DOCTOR OF SCIENCES

Design, Manipulation and Evolution of Hybrid Polystores

Gobert, Maxime

Award date:
2023

Awarding institution:
University of Namur

Link to publication

Design, Manipulation and Evolution of Hybrid Polystores

Maxime Gobert

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Supervised by Prof. Anthony Cleve

University of Namur

Namur Digital Institute

PReCISE Research Center

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

ABSTRACT

Today, data is increasingly important in the software of a wide variety of companies, so that the requirements in terms of volume, performance or storage have changed. As a result, the traditional relational model and the long-established database engineering processes are no longer sufficient. Indeed, other models, called NoSQL, have been developed to meet these new needs. These models, far from totally replacing the existing one, are on the contrary destined to co-exist in software ecosystems. These systems with multiple databases are called **polystores**.

Because of this co-existence of models, the tasks considered complex in database engineering will be all the more complex.

First, **modelling**, NoSQL systems allow a great variety of data representation relying on several data models. No current modelling language can unify these models while preserving the flexibility of representation of specific models.

Secondly, the **manipulation**, the multiplication of databases implies, to query them, to know their own query language. Moreover, this requires the writing of complex join code in case of overlapping or duplication on distinct heterogeneous databases.

And finally, the **evolution**, more models and more languages of manipulations are as many additional elements to be evolved in order to keep a functional system.

In this thesis we try to bring solutions to facilitate the management of these three challenges by proposing a new unified modelling language as well as a conceptual data access code generator facilitating the manipulation and the evolution of **hybrid polystores**.

Keywords: database engineering, nosql, polystores, modelling, code generator

RÉSUMÉ

Aujourd'hui les données ont une importance grandissante dans les logiciels d'une grande diversité d'entreprises, de sorte que les besoins en volumes, performances ou stockages ont évolués. Si bien que le traditionnel modèle relationnel et les processus longuement établis d'ingénierie des bases de données ne sont plus suffisants. En effet d'autres modèles, dit NoSQL, ont été développés afin de répondre à ces besoins nouveaux. Ces modèles, loin de remplacer totalement celui existant, sont à l'inverse voués à co-exister dans les écosystèmes logiciels. Ces systèmes à bases de données multiples sont appelés **polystores**.

Du fait de cette coexistence de modèles, les tâches réputées complexes de l'ingénierie de bases de données seront d'autant plus complexes.

Premièrement, la **modélisation**, les systèmes NoSQL permettent une grande variété de représentation des données reposant sur plusieurs modèles de données différents. Aucun langage de modélisation actuel ne permet d'unifier ces modèles tout en préservant la flexibilité de représentation des modèles spécifiques.

Deuxièmement, la **manipulation**, la multiplication des bases de données implique, pour les interroger, de connaitre leurs propres langages de requêtes. De plus, cela nécessite l'écriture de code complexe de jointure en cas de recouvrement ou de duplication sur des bases de données hétérogènes distinctes.

Et enfin, l'**évolution**, plus de modèles et plus de langages de manipulations sont d'autant d'éléments supplémentaires à faire évoluer afin de garder un système fonctionnel.

Dans cette thèse nous tentons d'apporter des solutions permettant de faciliter la gestion de ces trois challenges en proposant un nouveau langage de modélisation unifié ainsi qu'un générateur de code d'accès conceptuel aux données facilitant la manipulation et l'évolution des **polystores hybrides**.

ACKNOWLEDGEMENTS

This manuscript concludes a research process started in February 2016. This journey has taken place over several acts, each with its own actors and specific roles in the plot. One actor is constant throughout the play and deserves the award for best actor, or rather best director, it is my promoter, Anthony. I would like to thank Anthony for having, since my last year of master, given me his confidence to be up to the task in several of his research projects. Throughout these years, he inspired me, guided me and motivated me thanks to the famous "coup de fouet" sessions. Thanks also for his communicative pride regarding the accomplished work. And finally, thank you for being a great companion during our many trips, bars and restaurants where we met so many "amazing facilities".

The first act of this research journey started with the work done for the Sign Language/French dictionary. An exciting project with a concrete impact to which I am proud of having contributed. In addition to Anthony, I would like to thank Laurence Meurant for including me in this adventure as well as their enthusiasm during the progress presentations.

The second act brought more doubts and self-questioning, I therefore thank here particularly my partner Charlotte, who brings me the confidence, the support, the stability, as well as the fun necessary to a fulfilled personal life. And who along the way, brought us two beautiful children, Léonard and Sacha.

And finally the final act was the concretization of the reflections and work done so far. Here I would like to thank Loup who played the role of the needed combustible in the fire of research which was brewing. Thanks for the long brainstorming and co-coding sessions, sometimes accompanied by the sweet music of a screaming Léonard.

