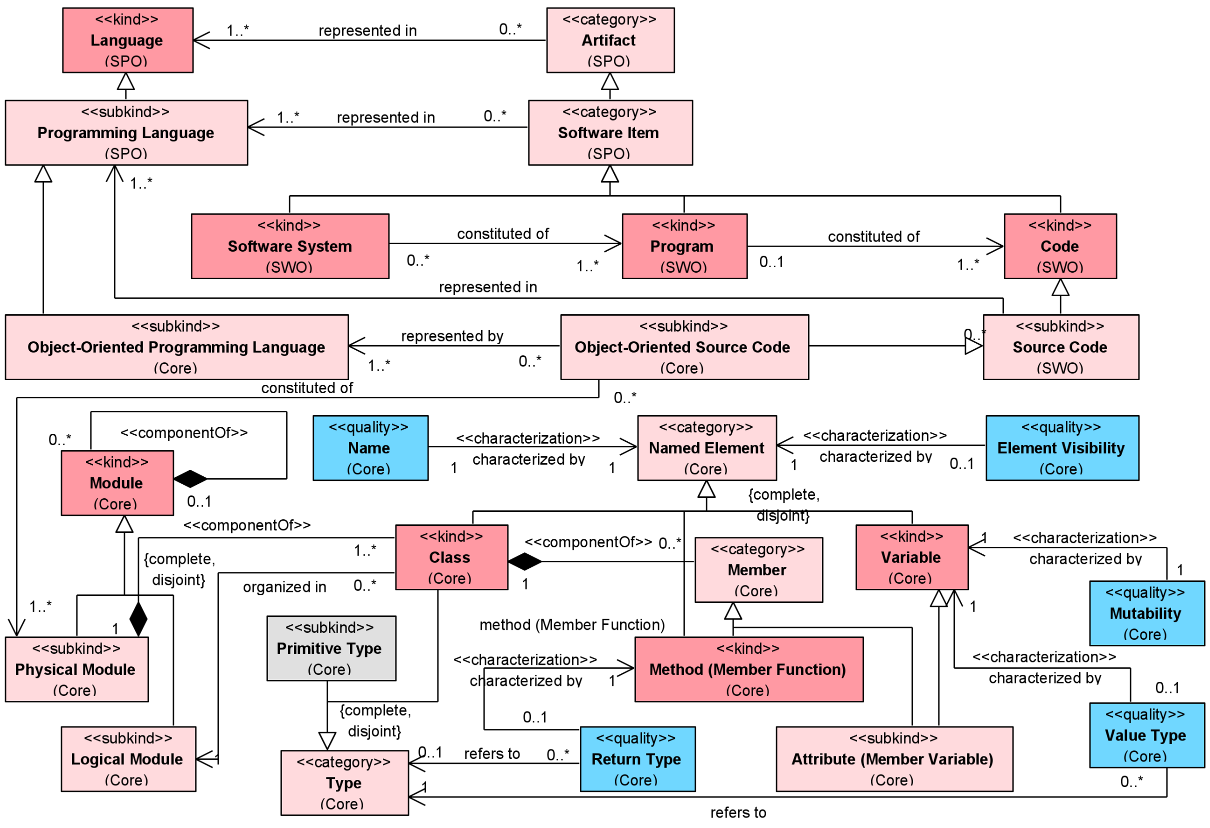
Appendices

APPENDIX A – Relational Schema Generated by One Table per Kind Strategy

```
CREATE TABLE IF NOT EXISTS organization (
         organization_id          INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        organization_type_id     INTEGER        NULL
,        name                     VARCHAR(20)    NOT NULL
,        address                  VARCHAR(20)    NOT NULL
,        credit_rating            DOUBLE         NULL
,        credit_limit             DOUBLE         NULL
,        is_corporate_customer    TINYINT(1)     NOT NULL DEFAULT FALSE
,        playground_size          INTEGER        NULL
,        capacity                 INTEGER        NULL
,        is_contractor            TINYINT(1)     NOT NULL DEFAULT FALSE
);

CREATE TABLE IF NOT EXISTS employment (
         employment_id            INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        organization_id          INTEGER        NOT NULL
,        person_id                INTEGER        NOT NULL
,        salary                   DOUBLE         NOT NULL
);

CREATE TABLE IF NOT EXISTS enrollment (
         enrollment_id            INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        person_id                INTEGER        NOT NULL
,        organization_id          INTEGER        NOT NULL
,        grade                    INTEGER        NOT NULL
);

CREATE TABLE IF NOT EXISTS person (
         person_id                INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        life_phase_id            INTEGER        NOT NULL
,        name                     VARCHAR(20)    NOT NULL
,        birth_date               DATE           NOT NULL
,        rg                       VARCHAR(20)    NULL
,        ci                       VARCHAR(20)    NULL
,        credit_rating            DOUBLE         NULL
,        credit_card              VARCHAR(20)    NULL
,        is_personal_customer     TINYINT(1)     NOT NULL DEFAULT FALSE
,        is_employee              TINYINT(1)     NOT NULL DEFAULT FALSE
);

CREATE TABLE IF NOT EXISTS supply_contract (
         supply_contract_id       INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        organization_customer_id INTEGER         NULL
,        person_id                INTEGER        NULL
,        organization_id          INTEGER        NOT NULL
,        contract_value           DOUBLE         NOT NULL
);

CREATE TABLE IF NOT EXISTS nationality (
         nationality_id           INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        nationality              VARCHAR(20)    NOT NULL
);

CREATE TABLE IF NOT EXISTS life_phase (
         life_phase_id            INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        life_phase               VARCHAR(20)    NOT NULL
);

CREATE TABLE IF NOT EXISTS organization_type (
         organization_type_id     INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        organization_type        VARCHAR(20)    NOT NULL
);

CREATE TABLE IF NOT EXISTS nationality_person (
         nationality_person_id    INTEGER        NOT NULL AUTO_INCREMENT PRIMARY
            KEY
,        person_id                INTEGER        NOT NULL
,        nationality_id           INTEGER        NOT NULL
);



ALTER TABLE organization ADD FOREIGN KEY ( organization_type_id ) REFERENCES
    organization_type ( organization_type_id );

ALTER TABLE employment ADD FOREIGN KEY ( organization_id ) REFERENCES
    organization ( organization_id );

ALTER TABLE employment ADD FOREIGN KEY ( person_id ) REFERENCES person (
    person_id );

ALTER TABLE enrollment ADD FOREIGN KEY ( person_id ) REFERENCES person (
    person_id );

ALTER TABLE enrollment ADD FOREIGN KEY ( organization_id ) REFERENCES
    organization ( organization_id );

ALTER TABLE person ADD FOREIGN KEY ( life_phase_id ) REFERENCES life_phase (
    life_phase_id );

ALTER TABLE supply_contract ADD FOREIGN KEY ( organization_customer_id )
    REFERENCES organization ( organization_id );

ALTER TABLE supply_contract ADD FOREIGN KEY ( person_id ) REFERENCES person (
    person_id );

ALTER TABLE supply_contract ADD FOREIGN KEY ( organization_id ) REFERENCES
```

```
    organization ( organization_id );

ALTER TABLE nationality_person ADD FOREIGN KEY ( person_id ) REFERENCES person (
     person_id );

ALTER TABLE nationality_person ADD FOREIGN KEY ( nationality_id ) REFERENCES
    nationality ( nationality_id );
```

APPENDIX B – Object-Oriented Code Ontology Project



<<kind>> Language (SPO)
<<category>> Artifact (SPO)
represented in
<<subkind>> Programming Language (SPO)
<<category>> Software Item (SPO)
<<kind>> Software System (SWO)
<<kind>> Program (SWO)
<<kind>> Code (SWO)
constituted of
<<subkind>> Object-Oriented Programming Language (Core)
<<subkind>> Object-Oriented Source Code (Core)
<<subkind>> Source Code (SWO)
represented by
<<kind>> Module (Core)
<<componentOf>>
<<quality>> Name (Core)
<<category>> Named Element (Core)
<<quality>> Element Visibility (Core)
<<characterization>>
characterized by
{complete, disjoint}
<<kind>> Class (Core)
<<category>> Member (Core)
<<kind>> Variable (Core)
<<quality>> Mutability (Core)
<<subkind>> Physical Module (Core)
<<subkind>> Logical Module (Core)
organized in
<<subkind>> Primitive Type (Core)
method (Member Function)
<<kind>> Method (Member Function) (Core)
<<quality>> Value Type (Core)
<<category>> Type (Core)
<<quality>> Return Type (Core)
<<subkind>> Attribute (Member Variable) (Core)
refers to

Figure 30 – Core Module.