Thanks to colleagues and friends for the good time spent, in afterworks or during the various university events that punctuate the year. Thanks also to the members of my accompanying committee/jury, Prof. Jean-Noël Colin, Prof. Davide Di Ruscio, Prof. Vincent Englebert, Dr. Csaba Nagy and Prof. Wim Vanhoof, for their attention, their questions, their recommendations and their compliments during the different stages of this thesis. Among them, I would like to thank in particular Csaba for the work done together and the beer times that marked our reunions.

This PhD thesis was supported by the F.R.S.-FNRS and FWO via the EOS project 30446992 SECO-ASSIST.

CONTENTS

LIST OF FIGURES

List of Tables

Introduction

INTRODUCTION

Context

Data intensive systems

More and more businesses rely heavily on data, the crucial role of data can be of economic, legal or security value. This increase of value does not come alone and is usually associated with an increase of quantity of data to handle. Usually those business have an ecosystem of applications, websites or databases managing the reading, the writing, the update or the deletion of data. Those systems are called data intensive systems.

Databases, data models and hybrid polystores

Relational databases, relying on relational algebra as data model, were de facto and common technologies for persisting data since decades. Trend initiated by large companies such as Google, Facebook, Twitter, *etc.* brought large data volumes to another scale and they eventually developed their own technologies and data models, *e.g.,* Google BigTable [31] and Cassandra by Facebook [85]. Those new databases have different data models than the common relational data model, they fall into the term **NoSQL** (Not Only SQL). They were developed with specific purposes ranging from performances via horizontal instead of vertical scalability, high availability or query focused data representation. However, these data models are not destined to completely replace traditional relational databases. To balance requirements for data consistency and availability, organizations increasingly migrate towards hybrid data persistence architectures comprising both relational and NoSQL databases for managing different subsets of their data. Such heterogeneous systems, called **hybrid polystores** bring new challenges for research and practice.

Database manipulation code and evolution

Database manipulation code is the part of a program that access data, it is usually seen as an outsider in the codebase of an information system. It lies between the programs and the database, so it belongs partially to both, but not entirely to one. It can also involve multiple development teams. For example, in larger systems, a complex database requires a department of DataBase Administrators (DBA) separated from the software engineers who maintain the application code.

Both teams are in charge of maintaining their own side, but they need to share responsibilities as far as program-database communication is concerned.

Shared responsibilities come at a price, and the *dual role* of database manipulation code leads software to particular vulnerability towards evolutions. This leads to software maintenance problems and application decay.

This part of a software program is however crucial in the operation of a data-intensive system and an error in the data reading or insertion code can have important consequences. The emergence of different types of databases and their manipulation languages brings an additional challenge in terms of the difficulty of building and maintaining this particular code.

Problem Statement

Designing, manipulating, and evolving data intensive systems are known as time-consuming, risky, and error-prone tasks. The main challenges related to those three processes originate from the possibly complex interdependencies between the application programs and their underlying databases. This manipulation code being crucial, we first explored how well a system's data manipulation code was being tested. What are the difficulties encountered by developers when writing these tests, and what are the recommendations made by the community?

Then, we consider that these three challenges may potentially become even more complex with the increasing popularity of NoSQL database technologies and the rise of hybrid polystores.

First, **data modelling** in a polystore context is challenging because of the level of maturity and variety of NoSQL data models. Indeed, relational database design is a well-known and defined process, supported by standard methods. In contrast, NoSQL database modelling is not yet as stable and mature as standard relational database design. In particular, NoSQL data representation does not often rely on a unique explicit schema. Even within the same paradigm, translating conceptual schema elements into physical data structures can be done in various different ways, depending on query purposes, performance requirements or technology-specific constraints. Several specific or unifying abstraction design methods and languages exist for NoSQL data modelling, but none of them integrate relational and NoSQL design while allowing developers to specify fine-grained physical data structures.

Second, **data querying** that involves a combination of relational and NoSQL databases, and which can manage different types of structured and textual data is complex and technically challenging. This, because, unlike relational databases which provide support for SQL and standard Application Programming Interface (API) such as JDBC/ODBC, each NoSQL database provides its own proprietary API and query language. As such, developers are required to be familiar with the query language of each database technology used in the polystore and transversal queries may even involve more than one database, thus necessitating in addition to write application glue code for data reconciliation.

Finally, **evolving** data-intensive systems relying on a single database is already a difficult task, as several software artefacts (schemas, data and programs) have to be evolved and kept consistent with each other. We argue that this co-evolution process is even more challenging in the context of hybrid polystores, due to the multiplicity of data models and the presence of possibly overlapping and interdependent data across multiple databases.

Goals

This dissertation focuses on data intensive systems and particularly the database access code in the context of polystore systems. Its main goals are to

(i) Explore current problems and solutions about tests of database access code.
(ii) Provide automated support when designing, manipulating and evolving hybrid polystores, and therefore reducing the work of developers.

Through those objectives we can formulate the thesis as follows :

A global conceptual approach to specify polystores and generate manipulation code helps developers reduce time and effort to design, manipulate and evolve multi database systems.