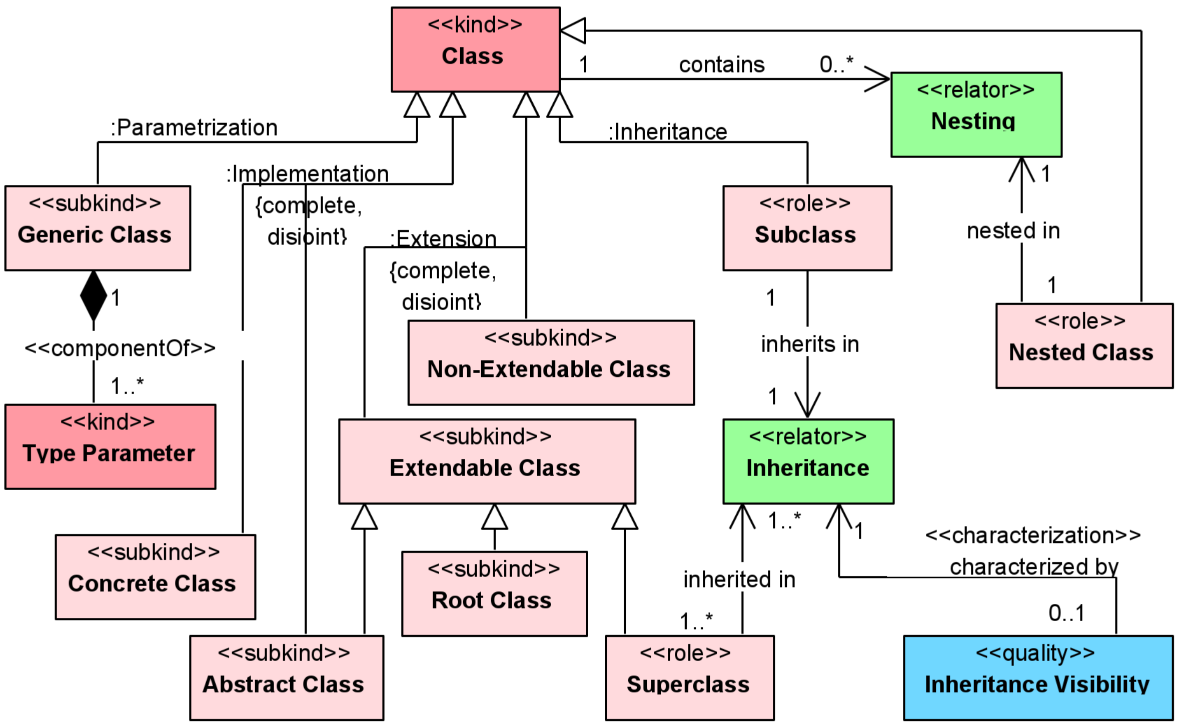
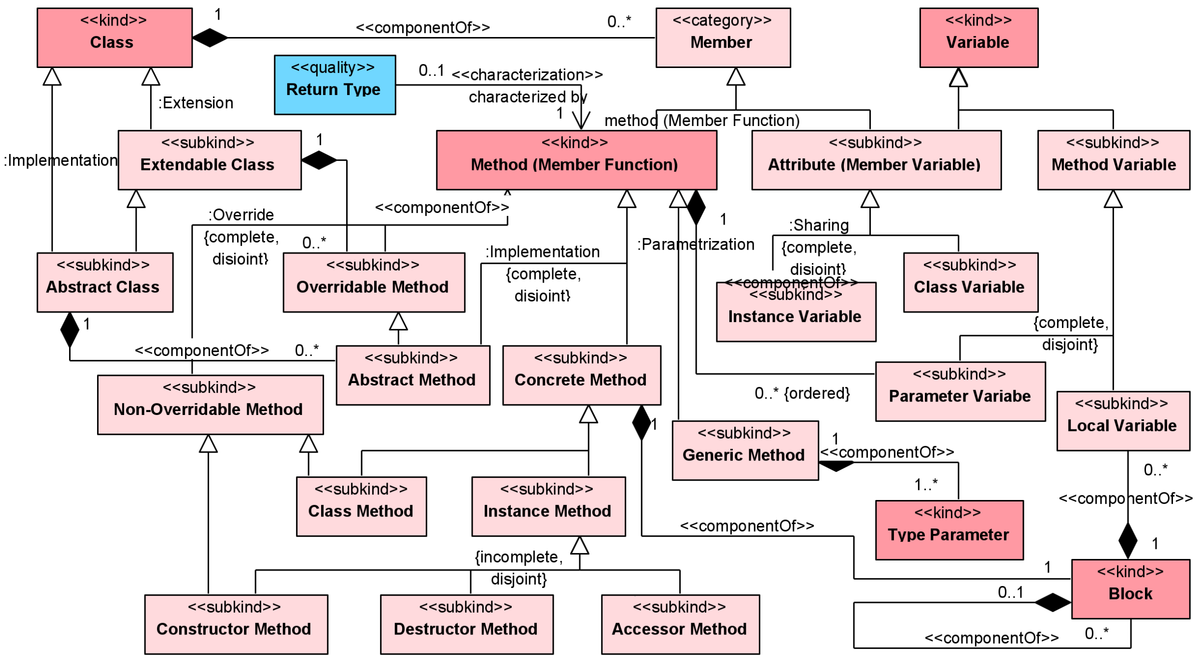


Figure 31 – Class Module.



<<kind>> Class
1
<<componentOf>>
0..*
<<category>> Member
<<kind>> Variable
<<quality>> Return Type
0..1
<<characterization>> characterized by
1
method (Member Function)
:Extension
:Implementation
<<subkind>> Extendable Class
1
<<kind>> Method (Member Function)
<<subkind>> Attribute (Member Variable)
<<subkind>> Method Variable
:Override {complete, disjoint}
<<componentOf>>
0..*
<<subkind>> Abstract Class
<<subkind>> Overridable Method
:Implementation {complete, disjoint}
:Parametrization
1
:Sharing {complete, disjoint}
<<componentOf>>
<<subkind>> Instance Variable
<<subkind>> Class Variable
1
<<componentOf>>
0..*
<<subkind>> Abstract Method
<<subkind>> Concrete Method
{complete, disjoint}
0..* {ordered}
<<subkind>> Parameter Variabe
<<subkind>> Non-Overridable Method
<<subkind>> Local Variable
1
<<subkind>> Generic Method
1
<<componentOf>>
1..*
<<kind>> Type Parameter
0..*
<<componentOf>>
<<subkind>> Class Method
<<subkind>> Instance Method
<<componentOf>>
1
{incomplete, disjoint}
1
<<kind>> Block
0..1
<<componentOf>>
0..*
<<subkind>> Constructor Method
<<subkind>> Destructor Method
<<subkind>> Accessor Method

Figure 32 – Class Member Module.

APPENDIX C – Queries used in our evaluations

C.1 Running Example Model

C.1.1 Queries for *One table per kind* strategy

Listing C.1 – Query 1.

```
select count(1)
from (
  select  person_id
  ,     credit_rating
  from  person
  where is_personal_customer = true

  union all

  select  organization_id
  ,     credit_rating
  from  organization
    where   is_corporate_customer = true
) as temp
;
```

Listing C.2 – Query 2.

```
select  count(1)
from(
  select  p.name person_name
  ,     e.grade
  ,     o.name organization_name
  ,     o.playground_size
  from  person p
  join  enrollment e
        on  p.person_id   = e.person_id
  join  organization o
        on  e.organization_id = o.organization_id
) as temp
;
```

Listing C.3 – Query 3.

```
select count(1)
from(
    select  p.name brazilian_name
  ,   p.birth_date
  ,   p.rg
  ,   em.salary
  ,   o.name organization_name
  ,   p2.name italian_name
  ,   sc.contract_value
```

```
    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'
)as temp
;
```

Listing C.4 – Query 4.

```
select count(1)
from(
    select  o.name
    ,     o.address
    ,     o.credit_rating
    ,     o.credit_limit
    ,     capacity
    ,     playground_size
    from  organization o
) as temp
;
```

Listing C.5 – Query 5.

```
select count(1)
from(
    select  p.name person_name
    ,   sc.contract_value
    ,   o.name organization_name
    from  person p
    join  supply_contract sc
        on  p.person_id              = sc.person_id
    join  organization o
        on  sc.organization_id       = o.organization_id
            and o.organization_type_enum  = 'HOSPITAL'
    where p.ci                = '433019254'
) as temp
;
```

C.1.2 Queries for *One table per class* strategy

Listing C.6 – Query 1.

```
select count(1)
from (
  select  pc.named_entity_id
  ,     c.credit_rating
  from  named_entity ne
  join  personal_customer pc
    on  ne.named_entity_id  = pc.named_entity_id
  join  customer c
    on  pc.customer_id    = c.customer_id

  union all

  select  cc.named_entity_id
  ,     c.credit_rating
  from  named_entity ne
  join  corporate_customer cc
    on  ne.named_entity_id  = cc.named_entity_id
  join  customer c
    on  cc.customer_id    = c.customer_id

) as temp
;
```

Listing C.7 – Query 2.

```
select  count(1)
from(
  select  ne_p.name person_name
  ,     e.grade
  ,     ne_o.name organization_name
  ,     ps.playground_size
  from  named_entity ne_p
  join  enrollment e
        on  ne_p.named_entity_id    = e.named_entity_id
  join  named_entity ne_o
        on  e.named_entity_primary_school_id  = ne_o.named_entity_id
  join  primary_school ps
        on  ne_o.named_entity_id  = ps.named_entity_id
) as temp
;
```