Research Questions

From this statement we derived several research questions:

- **RQ 1** What are the problems developers face when testing database manipulation code and what solutions are recommended by the community?
- **RQ 2** How to build a conceptual and physical modelling language able to keep specific data model constructions?
- **RQ 3** To what extent can we use this model to generate data manipulation code at the conceptual level independent of any physical storage data representation?
- **RQ 4** How can we support the databases' evolution of a polystore with regard to their impacted artefacts?
- **RQ 5** What is the performance, usability and usefulness of the developed solution?

Contributions

- A taxonomy of problems faced by developers when testing database access code (*RQ 1*).
- A taxonomy of developers solutions regarding best practices about tests in database access code (*RQ 1*).
- A language to design polystores conceptually and describe data model specific structures physically (*RQ 2*).
- A tool generating code to access data at the conceptual level based on designed polystore model language(*RQ 3*).
- A generated conceptual API Code resistant to physical structure evolution (*RQ 4*).

- A query adaptation tool to change existing polystore queries according to given schema evolutions (*RQ 4*).
- A theoretical framework to evolve polystores (*RQ 4*).
- A quantitative and qualitative evaluation of the framework (*RQ 5*).

Publications

(i) Gobert, M. (2020). Schema evolution in hybrid databases systems. In Proceedings of the 46th International Conference on Very Large Data Bases (VLDB 2020): PhD workshop track. ACM Press. [52]

(ii) Fink, J., Gobert, M., & Cleve, A. (2020, September). Adapting queries to database schema changes in hybrid polystores. In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM 2020) (pp. 127-131). IEEE. **Best Paper Award** [48]

(iii) Gobert, M., Nagy, C., Rocha, H., Demeyer, S., & Cleve, A. (2021, June). Challenges and Perils of Testing Database Manipulation Code. In International Conference on Advanced Information Systems Engineering (pp. 229-245). Springer, Cham. (CAiSE 2021) [56]

(iv) Gobert, M., Meurice, L., & Cleve, A. (2021, October). Conceptual Modeling of Hybrid Polystores. In International Conference on Conceptual Modeling (pp. 113-122). Springer, Cham. (ER 2021) [54]

(v) Gobert, M., Meurice, L., & Cleve, A. (2022). HyDRa: A Framework for Modeling, Manipulating and Evolving Hybrid Polystores. In Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022). [55]

(vi) Gobert, M., Nagy, C., Rocha, H., Demeyer, S., & Cleve, A. (2023) Best Practices of Testing Database Manipulation Code. In Information Systems, Volume 111 (InfoSys) [58]

(vii) Gobert, M., Meurice, L., & Cleve, A. (2023 - Accepted). Modeling, Manipulating and Evolving Hybrid Polystores with HyDRa. In Original Software Publications, Science of Computer Programming.

Structure of the thesis

Chapter 1 gives the basic required definitions that are at the roots of this thesis.
Chapter 2 explores the state of the art of the research areas addressed. These works are then summarized and the gaps in relation to our research questions are highlighted.
Chapter 3 presents an empirical study of the problems and recommendations of best practices regarding the testing of database access code. This study exploited questions and answers found on several developer community websites.
Chapter 4 details the modelling language developed to represent a hybrid polystore system.
Chapter 5 specifies the data access library automatically generated on the basis of a schema expressed in the developed language.

Chapter 6 presents the theoretical framework and the query adaptation tool developed to facilitate the evolution of polystores.
Chapter 7 exposes the experiment carried out on the use of the modelling language and the generated access library.
Chapter 8 describes how the correctness and the performances of the library were established via the application of systematic tests.
Chapter 9 lists others use cases of the proposed modelling or manipulation solutions.
Chapter 10 concludes this thesis by reviewing the research questions, the contributions made and by identifying paths for future research.

Part I

Background

CHAPTER 1

CONCEPTUAL BACKGROUND

Contents

This chapter presents the important definitions and concepts that constitutes the basis of this thesis. We present the process of database design, the emergence of new data models called NoSQL, the role and importance of database manipulation code and last we introduce what it means to evolve a database.

1.1 Database Design

Database design is the process of modelling and implementing a database that meets specific user requirements. Databases relying on the relational model were the most used and de facto standard when using databases for a long period. However, the emergence of big data brought new challenges and exposed the limitation of the relational model in this context. New data models (NoSQL) were developed that may have an impact on how to design a database. This section presents the standard database design process and then introduces the most common NoSQL data models.

1.1.1 Standard database design process

The database design process has been described extensively in the literature [24] and has been available for several decades in standard methodologies and Computer-Aided Software Engineering (CASE) tools. As shown in Figure 1.1, database design is typically made up of four main sub processes:

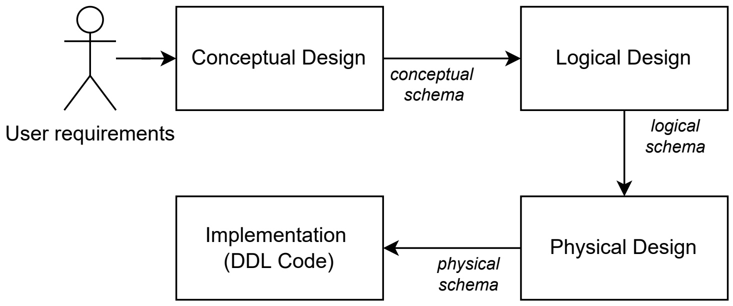


Figure 1.1: Standard database design processes.

(1) **Conceptual design** is intended to translate user requirements into a **conceptual schema** that identifies and describes the domain entities, their properties and their associations in a platform-independent way. This abstract specification of the database collects all the informational structures and constraints of interest.

(2) **Logical design** produces an operational **logical schema** that translates the constructs of the conceptual schema according to a specific technology family, in principle, without loss of semantics.

(3) **Physical design** adds performance-oriented construction and parameters, such as indexes to the logical schema. The output of this process is the **physical schema** of the database.

(4) **Implementation** translates the physical schema (and some other artefacts) into the Data Definition Language (DDL) **code**, compliant with the target database management system. Structural DDL declaration code as well as components such as checks, triggers and stored procedures are written/generated to implement the data structures and constraints of the physical schema.

The database schemas express the structures and constraints of a database. As mentioned above, they are usually classified according to the level of abstraction they belong to: **conceptual**, **logical** and **physical**.

- **Conceptual schemas** A conceptual schema is a **platform-independent** model of the database. It mainly specifies **entity types**, **relationship types** and **attributes**. Entity types represent the main concepts of the application domain. They can be organized into **is-a** hierarchies, organizing supertypes and subtypes. Relationship types represent relationships between entity types. Attributes represent common properties of the entity type instances.
- **Logical schemas** A logical schema is a **paradigm-dependent** data structure definition, that must comply with a given data model. The most commonly

used families of models include the relational model, the XML model, the object-oriented model, the NoSQL models (column-oriented, graph-oriented, document-oriented, key value stores, *etc.*).

- **Physical schemas** A physical schema is a logical schema enriched with all the information needed to implement efficiently the database on top of a given data management system. This includes **platform-dependent** technical specifications such as indexes, physical device and site assignment, page size, file size, buffer management or access right policies.

Figure 1.2 gives an example of conceptual, logical and physical schemas, expressed in the Generic Entity-Relationship (GER) data model [69]. In this example, all the constructs belonging to the conceptual schema have been translated into equivalent constructs in the logical schema of the relational model. In the conceptual schema part we find several important concepts of the considered application domain, represented by the entity types *Customer*, *Order* and *Product*, each of these entity types having its own attributes. This schema also contains two relationship types. Firstly, *places* represents the fact that an instance of *Customer* can place more than one *Order*, expressed by the presence of the cardinality 0-N, conversely an instance of *Order* can only be linked to one instance of *Customer* (due to the cardinality 1-1), this type of relationship is called a **one-to-many** relationship. Secondly, *detail* expresses that an order contains several (0-N) products *Product*, and these can be in several orders as well, this is a **many-to-many** relation. Lastly, since a product is present in an order in a certain quantity, *quantity* is specific to the relation linking two instances of the entity types and is thus a relationship type attribute of *detail*.

The equivalent logical schema is found in the middle of Figure 1.2. The logical schema translates entity types and relationship types into equivalent structures compatible with the chosen data model, in this case the relational model. The entity types become *tables* and the relationship types become reference structures, either (1) **foreign keys** *i.e.,* reference attributes (*e.g., Order* table contains an attribute with the identifier of *Customer*); or (2) an intermediate table containing two references *i.e.,* join structures (*e.g.,* **many-to-many** relationships *detail* becomes the table *detail* with two references, one to the table *Order* and one to the table *product*).

Finally, the physical schema, represented in the lower part of the figure, is similar to the logical schema except for the mention *acc* which indicates the presence of an index on the concerned element. The small difference between these two schemas means that by abuse of language the terms physical schema or logical schema can be interchanged in the remaining chapters of this thesis.

1.1.2 NoSQL data models

The number of NoSQL systems is still growing and they represent now nearly half (171) of the total number of database management systems on the market [1]. But, in contrast with relational database management systems, NoSQL engines do not rely on the same theoretical data model.