Listing C.8 – Query 3.

```
select count(1)
from(
    select  ne_p.*
    from  named_entity ne_p
    join  person p
      on  ne_p.named_entity_id  = p.named_entity_id
    join  brazilian_citizen bc
          on  p.named_entity_id   = bc.named_entity_id
    join  employment em
          on  p.named_entity_id     = em.named_entity_id
    join  hospital h
        on  em.named_entity_organization_id = h.named_entity_id
  join  named_entity ne_o
      on  h.named_entity_id       = ne_o.named_entity_id
     join supply_contract sc
          on  h.named_entity_id     = sc.named_entity_id
  join  customer c
      on  sc.customer_id         = c.customer_id
  join  personal_customer pc
      on  c.customer_id       = pc.customer_id
     join named_entity ne_i
          on  pc.named_entity_id    = ne_i.named_entity_id
     join italian_citizen ic
```

```sql
            on  ne_i.named_entity_id     = ic.named_entity_id
) as temp
;
```

Listing C.9 – Query 4.

```sql
select count(1)
from(
    select  ne.name
    ,        o.address
    ,     c.credit_rating
    ,     cc.credit_limit
    ,     h.capacity
    ,     pc.playground_size
    from  named_entity ne
    join organization o
        on ne.named_entity_id         = o.named_entity_id
    left join corporate_customer cc
        on  o.named_entity_id = cc.named_entity_id
    left  join customer c
  on  cc.customer_id    = c.customer_id
    left join hospital h
        on  o.named_entity_id = h.named_entity_id
    left join primary_school pc
        on  o.named_entity_id = pc.named_entity_id
) as temp
;
```

Listing C.10 – Query 5.

```sql
select count(1)
from(
    select  ne.name person_name
    ,     sc.contract_value
    ,     ne_o.name organization_name
    from  italian_citizen ic
  join  named_entity ne
      on  ic.named_entity_id      = ne.named_entity_id
  join  personal_customer pc
      on  ne.named_entity_id      = pc.named_entity_id
  join  supply_contract sc
          on  pc.customer_id         = sc.customer_id
    join  named_entity ne_o
          on  sc.named_entity_id      = ne_o.named_entity_id
  join  hospital h
      on  sc.named_entity_id      = h.named_entity_id
    where ic.ci = '999984780'
) as temp
;
```

C.1.3 Queries for *One table per concrete class* strategy

Listing C.11 – Query 1.

```sql
select count(1)
    from (
    select  person_id
    ,     credit_rating
    from  personal_customer

    union all

    select  organization_id
    ,     credit_rating
    from  corporate_customer
) as temp;
```

Listing C.12 – Query 2.

```
select  count(1)
from(
  select  p.name person_name
  ,     e.grade
  ,     o.name organization_name
  ,     ps.playground_size
  from  person p
  join  child c
    on  p.person_id   = c.person_id
  join  enrollment e
        on  c.person_id   = e.person_id
  join  organization o
        on  e.organization_id = o.organization_id
  join  primary_school ps
        on  o.organization_id = ps.organization_id
) as temp
;
```

Listing C.13 – Query 3.

```
select count(1)
from(
    select  p.name brazilian_name
    ,     p.birth_date
    ,     bc.rg
    ,     em.salary
    ,     o.name organization_name
    ,     p2.name italian_name
    ,     sc.contract_value
    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  em.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
) as temp
;
```

Listing C.14 – Query 4.

```
select count(1)
from(
    select  o.name
    ,           o.address
    ,     cc.credit_rating
    ,     cc.credit_limit
    ,     h.capacity
    ,     pc.playground_size
    from  corporate_customer cc
    join  organization o
          on  cc.organization_id  = o.organization_id
    left join hospital h
          on  o.organization_id = h.organization_id
    left join primary_school pc
          on  o.organization_id = pc.organization_id
) as temp
;
```

Listing C.15 – Query 5.

```
select count(1)
from(
    select  p.name person_name
    ,     sc.contract_value
    ,     o.name organization_name
    from  italian_citizen ic
    join  person p
          on  ic.person_id    = p.person_id
    join  supply_contract sc
          on  p.person_id     = sc.person_id
    join  organization o
          on  sc.organization_id  = o.organization_id
    join  hospital h
      on  o.organization_id = h.organization_id
    where ic.ci = '999984780'
) as temp
;
```

C.1.4 Queries for *One flattened table per concrete class* strategy

Listing C.16 – Query 1.

```
select count(1)
from (
    select  person_id
    ,     credit_rating
    from  personal_customer

    union all

    select  organization_id
    ,     credit_rating
    from  corporate_customer
) as temp;
```

Listing C.17 – Query 2.

```
select  count(1)
from(
  select  c.name person_name
  ,     e.grade
  ,     ps.name organization_name
  ,     ps.playground_size
  from  child c
  join  enrollment e
        on  c.person_id   = e.person_id
  join  primary_school ps
        on  e.organization_id = ps.organization_id
) as temp
;
```

Listing C.18 – Query 3.

```
select count(1)
from(
  select  bc.name brazilian_name
  ,     bc.birth_date
  , bc.rg
  , em.salary
  , h.name organization_name
  , ic.name italian_name
  , sc.contract_value
  from  brazilian_citizen bc
  join  employment em
    on  bc.person_id      = em.person_id
  join  hospital h
    on  em.organization_hospital_id = h.organization_id
  join  supply_contract sc
    on  h.organization_id   = sc.organization_id
  join  italian_citizen ic
    on  sc.personal_customer_id   = ic.person_id
) as temp
;
```

Listing C.19 – Query 4.

```
select count(1)
from(
    select  o.name
    ,   o.address
    ,     cc.credit_rating
    ,     cc.credit_limit
    ,     h.capacity
    ,     pc.playground_size
    from  organization o
  left join corporate_customer cc
      on  o.organization_id   = cc.organization_id
    left join hospital h
          on  o.organization_id = h.organization_id
    left join primary_school pc
          on  o.organization_id = pc.organization_id
) as temp
;
```

Listing C.20 – Query 5.

```
select count(1)
from(
    select  ic.name person_name
    ,     sc.contract_value
    ,     h.name organization_name
    from  italian_citizen ic
    join  supply_contract sc
          on  ic.person_id      = sc.personal_customer_id
    join  hospital h
      on  sc.organization_id    = h.organization_id
    where ic.ci = '999984780'
) as temp
;
```

C.1.5 Queries for *One flattened table per leaf class* strategy

Listing C.21 – Query 1.

```
select count(1)
from(
  select  person_id
    ,     credit_rating
    from  personal_customer

  union all

  select  organization_id
    ,     credit_rating
    from  corporate_customer
) as temp;
```

Listing C.22 – Query 2.

```
select  count(1)
from(
  select  c.name person_name
  ,     e.grade
  ,     ps.name organization_name
  ,     ps.playground_size
  from  child c
  join  enrollment e
        on  c.person_id   = e.person_id
  join  primary_school ps
        on  e.organization_id = ps.organization_id
) as temp
;
```

Listing C.23 – Query 3.

```
select count(1)
from(
    select  bc.name brazilian_name
    ,     bc.birth_date
    ,     bc.rg
    ,     em.salary
    ,     h.name organization_name
    ,     ic.name italian_name
    ,     sc.contract_value
    from  brazilian_citizen bc
    join  employment em
          on  bc.person_id      = em.person_id
    join  hospital h
      on  em.organization_hospital_id   = h.organization_id
    join  supply_contract sc
          on  h.organization_id   = sc.organization_id
    join  italian_citizen ic
          on  sc.personal_customer_id = ic.person_id
            and ic.is_only_person       = false
) as temp
;
```

Listing C.24 – Query 4.

```
select count(1)
from(
    select  cc.name
    ,      cc.address
    ,      cc.credit_rating
    ,      cc.credit_limit
    ,      h.capacity
    ,      pc.playground_size
    from  corporate_customer cc
    left join hospital h
          on  cc.organization_id  = h.organization_id
    left join primary_school pc
          on  cc.organization_id  = pc.organization_id

  union

  select  h.name
  ,      h.address
  ,      0 credit_rating
  ,      0 credit_limit
  ,      h.capacity
  ,      0 playground_size
  from  hospital h
  where not exists( select 1
            from corporate_customer cc
            where cc.organization_id  = h.organization_id
             )

  union

  select  ps.name
  ,      ps.address
  ,      0 credit_rating
  ,      0 credit_limit
  ,      0 capacity
  ,      ps.playground_size
  from  primary_school ps
  where not exists (select 1
            from corporate_customer cc
            where ps.organization_id  = cc.organization_id
            )

) as temp
;
```

Listing C.25 – Query 5.

```
select count(1)
from(
    select  ic.name person_name
    ,      sc.contract_value
    ,      h.name organization_name
    from  italian_citizen ic
```

```
  join  supply_contract sc
          on  ic.person_id      = sc.personal_customer_id
  join  hospital h
      on  sc.organization_id    = h.organization_id
    where ic.ci = '999984780'
) as temp
;
```

C.1.6 Queries for *Generic structure* strategy

Listing C.26 – Query 1.

```
select   count(1)
from(
  select  v.*
  from  value customer
  join  value v
    on  customer.instance_id    = v.instance_id
  where customer.attribute_id       in (19, 22)
  and (
      ( customer.attribute_id = 19 and v.attribute_id in (19, 2, 20) )
      or
      ( customer.attribute_id = 22 and v.attribute_id in (22, 12, 23) )
    )
  order by 1,2
) as temp
;
```

Listing C.27 – Query 2.

```
select  count(1)
from(
  select  p.val person_name
  ,     e_grade.val grade
  ,   o_name.val organization_name
  ,   s.val playground_size
  -- child
  from  value c
  -- person
  join  value p
    on  c.instance_id   = p.instance_id
    and p.attribute_id    = 2 -- name
  -- enrollment
  join  value e
    on  c.val        = e.val
    and e.attribute_id    = 31 -- person_id (enrollment)
  join  value e_grade
    on  e.instance_id     = e_grade.instance_id
    and e_grade.attribute_id  = 32 -- grade
  join  value e_org
    on  e.instance_id     = e_org.instance_id
    and e_org.attribute_id    = 30 -- organization_id (enrollment)
  -- organization
  join  value o
    on  e_org.val       = o.val
    and o.attribute_id      = 11 -- organization_id (organization)
  join  value o_name
    on  o.instance_id     = o_name.instance_id
    and o_name.attribute_id   = 12
  -- primary_school
```

```sql
  join  value s
    on  o.instance_id     = s.instance_id
    and s.attribute_id      = 17 -- playground_size

  where c.attribute_id    = 8 -- person_id (child)
  ) as temp
;
```

Listing C.28 – Query 3.

```sql
select  count(1)
from(
  select  person_n.val    brazilian_name
  ,   person_b.val    birth_date
  ,   person_rg.val     rg
  ,   employment_s.val  salary
  ,   organization_n.val  organization_name
  ,   supply_contract_v.val contract_value
  ,   italian_n.val   italian_name

  from  value brazilian
  -- person
  join  value person_n
      on  brazilian.instance_id     = person_n.instance_id
      and person_n.attribute_id     = 2 -- name
  join  value person_b
      on  brazilian.instance_id     = person_b.instance_id
      and person_b.attribute_id     = 3 -- birth_date
  join  value person_rg
      on  brazilian.instance_id     = person_rg.instance_id
      and person_rg.attribute_id      = 5 -- rg
  -- employment
  join  value employment
      on  brazilian.val         = employment.val
      and employment.attribute_id     = 27 -- person_id (employment)
  join  value employment_s
      on  employment.instance_id      = employment_s.instance_id
      and employment_s.attribute_id   = 28 -- salary
  join  value employment_o
      on  employment.instance_id      = employment_o.instance_id
      and employment_o.attribute_id   = 26 -- organization_id (employment)
  -- organization
  join  value organization
      on  employment_o.val        = organization.val
      and organization.attribute_id   = 14 -- orgnization_id (hospital)
  join  value organization_n
      on  organization.instance_id    = organization_n.instance_id
      and organization_n.attribute_id   = 12 -- name
  -- supply_contract
  join  value supply_contract
      on  employment_o.val        = supply_contract.val
      and supply_contract.attribute_id  = 36 -- organization_id (supply_contract)
  join  value supply_contract_v
      on  supply_contract.instance_id     = supply_contract_v.instance_id
      and supply_contract_v.attribute_id    = 37   -- contract_value
  join  value supply_contract_p
      on  supply_contract.instance_id     = supply_contract_p.instance_id
      and supply_contract_p.attribute_id    = 35   -- person_id (supply_contract)
  -- italian
  join  value italian
      on  supply_contract_p.val       = italian.val
      and italian.attribute_id        = 6 -- person_id (italian_citizen)
  join  value italian_n
      on  italian_n.instance_id          = italian_n.instance_id
      and italian_n.attribute_id         = 2 -- name

  where brazilian.attribute_id       = 4 -- 4 person_id (brazilian_citizen)
  ) as temp
;
```

Listing C.29 – Query 4.

```
select count(1)
from(
    select  organization_n.val      organization_name
  ,   organization_a              organization_address
  ,   customer_r.val         credit_rating
  ,   customer_l.val         credit_limit
  ,   hospital.val         capacity
  ,   school.val           playground_size
  from  value organization
  -- organization
  join  value organization_n
      on  organization.instance_id       = organization_n.instance_id
      and organization_n.attribute_id      = 12 -- name
  join  value organization_a
      on  organization.instance_id       = organization_a.instance_id
      and organization_a.attribute_id      = 13 -- address
  -- customer
  left join value customer_r
      on  organization.instance_id       = customer_r.instance_id
      and customer_r.attribute_id        = 23 -- credit_rating
  left join value customer_l
      on  organization.instance_id       = customer_l.instance_id
      and customer_l.attribute_id        = 24 -- credit_limit
  -- hospital
  left join value hospital
      on  organization.instance_id       = hospital.instance_id
      and hospital.attribute_id        = 15 -- capacity
  -- primary_chool
  left join value school
      on  organization.instance_id       = school.instance_id
      and school.attribute_id          = 17 -- playground_size
  where organization.attribute_id = 11 -- organization_id
) as temp
;
```

Listing C.30 – Query 5.

```
select  count(1)
from(
    select  person_n.val        person_name
    ,   contract_v.val        contract_value
    ,   organization_n.val      organization_name
    from  value italian
    -- person
    join  value person
        on  italian.instance_id      = person.instance_id
        and person.attribute_id      = 1
    join  value person_n
        on  italian.instance_id      = person_n.instance_id
        and person_n.attribute_id    = 2 -- name (person)
    -- supply_contract
    join  value contract
        on   person.val             = contract.val
        and  contract.attribute_id     = 35 -- person_id (supply_contract)
    join  value contract_v
        on   contract.instance_id      = contract_v.instance_id
        and  contract_v.attribute_id     = 37 -- contract_value
    join  value contract_o
        on   contract.instance_id      = contract_o.instance_id
        and  contract_o.attribute_id     = 36 -- organization_id (supply_contract)
    -- hospital
    join  value hospital
        on   contract_o.val          = hospital.val
        and  hospital.attribute_id     = 14 -- organization_id (hospital)
    join  value organization_n
        on   hospital.instance_id      = organization_n.instance_id
        and  organization_n.attribute_id   = 12 -- name (organization)

    where italian.val       = '229602246'
  ) as temp
;
```

C.2 OOC-O Project

C.2.1 Queries for *One table per kind* strategy

Listing C.31 – Query 1.

```
select  count(1)
from  (
    select  class_id
    ,   name
    from  class

    union all

    select  method_member_function_id
    ,   name
    from  method_member_function

    union all

    select  variable_id
    ,   name
    from  variable
  ) as temp;
```

Listing C.32 – Query 2.

```
select  count(1)
from  (
    select  c.name    class_name
    , m.name    method_name
    , v.value_type
    from  class c
    join  method_member_function m
      on  c.class_id        = m.class_member_id
    join  variable v
      on  m.method_member_function_id = v.method_member_function_id
    where m.implementation_enum1       = 'CONCRETEMETHOD'
  ) as temp;
```

Listing C.33 – Query 3.

```
select  count(1)
from  (
    select  pr.program_id
    ,   va.name
    from  program pr
    join  code co
      on  pr.program_id   = co.program_id
    join  object_oriented_source_code_physical_module  oo
      on  co.code_id    = oo.code_id
    join  module mo
      on  oo.module_id    = mo.module_id
    join  class cl
      on  mo.module_id    = cl.module_physical_module_id
    join  variable va
      on  cl.class_id   = va.class_id
    where cl.implementation_enum  = 'CONCRETECLASS'
  ) as temp;
```

Listing C.34 – Query 4.

```
select  count(1)
from  (
    select  module_id, module_type_enum
    from  module
  ) as temp;
```

Listing C.35 – Query 5.

```
select  count(1)
from  (
    select  po.*
    from  variable va
    join  class cl
      on  va.class_id   = cl.class_id
    join  module mo
      on  module_physical_module_id   = mo.module_id
    join  object_oriented_source_code_physical_module oo
      on  mo.module_id    = oo.module_id
    join  code co
      on  oo.code_id    = co.code_id
    join  program po
      on  co.program_id   = po.program_id
    where va.name       = 'XLACXMYSCL'
  ) as temp;
```

C.2.2 Queries for *One table per concrete class* strategy

Listing C.36 – Query 1.

```
select  count(1)
from  (
    select  class_id
    ,   name
    from  class

    union all

    select  method_member_function_id
    ,   name
    from  method_member_function

    union all

    select  variable_id
    ,   name
    from  variable
  ) as temp;
```

Listing C.37 – Query 2.

```
select  count(1)
from  (
    select  cl.name   class_name
    ,   mmf.name  method_name
    ,     va.value_type
    ,   va.variable_id
    from  class cl
    join  concrete_class cc
      on  cl.class_id     = cc.class_id
    join  method_member_function mmf
      on  cc.class_id     = mmf.class_id
    join  parameter_variabe pv
      on  mmf.method_member_function_id = pv.method_member_function_id
    join  variable va
      on  pv.variable_id      = va.variable_id

        union all

    select  cl.name   class_name
    ,   mmf.name  method_name
    ,     v.value_type
    ,   v.variable_id
    from  class cl
    join  concrete_class cc
        on  cl.class_id     = cc.class_id
    join  method_member_function mmf
        on  cc.class_id     = mmf.class_id
    join  concrete_method cm
        on  mmf.method_member_function_id = cm.method_member_function_id
    join  block b
        on  cm.block_id     = b.block_id
    join  local_variable lv
        on  b.block_id      = lv.block_id
    join  variable v
        on  lv.variable_id    = v.variable_id
  ) as temp;
```