[1]https://db-engines.com/en/ranking_categories consulted in June 2022

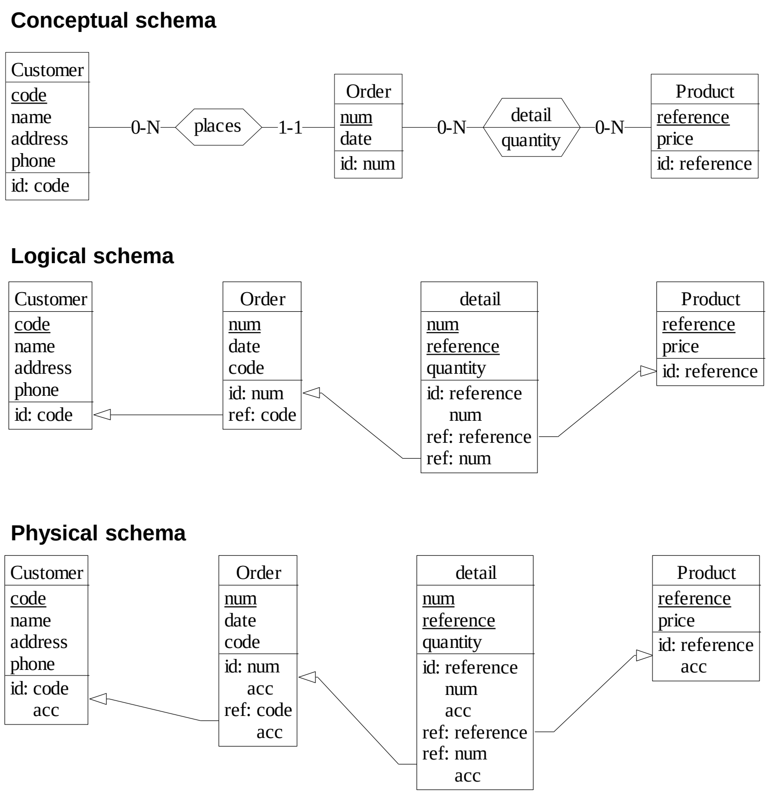


Figure 1.2: Example of conceptual, logical and physical schemas of a relational database.

Those systems can be grouped into four principal different data models, each having its specific requirements and advantages. We further explain and summarize them below based on a survey provided by Hecht and Jablonski [72].

1.1.3 Key value stores

The key value store model is the most "schema-less" model among NoSQL systems. It is indeed based on a simple key-value pair, with no constructs allowing to define explicit relationships between data instances. The values are stored in byte arrays and therefore the only way to retrieve data is by means of the keys. They are useful for very simple operations and basic application usage as they only provide *put, get* and *delete* operations. Other operations or queries have to be managed in the application code. This apparent simplicity may lead to possibly complex schema design problems when deciding how to organize the data. One needs to carefully design and manage the chosen key patterns. Design methods [19,106] and best practices [7] identified two main patterns. The first one, called **key value per field**, creates a key-value pair for each atomic field. The key is composed of elements identifying the entity type and the attribute, and another element identifying a particular atomic instance. Examples of key patterns for this design includes **ENTITY:[identifier]:FIELD**. It results in data illustrated in Table 1.1, the table represents data about an entity type named *MOVIE*, and more specifically the attributes *TITLE* and *DURATION* of two movies (*tt00118715* and *tt5126354*). *MOVIE*, *TITLE* and *DURATION* are the common static components of the key. The dynamic component contains the identifier of a movie instance. The combination of the static and dynamic components allows to build a complete key and to retrieve the value of the requested instance attribute. The second design type, **key value per object**, uses complex data types instead of simple atomic value, the value contains now multiple fields. Table 1.2 contains the same semantic data as Table 1.1 but instead of representing it using four key value pairs, we here only use a single pair for a complete *MOVIE* instance. Example platforms based on the key value data model are Redis [7], ArangoDB [15] and Riak [103].

Key	**Value**
MOVIE:tt0118715:TITLE	The Big Lebowski
MOVIE:tt5126354:TITLE	Vanilla Sky
MOVIE:tt0118715:DURATION	102
MOVIE:tt5126354:DURATION	126

Table 1.1: Example of key value data with key value per field design pattern.

Key	**Value**
MOVIE:tt0118715	title : "The Big Lebowski", duration : 102
MOVIE:tt5126354	title: "Vanilla Sky", duration : 126

Table 1.2: Example of key value data using key value per object design pattern.

1.1.4 Document stores

The document store data model has a similar structure as the key value model. Data is stored as key-value pairs, but they are wrapped in a JSON like document. This model offers the flexibility of a schema less data store, but it also helps the developer to query data using a specific query language. Document store model have a set of mapping rules to map a relational model. MongoDB[2], the leading technology for this data model, for example provides a set of example on their website [3].

The flexibility of the document model allows to have multiple different documents in the same collection. Figure 1.3 shows three documents, each of them have a different set of attributes, the first one is composed of *first*, *last* and *country* fields, the second document has *city* instead of *country* and the third one does have neither but have a complex field *phone*. *phone* is an object, which also contains two sub attributes, *home* and *work*. Despite having different fields all those documents can

[2]https://www.mongodb.com/
[3]https://docs.mongodb.com/manual/reference/sql-comparison/

be stored in the same collection. MongoDB [1], Couchbase [36], CouchDB [37] and ArangoDB [15] are examples of document store model database system.