Listing C.38 – Query 3.

```
select  count(1)
from  (
    select  pr.program_id
    ,   va.name
    from  program pr
    join  code co
      on  pr.program_id   = co.program_id
    join  object_oriented_source_code_physical_module oosc
      on  co.code_id    = oosc.code_id
    join  class cl
      on  oosc.module_id    = cl.module_physical_module_id
    join  concrete_class cc
      on  cl.class_id   = cc.class_id
    join  attribute_member_variable amv
      on  cl.class_id   = amv.class_id
    join  variable va
      on  amv.variable_id   = va.variable_id

    union all

    select  pr.program_id
    ,   v.name
    from  program pr
    join  code co
        on  pr.program_id     = co.program_id
    join  object_oriented_source_code_physical_module oosc
        on  co.code_id        = oosc.code_id
    join  class cl
        on  oosc.module_id       = cl.module_physical_module_id
    join  concrete_class cc
        on  cl.class_id       = cc.class_id
    join  method_member_function mmf
        on  cc.class_id       = mmf.class_id
    join  concrete_method cm
        on  mmf.method_member_function_id = cm.method_member_function_id
    join  block b
        on  cm.block_id     = b.block_id
    join  local_variable lv
        on  b.block_id      = lv.block_id
    join  variable v
        on  lv.variable_id    = v.variable_id

    union all

    select  pr.program_id
    ,   v.name
    from  program pr
    join  code co
        on  pr.program_id     = co.program_id
    join  object_oriented_source_code_physical_module oosc
        on  co.code_id        = oosc.code_id
    join  class cl
        on  oosc.module_id       = cl.module_physical_module_id
    join  concrete_class cc
        on  cl.class_id       = cc.class_id
    join  method_member_function mmf
        on  cc.class_id       = mmf.class_id
    join  parameter_variabe pv
        on  mmf.method_member_function_id   = pv.method_member_function_id
    join  variable v
        on  pv.variable_id    = v.variable_id
  ) as temp;
```

Listing C.39 – Query 4.

```
select  count(1)
from  (
    select  module_id, 'PHYSICALMODULE'
    from  physical_module

    UNION ALL

    select  module_id, 'LOGICALMODULE'
    from  logical_module
  ) as temp;
```

Listing C.40 – Query 5.

```
select count(1)
from  (
    select  po.*
    from  variable va
    join  attribute_member_variable amv
        on  va.variable_id    = amv.variable_id
    join  class cl
      on  amv.class_id    = cl.class_id
    join  object_oriented_source_code_physical_module oopm
        on  cl.module_physical_module_id    = oopm.module_id
    join  code co
      on  oopm.code_id    = co.code_id
    join  program po
      on  co.program_id   = po.program_id
    where va.name       = 'XLACXMYSCL'

    union

    select  po.*
    from  variable va
    join  parameter_variabe pv
        on  va.variable_id    = pv.variable_id
    join  method_member_function mmf
        on  pv.method_member_function_id    = mmf.method_member_function_id
    join  class cl
      on  mmf.class_id    = cl.class_id
    join  object_oriented_source_code_physical_module oopm
        on  cl.module_physical_module_id    = oopm.module_id
    join  code co
      on  oopm.code_id    = co.code_id
    join  program po
      on  co.program_id   = po.program_id
    where va.name       = 'XLACXMYSCL'

    union

    select  po.*
    from  variable va
    join  local_variable lv
        on  va.variable_id    = lv.variable_id
    join  concrete_method cm
        on  lv.block_id     = cm.block_id
    join  method_member_function mmf
        on  cm.method_member_function_id    = mmf.method_member_function_id
    join  class cl
      on  mmf.class_id    = cl.class_id
    join  object_oriented_source_code_physical_module oopm
        on  cl.module_physical_module_id    = oopm.module_id
    join  code co
      on  oopm.code_id    = co.code_id
    join  program po
```

```sql
      on   co.program_id    = po.program_id
    where va.name        = 'XLACXMYSCL'
  ) as temp;
```

C.2.3 Queries for *One flattened table per leaf class* strategy

Listing C.41 – Query 1.

```sql
select  count(1)
from  (
    select  abstract_class_id
    ,   name
    from  abstract_class

    union all

    select  concrete_class_id
    ,   name
    from  concrete_class

    union all

    select  generic_method_id
    ,   name
    from  generic_method

    union all

    select  instance_variable_id
    ,   name
    from  instance_variable

    union all

    select  parameter_variabe_id
    ,   name
    from  parameter_variabe

    union all

    select  class_variable_id
    ,   name
    from  class_variable

    union all

    select  local_variable_id
    ,   name
    from  local_variable
  ) as temp;
```

Listing C.42 – Query 2.

```sql
select  count(1)
from  (
      select  cc.name   class_name
      ,   gm.name   method_name
      ,   pv.value_type
      from  concrete_class cc
      join  generic_method gm
```

```
            on  cc.concrete_class_id    = gm.generic_class_id
        join  parameter_variabe pv
            on  gm.generic_method_id    = pv.generic_method_id

        union all

        select  cc.name   as class_name
        ,   cm.name   as method_name
        ,   lv.value_type
        from  concrete_class cc
        join  (
              select  concrete_class_id as class_id
              ,   name
              ,   block_id
              from  constructor_method

              union all

              select  concrete_class_id
              ,   name
              ,   block_id
              from  destructor_method

              union all

              select  concrete_class_id
              ,   name
              ,   block_id
              from  accessor_method

              union all

              select  concrete_class_id
              ,   name
              ,   block_id
              from  class_method
          ) as cm
            on  cc.concrete_class_id    = cm.class_id
        join  local_variable lv
            on  cm.block_id   = lv.block_id
    ) as temp;
```

Listing C.43 – Query 3.

```
select  count(1)
from  (
    select  pr.program_id
    , amv.name
    from  program pr
    join  object_oriented_source_code oosc
      on  pr.program_id       = oosc.program_id
    join  object_oriented_source_code_physical_module  oopm
      on  oosc.object_oriented_source_code_id   = oopm.object_oriented_source_code_id
    join  physical_module pm
      on  oopm.physical_module_id       = pm.physical_module_id
    join  concrete_class cc
      on  pm.physical_module_id       = cc.physical_module_id
    join  (
          select  instance_variable_id
          , concrete_class_id
          , name
          from  instance_variable

          union

          select  class_variable_id
          , concrete_class_id
          , name
          from  class_variable
      ) as amv
        on  cc.concrete_class_id        = amv.concrete_class_id

    union all

    select  pr.program_id
    , cm.name
    from  program pr
    join  object_oriented_source_code oosc
      on  pr.program_id       = oosc.program_id
    join  object_oriented_source_code_physical_module  oopm
      on  oosc.object_oriented_source_code_id   = oopm.object_oriented_source_code_id
    join  physical_module pm
      on  oopm.physical_module_id       = pm.physical_module_id
    join  concrete_class cc
      on  pm.physical_module_id       = cc.physical_module_id
    join  (
          select  concrete_class_id as class_id
          ,   name
          ,   block_id
          from  constructor_method

          union all

          select  concrete_class_id
          ,   name
          ,   block_id
          from  destructor_method

          union all

          select  concrete_class_id
          ,   name
          ,   block_id
          from  accessor_method

          union all

          select  concrete_class_id
          ,   name
          ,   block_id
          from  class_method
      ) as cm
        on  cc.concrete_class_id    = cm.class_id
    join  local_variable lv
        on  cm.block_id   = lv.block_id

    union all

    select  pr.program_id
    , lv.name
    from  program pr
    join  object_oriented_source_code oosc
      on  pr.program_id       = oosc.program_id
    join  object_oriented_source_code_physical_module  oopm
      on  oosc.object_oriented_source_code_id   = oopm.object_oriented_source_code_id
    join  physical_module pm
      on  oopm.physical_module_id       = pm.physical_module_id
    join  concrete_class cc
      on  pm.physical_module_id        = cc.physical_module_id
    join  generic_method gm
      on  cc.concrete_class_id     = gm.concrete_class_id
    join  parameter_variabe lv
        on  gm.generic_method_id     = lv.generic_method_id

  ) as temp;
```

Listing C.44 – Query 4.

```
select  count(1)
from  (
    select  physical_module_id, 'PHYSICALMODULE'
    from  physical_module

    UNION ALL

    select  logical_module_id, 'LOGICALMODULE'
    from  logical_module
  ) as temp;
```

Listing C.45 – Query 5.

```
select  count(1)
from  (
  select  pr.*
  from  program pr
  join  object_oriented_source_code oosc
    on  pr.program_id       = oosc.program_id
  join  object_oriented_source_code_physical_module  oopm
    on  oosc.object_oriented_source_code_id   = oopm.object_oriented_source_code_id
  join  (
      select  concrete_class_id as class_id
      , physical_module_id
      from  concrete_class

      union all

      select  abstract_class_id as class_id
      ,   physical_module_id
      from  abstract_class
    ) as cl
      on  oopm.physical_module_id      = cl.physical_module_id

  join  (
      select  case when concrete_class_id is not null
            then concrete_class_id
            else abstract_class_id
          end as class_id
      ,   name

      from  instance_variable
      where name = 'LZERXRWANL'

      union all

      select  case when concrete_class_id is not null
            then concrete_class_id
            else abstract_class_id
          end as class_id
      ,   name
      from  class_variable
      where name = 'LZERXRWANL'
```

APPENDIX D – SPARQL queries used in Chapter 5

Listing D.1 – Retrieve the credit rating of each customer.

```
PREFIX : <https://example.com#>

SELECT ?id1 ?creditRating{

  ?id1      a                :Customer ;
          :creditRating   ?creditRating .
}
```

Listing D.2 – Retrieve the name of each child along with the playground size of the schools in which the child is enrolled.

```
PREFIX : <https://example.com#>

SELECT ?name_child ?grade ?name_school ?size{

  ?id1      a                 :Child ;
            :name             ?name_child .

  ?id2      a                 :Enrollment ;
            :grade            ?grade ;
            :hasChild         ?id1 ;
            :hasPrimarySchool ?id3.

  ?id3      a                 :PrimarySchool ;
            :name             ?name_school ;
            :playgroundSize   ?size.
}
```

Listing D.3 – Retrieve 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.