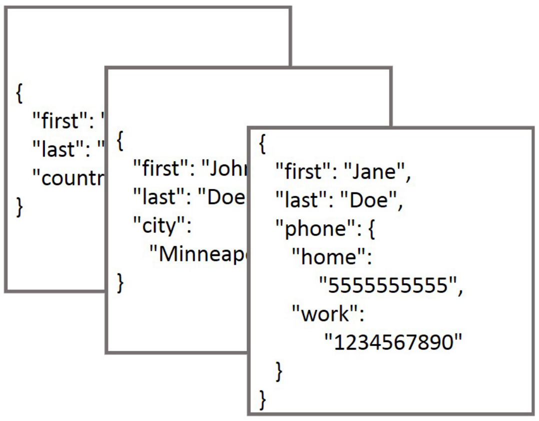


Figure 1.3: Document oriented data model.

According to the design guidelines [5] of MongoDB [1], data expressing a relationship can be stored in different ways, namely **embedding, referencing or denormalization** each having their specific benefits. Those design techniques are explained below.

Embedding Embedding is the ability to store an object in an attribute. This technique is particularly used for relationships of the type **one-to-few**, *i.e.,* where there is a limited number of links between the entities. In Listing 1.1 we have a person containing an array of addresses, that are in their turn objects with their own attributes. The advantages of this design is that users can retrieve client and address information can be retrieved in a single query. The drawback is that addresses do not exist as stand-alone entities and this makes queries on addresses more difficult.

```
  {
  name: 'Kate Monster',
  ssn: '123-456-7890',
  addresses : [
     { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },
     { street: '123 Avenue Q', city: 'New York', cc: 'USA' }
  ]
}
```

Listing 1.1: A document with embedded data expressed in textual format

Child referencing This design is used when you need data to exist as stand-alone entities. This is particularly useful when one wants to use the same object for multiple references. Two collections exist containing the data for each entity types, and the link between the two is expressed via a foreign key attribute (*i.e.,* a field containing an identifier of the other collection). It usually applies to **one-to-many** relationships. Listings 1.2 and 1.3 represent respectively the collection containing *Parts* and *Products*. We notice that a product is composed of parts and this link is expressed using the part object identifier (in the example the *_id* field). To read part objects of a particular product it is now mandatory to use two distinct queries on both collections.

```
{
  _id : ObjectID('AAAA'),
  partno : '123-aff-456',
  name : '#4 grommet',
  qty: 94,
  cost: 0.94,
  price: 3.99
},...
```

Listing 1.2: A collection containing parts documents

```
{
  name : 'left-handed smoke shifter',
  manufacturer : 'Acme Corp',
  catalog_number: 1234,
  parts : [     // array of references to Part documents
      ObjectID('AAAA'),    // reference to the #4 grommet above
      ObjectID('F17C'),    // reference to a different Part
      ObjectID('D2AA'),
      // etc
  ]
},...
```

Listing 1.3: A collection containing products documents with references attributes to parts documents

Parent referencing Parent referencing is a specific design to mitigate the technology constraint that a document must not exceed 16MB. Because of the number of references that may exist, the size of the array may be too big and therefore child referencing is not possible. *e.g.,* if we consider data entities of *Cities* and *Residents*. We cannot embed the inhabitants in a city document, specially for big cities such as New York, as millions of references would have to be in the array. In those situation we use **parent-referencing** where the residents will hold the reference to the city. This design technique is applied to represent so called **one-to-millions** relationships.

Two-way referencing If both advantages of referencing one way or the other are needed one can combine child and parent referencing. Of course, it comes at the price of having to perform multiple update queries when modifying data.

Denormalization Denormalization saves the application the complexity of writing joined queries needed when attributes of referenced data are to be retrieved. Instead, the referencing and referenced data are stored in the same structure. In the *Product* and *Part* example, one need to query the product collection and then perform another query on the part collection to retrieve *Part* data. If only the part attribute *name* is needed, one can add it alongside the id of part in the product document, as depicted in Listing 1.4. This way the product and all its parts names are retrieved in one single query. This design technique is to use in a context of a high read vs writes operations. Indeed, in this situation when an update of a part *name* occurs, both collections have to be updated, and this cannot be done in a single atomic operation.

```
  {
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [ // Part name is denormalized
        { id : ObjectID('AAAA'), name : '#4 grommet' },
        { id: ObjectID('F17C'), name : 'fan blade assembly' },
        { id: ObjectID('D2AA'), name : 'power switch' },
        // etc
    ]
}
```

Listing 1.4: Example of denormalized data

1.1.5 Column family stores

The column-oriented model borrows concepts from the relational and key-value models. Indeed, it is composed of a *table* data structure containing *columns* and *rows*, these rows are moreover identified by an identifier *rowkey* as depicted in Figure 1.4. The columns are represented as key-value pairs, the particular specificity of the column-oriented model is the possibility of grouping several columns under the concept of **column family**, these groupings of columns are common to all the rows, they must be defined during the definition of the table. However, although these families are defined in advance, it is not mandatory for a row to populate each specific column of a column family with a value. This model particularly exploits the denormalization-based design, where other data structures become column families, allowing for join-avoidance queries.