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

Listing D.4 – Retrieve all data of organizations regardless of whether it is registered as a Hospital or Primary School.

```
PREFIX : <https://example.com#>

SELECT ?name ?capacity ?playgroundSize ?creditRating ?creditLimit
WHERE
{
    ?organization    a              :CorporateCustomer ;
                     :name           ?name ;
                     :creditRating   ?creditRating ;
                     :creditLimit    ?creditLimit .
    OPTIONAL { ?organization     a            :Hospital;
                                :capacity     ?capacity.
    }
    OPTIONAL { ?organization     a                :PrimarySchool;
                                :playgroundSize ?playgroundSize .
    }
}
```

Listing D.5 – Retrieve the CI of an Italian citizen the name of the Hospital with which has a contract and the value of that contract.

```
PREFIX : <https://example.com#>

select ?personName ?CI ?contract_value ?organizationName{

  ?id1   a                :ItalianCitizen ;
         :name            ?personName ;
         :CI              ?CI.

  ?id1    a               :PersonalCustomer.

  ?id2    a               :SupplyContract ;
          :hasCustomer    ?id1 ;
          :contractValue ?contract_value ;
          :hasContractor ?id3 .

  ?id3    a               :Hospital;
          :name           ?organizationName .

  FILTER (?CI = '229732380') .
}
```

APPENDIX E – ONTOP SQL Queries

E.1 Generated by ONTOP

Listing E.1 – Ontop Query 1.

```sql
select count(1)
from(
    SELECT  v5.credit_rating2m11 AS credit_rating2m11
    ,       v5.organization_id1m1 AS organization_id1m1
    ,       v5.person_id1m2 AS person_id1m2
    ,       v5.v0 AS v0
    FROM    (
                SELECT  v1.credit_rating AS credit_rating2m11
                ,       v1.organization_id AS organization_id1m1
                ,       NULL AS person_id1m2
                ,       0 AS v0
                FROM    organization v1
                WHERE(  v1.credit_rating          IS NOT NULL
                AND     (v1.is_corporate_customer = true))

                UNION ALL

                SELECT  v3.credit_rating AS credit_rating2m11
                ,       NULL AS organization_id1m1
                ,       v3.person_id AS person_id1m2
                ,       1 AS v0
                FROM    person v3
                WHERE   (v3.credit_rating         IS NOT NULL
                AND     (v3.is_personal_customer = true)
                AND     'ADULT'                     = v3.life_phase_enum)
            ) v5
) as temp
;
```

Listing E.2 – Ontop Query 2.

```sql
select  count(1)
from(
    SELECT  v2.grade AS grade1m9
    ,       v3.name AS name2m4
    ,       v1.name AS name2m8
    ,       v3.playground_size AS playground_size1m17
    FROM    person v1
    ,       enrollment v2
    ,       organization v3
    WHERE(  v3.playground_size    IS NOT NULL
    AND     v1.person_id          = v2.person_id
    AND     v2.organization_id    = v3.organization_id
    AND     'PRIMARYSCHOOL'         = v3.organization_type_enum)
) as temp
;
```

Listing E.3 – Ontop Query 3.

```sql
select  count(1)
```

```
from(
SELECT  v9.contract_value1m18 AS contract_value1m18
,       v9.name2m15 AS name2m15
,       v9.name2m4 AS name2m4
,        v9.name2m8 AS name2m8
FROM    (SELECT DISTINCT v7.contract_value AS contract_value1m18
        ,       v5.employment_id AS employment_id1m5
        ,       v3.name AS name2m15
        ,       v6.name AS name2m4
        ,       v4.name AS name2m8
        ,       v5.organization_id AS organization_id1m6
        ,       v2.person_id AS person_id1m30
        ,       v1.person_id AS person_id1m57
        ,       v7.supply_contract_id AS supply_contract_id1m1
        FROM    nationality v1
        ,       nationality v2
        ,       person v3
        ,       person v4
        ,       employment v5
        ,       organization v6
        ,       supply_contract v7
        WHERE ( (v3.is_personal_customer  = true)
        AND     v2.person_id              = v3.person_id
        AND     v1.person_id              = v4.person_id
        AND     v1.person_id              = v5.person_id
        AND     v5.organization_id        = v6.organization_id
        AND     v5.organization_id        = v7.organization_id
        AND     v2.person_id              = v7.person_id
        AND     'BRAZILIANCITIZEN'          = v1.nationality_enum
        AND     'ITALIANCITIZEN'            = v2.nationality_enum
        AND     'ADULT'                     = v3.life_phase_enum
        AND     'HOSPITAL'                  = v6.organization_type_enum)
) v9
) as temp
;
```

Listing E.4 – Ontop Query 4.

```
select count(1)
from(
    SELECT  v1.capacity AS capacity1m2
    ,       v1.credit_limit AS credit_limit1m14
    ,       v1.credit_rating AS credit_rating2m11
    ,       v1.name AS name2m8
    ,       v1.playground_size AS playground_size1m1
    ,      v1.organization_type_enum AS v0
    FROM  organization v1
    WHERE(  v1.credit_rating          IS NOT NULL
    AND     (v1.is_corporate_customer = true)
    AND     v1.credit_limit           IS NOT NULL)
) as temp
;
```

Listing E.5 – Ontop Query 5.

```
select count(1)
from(
    SELECT  v6.contract_value1m18 AS contract_value1m18
    ,       v6.name2m6 AS name2m6
    ,       v6.name2m8 AS name2m8
    FROM(   SELECT DISTINCT v3.contract_value AS contract_value1m18
            ,       v4.name AS name2m6
            ,       v1.name AS name2m8
            ,       v3.organization_id AS organization_id1m1
            ,       v1.person_id AS person_id1m20
            ,       v3.supply_contract_id AS supply_contract_id0m2
            FROM    person v1
            ,       nationality v2
            ,       supply_contract v3
            ,       organization v4
            WHERE(  (v1.is_personal_customer      = true)
            AND     v1.person_id                  = v2.person_id
            AND     v1.person_id                  = v3.person_id
            AND     v3.organization_id            = v4.organization_id
            AND     '433019254'                     = v1.ci
            AND     'ADULT'                         = v1.life_phase_enum
            AND     'ITALIANCITIZEN'                = v2.nationality_enum
            AND     'HOSPITAL'                      = v4.organization_type_enum)
    ) v6
) as temp
;
```

E.2 ONTOP Queries Optimized

Listing E.6 – Ontop Query 1 Optimized.

```
select count(1)
from(
    SELECT  v5.credit_rating2m11 AS credit_rating2m11
    ,       v5.organization_id1m1 AS organization_id1m1
    ,       v5.person_id1m2 AS person_id1m2
    ,       v5.v0 AS v0
    FROM    (
                SELECT  v1.credit_rating AS credit_rating2m11
                ,       v1.organization_id AS organization_id1m1
                ,       NULL AS person_id1m2
                ,       0 AS v0
                FROM    organization v1
                WHERE(  (v1.is_corporate_customer = true))

                UNION ALL

                SELECT  v3.credit_rating AS credit_rating2m11
                ,       NULL AS organization_id1m1
                ,       v3.person_id AS person_id1m2
                ,       1 AS v0
                FROM    person v3
                WHERE   ((v3.is_personal_customer = true) )
            ) v5
) as temp
;
```

Listing E.7 – Ontop Query 2 Optimized.

```
select  count(1)
from(
    SELECT  v2.grade AS grade1m9
    ,       v3.name AS name2m4
    ,       v1.name AS name2m8
    ,       v3.playground_size AS playground_size1m17
    FROM    person v1
    ,       enrollment v2
    ,       organization v3
    WHERE(  v1.person_id          = v2.person_id
    AND     v2.organization_id    = v3.organization_id)
) as temp
;
```