The column families impact the way the data is stored, indeed the data in these databases is physically ordered by *rowkey* and *column family*, which makes range selection queries on a very large volume of data much more efficient.

Systems implementing this data model are HBase [3] or Cassandra [2].

1.1.6 Graph databases

The last important data model category is the graph database. This has been built in order to specifically manage heavily linked data. It can consist of simple triple values like Resource Description Framework (RDF) or more complex structures containing key value pairs. Graph databases can be queried in two different ways, either by trying to find a part of the graph that matches a criterion or by exploring the graph by starting in a specific node. It is also possible to express constraints that limit the type of edges applicable to certain type of nodes. An advantage that no other

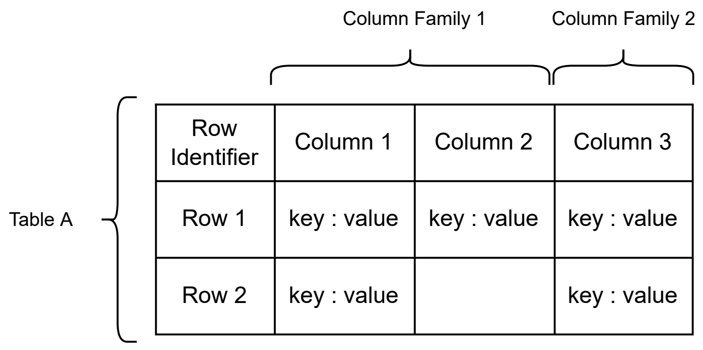


Figure 1.4: Column oriented data model.

data models gives is that it can also be queried with a standard language (SPARQL Protocol and RDF Query Language). Which is common to more than one graph database management system. Neo4j [97], GraphDB [65] and ArangoDB [15] are examples of such databases implementing the graph data model.

1.2 Database Manipulation Code

Choosing the right database and data model for its data is one step of a data intensive system construction. Afterwards one has to manipulate the data inside the client applications. This part of the code is particularly important as it stands between the core application functionalities and the database. A change of business requirements may impact the database manipulation code while an evolution of data model may change the database and therefore the manipulation code as well. In large companies those two responsibilities may be spread between two different teams. This **dual role** of code, Stonebraker *et al.* argue that it is the most significant factor of database or application decay [122].

Data intensive system's applications may be written using different programming languages, and moreover multiple ways to access the data are possible depending on the chosen database. We explain below several of the most common possibilities to access data in applications code. The illustrative examples will consider Java as host programming language.

1.2.1 Native database libraries

Native database libraries are libraries published by official developers of a particular database. They allow users to access data using via specific methods or via methods interpreting native database query language.

JDBC Java DataBase Connectivity (JDBC) is a standard Application Programming Interface (API) able to connect, read, update and delete data of a **relational** database. This standard makes the usage of different database providers, such as MySQL, PostgreSQL, MariaDB interchangeable with regard to the application code. Code presented in Listing 1.5 illustrates how to connect to a relational database and how to execute a SQL selection query statement (line 7). A **ResultSet** object is then retrieved and contains each matching lines which can then be treated as needed. In the example each column of each row are displayed.

```java
1   Connection conn = DriverManager.getConnection(
2       "jdbc:somejdbcvendor:other data needed by some jdbc vendor",
3       "myLogin",
4       "myPassword");
5
6   try (Statement stmt = conn.createStatement()){
7       ResultSet rs = stmt.executeQuery("SELECT * FROM MyTable");
8       while (rs.next()) {
9           int numColumns = rs.getMetaData().getColumnCount();
10          for (int i = 1; i <= numColumns; i++) {
11            System.out.println( "COLUMN " + i + " = " + rs.getObject(i));
12          }
13      }
14    } catch (SQLException e) {
15        logger.warn("Could not close JDBC Connection",e);
16    }
```

Listing 1.5: Example of Java code for relational database using JDBC

Mongo Java Driver The mongo java driver [4] is the official java driver provided by MongoDB to query their document based database. Listing 1.6 illustrates how to connect to a MongoDB database and retrieve the data in *productCollection* with an attribute *price* greater than 100. The resulting document are then displayed using the JSON format.

```java
String uri = "mongodb://user:pass@sample.host:27017/";

try (MongoClient mongoClient = MongoClients.create(uri))
{
    MongoDatabase database = mongoClient.getDatabase("admin");
    MongoCollection collection = mongoClient.getDatabase("myMongoDB").
     getCollection("productCollection");
    Bson filter = Filters.gt("price", 100);
    collection.find(filter).forEach(doc -> System.out.println(doc.toJson()))
     ;
}
```

Listing 1.6: Example of Java code for MongoDB document database

Jedis Jedis is the Java driver to access the key value database Redis. Querying a Redis database is done using commands instead of a query language. The value of a key value pair can be a string value, a map, a list, a set, a sorted set *etc.* There exists specific command for each data type available. In Listing 1.7, line 3 shows how to select a pair containing a string as value. Line 4 exposes how to retrieve all fields of a map value.

[4] https://mongodb.github.io/mongo-java-driver/

```java
Statement stmt = conn.
    createStatement();
ResultSet rs = stmt.executeQuery("
    SELECT id, first_name,
    last_name FROM persons where
    id=10");
String firstName, lastName;
 while (rs.next()) {
    firstName = rs.getString(2);
    lastName = rs.getString(3);
  }
```

Listing 1.9: Java code using JDBC to retrieve attribute data

```java
Person p = Person.getById(10);'
String firstName = p.getFirstName()
    ;
String lastName = p.getLastName();
```

Listing 1.10: Java code using ORM

```java
  Jedis jedis = new Jedis("localhost", 6666);

  String name = jedis.hget("user#1", "name");
  Map<String, String> fields = jedis.hgetAll("user#1");
  fields.forEach((k, v) -> System.out.println((k + ":" + v)));
```

Listing 1.7: Example of Java code for Redis key value database

1.2.2 Object mappers

Object mappers are an intermediary method to query databases. Mappers for relational databases, Object Relational Mapper (ORM) exist since decades and have since been standardised via Java Persistence API (JPA) and Java Persistence Query Language (JPQL).

Instead of manipulating data using native query language or specific drivers methods, mappers allow the developer to manipulate Java objects representing data, see Listing 1.8. A mapping between the object and the effective database structures is declared through annotation or through a configuration file. Finally, read, update find and delete operations are available on those objects. In Listings 1.9 and 1.10 we compare the code to retrieve *lastname* and *firstname* of particular person using native libraries versus using ORM library.

```java
  @Entity
  @Table(name="PERSONTABLE")
  public class Person {
    @Id
    private Long id;
    private String firstName;
    private String lastName;

    // getters and setters
  }
```

Listing 1.8: Class declaration of an Entity

Those technologies also exist for NoSQL technologies, Object NoSQL Mapper (ONM). In contrast to the mapping between relational model and object class diagram which is well studied and benefits from decades of experience, the NoSQL data models mapping to object-oriented class is still an open research area. As we saw in Section 1.1.4, in document database a relationship between two entities can be stored in many physical representations such as embedding, child referencing, parent referencing *etc.* Existing ONMs are generally focused on one single NoSQL data model (*e.g.,* they are called Object Document Mapper (ODM) for document databases) and the possibilities of custom physical data representation are limited. Those are some limitations among others identified in a survey about ONMs done by Störl *et al.* [123].

1.3 Polystores

We briefly described in the introduction the emergence of systems using databases of different types, mixing both the relational model and the NoSQL models, those systems are called **hybrid polystores**. Indeed, each of these data models fulfils specific needs and use cases of a system in terms of data can be oriented towards one model as well as another depending on user requirements. In Benats *et al.* [26] we studied more than 40,000 open source projects using databases, analysed which models were the most popular, for which programming languages, what were the most common combinations in hybrid systems, and finally studied the evolution of these systems (i.e., what type of database was added or removed in these systems). We concluded that the trend was upward for the proportion of systems using more than one type of database (16% of the 40,000 projects in 2020) and highlighted that in single-database systems the proportion of relational databases versus NoSQL models was still larger but slightly decreasing (54.72% vs 45.28% in 2020).

Below we will illustrate an example of a hybrid database system and the different reasons that can push to adopt this kind of architecture. Figure 1.5 shows a conceptual schema of an e-commerce application. This application allows a user to place orders, which they can pay with credit cards. These orders consist of a set of products, for which there are pictures as well as reviews, which can be subject to comments. The reviews and comments are placed by the users.

Figure 1.6 illustrates the databases that can be chosen for such a system. For each of these databases is associated the logical (or physical) schema of the data it contains. Below we describe these and the reasons that led to such choices. In such a system a key value or column oriented database can be a good choice to store product pictures as the volume of data can quickly be large and while maintaining a high availability is a crucial requirement, moreover only direct access query and no query on the value will happen.

Secondly, it is common in an e-commerce application to offer purchase recommendations for similar products, or products that have also been ordered together by other users. This recommendation functionality intensively exploits the links that exist between data and for this kind of task traditional relational databases are not the best choice due to the multiplicity of joins and complex building of such queries. Therefore, databases based on the graph model are a more appropriate choice.

Next, document databases are effective in building unique data structures containing several related concepts, which in a standard relational database would have