Listing E.8 – Ontop Query 3 Optimized.

```
select  count(1)
from(
SELECT  v9.contract_value1m18 AS contract_value1m18
,       v9.name2m15 AS name2m15
,       v9.name2m4 AS name2m4
,        v9.name2m8 AS name2m8
FROM    (SELECT DISTINCT v7.contract_value AS contract_value1m18
        ,       v5.employment_id AS employment_id1m5
        ,       v3.name AS name2m15
        ,       v6.name AS name2m4
        ,       v4.name AS name2m8
        ,       v5.organization_id AS organization_id1m6
        ,       v2.person_id AS person_id1m30
        ,       v1.person_id AS person_id1m57
        ,       v7.supply_contract_id AS supply_contract_id1m1
        FROM    nationality v1
        ,       nationality v2
        ,       person v3
        ,       person v4
        ,       employment v5
        ,       organization v6
        ,       supply_contract v7
        WHERE ( v2.person_id              = v3.person_id
        AND     v1.person_id              = v4.person_id
        AND     v1.person_id              = v5.person_id
        AND     v5.organization_id        = v6.organization_id
        AND     v5.organization_id        = v7.organization_id
        AND     v2.person_id              = v7.person_id
        AND     'BRAZILIANCITIZEN'          = v1.nationality_enum
        AND     'ITALIANCITIZEN'            = v2.nationality_enum
        AND     'HOSPITAL'                  = v6.organization_type_enum)
) v9
) as temp
;
```

Listing E.9 – Ontop Query 4 Optimizd.

```
select count(1)
from(
    SELECT  v1.capacity AS capacity1m2
    ,       v1.address AS address1m3
    ,       v1.credit_limit AS credit_limit1m14
    ,       v1.credit_rating AS credit_rating2m11
    ,       v1.name AS name2m8
    ,       v1.playground_size AS playground_size1m1
    FROM  organization v1
    WHERE(  (v1.is_corporate_customer = true) )
) as temp
;
```

Listing E.10 – Ontop Query 5 Optimized.

```
select count(1)
from(
    SELECT  v6.contract_value1m18 AS contract_value1m18
    ,       v6.name2m6 AS name2m6
    ,       v6.name2m8 AS name2m8
    FROM(   SELECT DISTINCT v3.contract_value AS contract_value1m18
            ,       v4.name AS name2m6
            ,       v1.name AS name2m8
            ,       v3.organization_id AS organization_id1m1
            ,       v1.person_id AS person_id1m20
            ,       v3.supply_contract_id AS supply_contract_id0m2
            FROM    person v1
            ,       nationality v2
            ,       supply_contract v3
            ,       organization v4
            WHERE(  v1.person_id                  = v2.person_id
            AND     v1.person_id                  = v3.person_id
            AND     v3.organization_id            = v4.organization_id
            AND     '274124858'                     = v1.ci
            AND     'ITALIANCITIZEN'                = v2.nationality_enum
            AND     'HOSPITAL'                      = v4.organization_type_enum)
    ) v6
) as temp
;
```

APPENDIX F – OBDA File Generated for the Relational Schema through One Table per Kind Strategy

```
[PrefixDeclaration]
:       https://example.com#
gufo:   http://purl.org/nemo/gufo#
rdf:    http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:   http://www.w3.org/2000/01/rdf-schema#
owl:    http://www.w3.org/2002/07/owl#
xsd:    http://www.w3.org/2001/XMLSchema#

[MappingDeclaration] @collection [[
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-NamedEntity40
target       :RunExample/organization/{organization_id} a :NamedEntity ; :name {
    name}^^xsd:string .
source       SELECT organization.organization_id, organization.name
             FROM organization

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

mappingId    RunExample-Organization
target       :RunExample/organization/{organization_id} a :Organization ; :
    address {address}^^xsd:string .
source       SELECT organization.organization_id, organization.address
             FROM organization

mappingId    RunExample-BrazilianCitizen
target       :RunExample/person/{person_id} a :BrazilianCitizen ; :RG {rg}^^xsd:
    string .
source       SELECT person.person_id, person.rg
             FROM person
             INNER JOIN nationality
                     ON person.person_id = nationality.person_id
                     AND nationality.nationality_enum = 'BRAZILIANCITIZEN'

mappingId    RunExample-ItalianCitizen
target       :RunExample/person/{person_id} a :ItalianCitizen ; :CI {ci}^^xsd:
    string .
```

```
source        SELECT person.person_id, person.ci
              FROM person
              INNER JOIN nationality
                      ON person.person_id = nationality.person_id
                      AND nationality.nationality_enum = 'ITALIANCITIZEN'

mappingId     RunExample-Child
target        :RunExample/person/{person_id} a :Child .
source        SELECT person.person_id
              FROM person
              WHERE life_phase_enum = 'CHILD'

mappingId     RunExample-Adult
target        :RunExample/person/{person_id} a :Adult .
source        SELECT person.person_id
              FROM person
              WHERE life_phase_enum = 'ADULT'

mappingId     RunExample-PersonalCustomer
target        :RunExample/person/{person_id} a :PersonalCustomer ; :creditCard {
    credit_card}^^xsd:string .
source        SELECT person.person_id, person.credit_card
              FROM person
              WHERE is_personal_customer = TRUE
              AND   life_phase_enum = 'ADULT'

mappingId     RunExample-CorporateCustomer
target        :RunExample/organization/{organization_id} a :CorporateCustomer ; :
    creditLimit {credit_limit}^^xsd:decimal .
source        SELECT organization.organization_id, organization.credit_limit
              FROM organization
              WHERE is_corporate_customer = TRUE

mappingId     RunExample-Employee
target        :RunExample/person/{person_id} a :Employee .
source        SELECT person.person_id
              FROM person
              WHERE is_employee = TRUE
              AND   life_phase_enum = 'ADULT'

mappingId     RunExample-Employment
target        :RunExample/employment/{employment_id} a :Employment ; :salary {
    salary}^^xsd:decimal ; :hasOrganization :RunExample/organization/{
    organization_id}  ; :hasEmployee :RunExample/person/{person_id}  .
source        SELECT employment.employment_id, employment.salary, employment.
    organization_id, employment.person_id
              FROM employment

mappingId     RunExample-Customer
target        :RunExample/organization/{organization_id} a :Customer ; :
    creditRating {credit_rating}^^xsd:decimal .
source        SELECT organization.organization_id, organization.credit_rating
              FROM organization
              WHERE is_corporate_customer = TRUE

mappingId     RunExample-Customer41
target        :RunExample/person/{person_id} a :Customer ; :creditRating {
    credit_rating}^^xsd:decimal .
source       SELECT person.person_id, person.credit_rating
             FROM person
             WHERE is_personal_customer = TRUE
             AND   life_phase_enum = 'ADULT'

mappingId    RunExample-SupplyContract
target       :RunExample/supply_contract/{supply_contract_id} a :SupplyContract
    ; :contractValue {contract_value}^^xsd:decimal ; :hasCustomer :RunExample/
    organization/{organization_customer_id}  ; :hasCustomer :RunExample/person/{
    person_id}  ; :hasContractor :RunExample/organization/{organization_id}  .
source       SELECT supply_contract.supply_contract_id, supply_contract.
    contract_value, supply_contract.organization_customer_id, supply_contract.
    person_id, supply_contract.organization_id
             FROM supply_contract

mappingId    RunExample-PrimarySchool
target       :RunExample/organization/{organization_id} a :PrimarySchool ; :
    playgroundSize {playground_size}^^xsd:int .
source       SELECT organization.organization_id, organization.playground_size
             FROM organization
             WHERE organization_type_enum = 'PRIMARYSCHOOL'

mappingId    RunExample-Hospital
target       :RunExample/organization/{organization_id} a :Hospital ; :capacity
    {capacity}^^xsd:int .
source       SELECT organization.organization_id, organization.capacity
             FROM organization
             WHERE organization_type_enum = 'HOSPITAL'

mappingId    RunExample-Enrollment
target       :RunExample/enrollment/{enrollment_id} a :Enrollment ; :grade {
    grade}^^xsd:int ; :hasChild :RunExample/person/{person_id}  ; :
    hasPrimarySchool :RunExample/organization/{organization_id}  .
source       SELECT enrollment.enrollment_id, enrollment.grade, enrollment.
    person_id, enrollment.organization_id
             FROM enrollment

mappingId    RunExample-Contractor
target       :RunExample/organization/{organization_id} a :Contractor .
source       SELECT organization.organization_id
             FROM organization
             WHERE is_contractor = TRUE
]]
```

APPENDIX G – Survey Form

This appendix shows the questionnaire for the experiment reported in Chapter 4.

Note that, if in *Question 2* the participant marks the option “I have no knowledge of databases”, the respondent is thanked and the questionnaire no longer proceeds. The order of *Questions 4* and *5* is randomized by Google Forms.

G.1 Participant’s Profile

Question 1: What is your present occupation:

( ) Student.

( ) IT Professional.

( ) Professor/Researcher.

( ) Other:

Question 2: Please qualify your database experience (concerning relational modeling and SQL queries)?

( ) Only academic (I have taken one or more database courses).

( ) Professional (I work or have worked with databases).

( ) I have no knowledge of databases.

Question 3: How many years of database experience do you have?

( ) Less than 1 year.

( ) Between 1 and 3 years.

( ) Between 3 and 5 years.

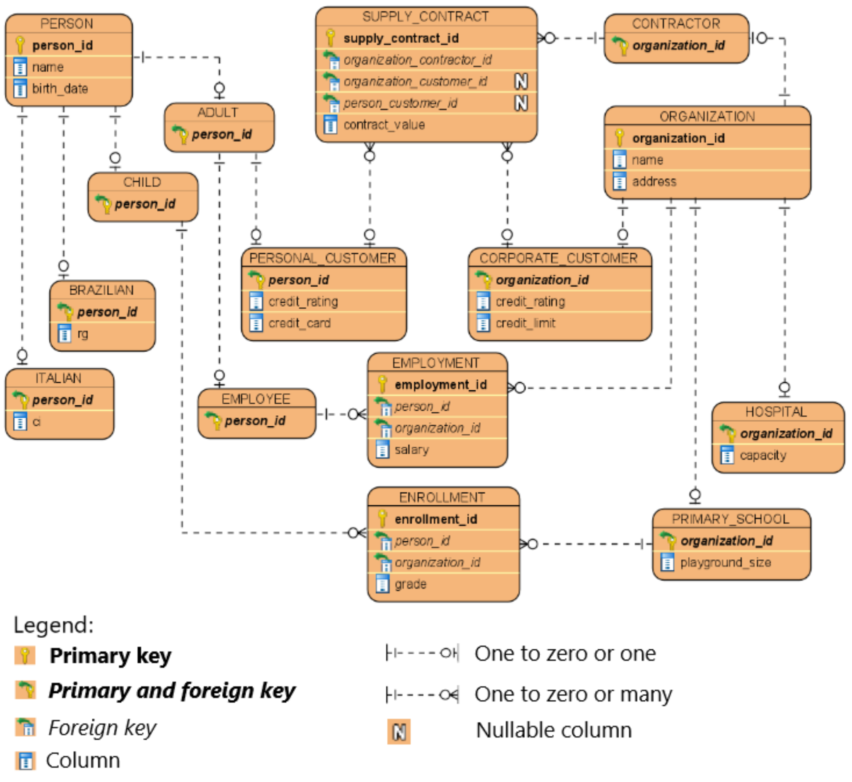
( ) More than 5 years.

G.2 Query Interpretation

Fill in your answers *strictly* in the order in which the questions appear below. The order is very important to us.

Question 4: Describe in free text the purpose of the query below (this query assumes the schema with 5 tables immediately below).

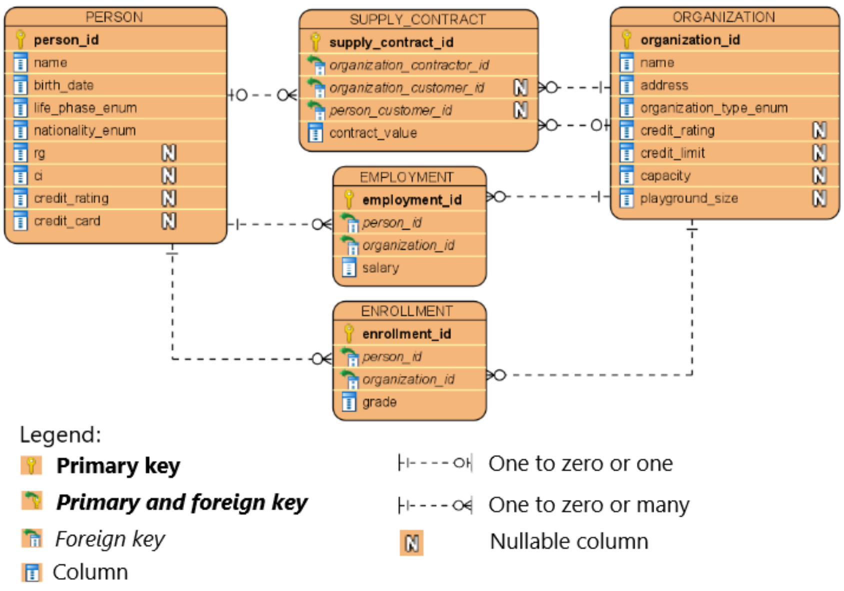
```
SELECT *
FROM person p
JOIN child c
     ON p.person_id                      = c.person_id
JOIN enrollment e
     ON c.person_id                      = e.person_id
JOIN primary_school ps
     ON e.organization_id                = ps.organization_id
JOIN organization o
     ON ps.organization_id               = o.organization_id
JOIN corporate_customer cc
     ON o.organization_id                = cc.organization_id
JOIN supply_contract sc
     ON cc.organization_id               = sc.organization_customer_id
JOIN contractor cont
     ON sc.organization_contractor_id  = cont.organization_id
JOIN organization cont_org
     ON cont.organization_id             = cont_org.organization_id
JOIN hospital h
     ON  cont_org.organization_id        = h.organization_id
```



Answer:

Question 5: Describe in free text the purpose of the query below (this query assumes the schema with 15 tables immediately below).

```
SELECT   *
FROM person p
JOIN enrollment e
     ON  p.person_id                       = e.person_id
JOIN organization o
     ON  e.organization_id                 = o.organization_id
     AND o.organization_type_enum          = 'PRIMARYSCHOOL'
JOIN supply_contract  sc
     ON  o.organization_id                 = sc.organization_customer_id
JOIN organization  cont_org
     ON  sc.organization_contractor_id    =  cont_org.organization_id
     AND cont_org.organization_type_enum  = 'HOSPITAL'
WHERE p.life_phase_enum                    = 'CHILD'
```



Answer:

G.3 Response Order Control

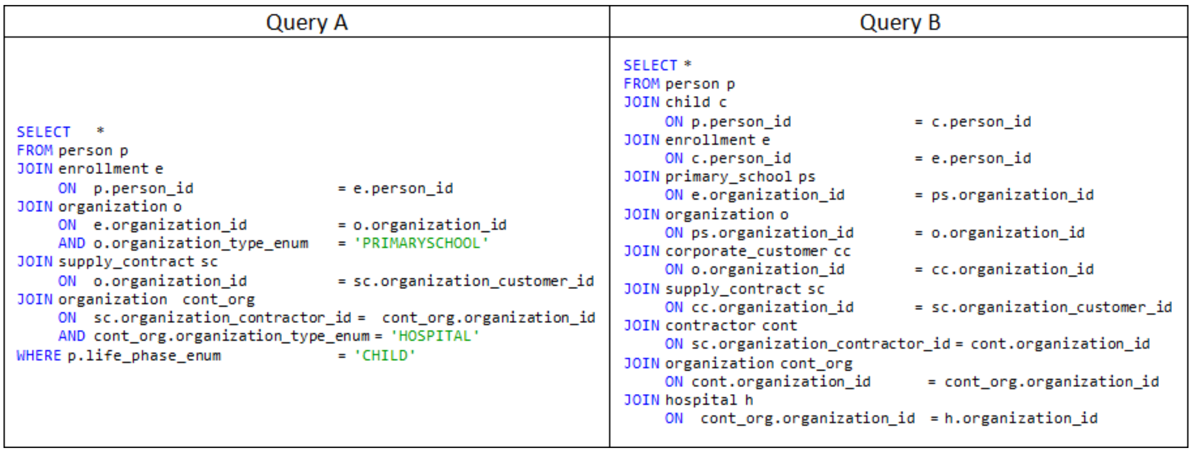
Question 6: On the previous screen, the questions were presented in random order by the system. What query was presented to you at the top of the page?

( ) Query on the schema with 5 tables.

( ) Query on the schema with 15 tables.

G.4 Empirical Study for the Evaluation of Relational Schemas

Question 7: Which of the two queries presented to you before was easier to understand?



Query A

```sql
SELECT   *
FROM person p
JOIN enrollment e
     ON  p.person_id                    = e.person_id
JOIN organization o
     ON  e.organization_id              = o.organization_id
     AND o.organization_type_enum       = 'PRIMARYSCHOOL'
JOIN supply_contract sc
     ON  o.organization_id              = sc.organization_customer_id
JOIN organization  cont_org
     ON  sc.organization_contractor_id =  cont_org.organization_id
     AND cont_org.organization_type_enum = 'HOSPITAL'
WHERE p.life_phase_enum                 = 'CHILD'
```

Query B

```sql
SELECT *
FROM person p
JOIN child c
     ON p.person_id                 = c.person_id
JOIN enrollment e
     ON c.person_id                 = e.person_id
JOIN primary_school ps
     ON e.organization_id           = ps.organization_id
JOIN organization o
     ON ps.organization_id          = o.organization_id
JOIN corporate_customer cc
     ON o.organization_id           = cc.organization_id
JOIN supply_contract sc
     ON cc.organization_id          = sc.organization_customer_id
JOIN contractor cont
     ON sc.organization_contractor_id = cont.organization_id
JOIN organization cont_org
     ON cont.organization_id          = cont_org.organization_id
JOIN hospital h
     ON  cont_org.organization_id   = h.organization_id
```

( ) Query A (on the schema with 5 tables).

( ) Query B (on the schema with 15 tables).

( ) Both had the same level of difficulty.

G.5 Feedback

Register here, any considerations, criticisms, or other remarks that you may have on the schemas and queries in this experiment.

Answer: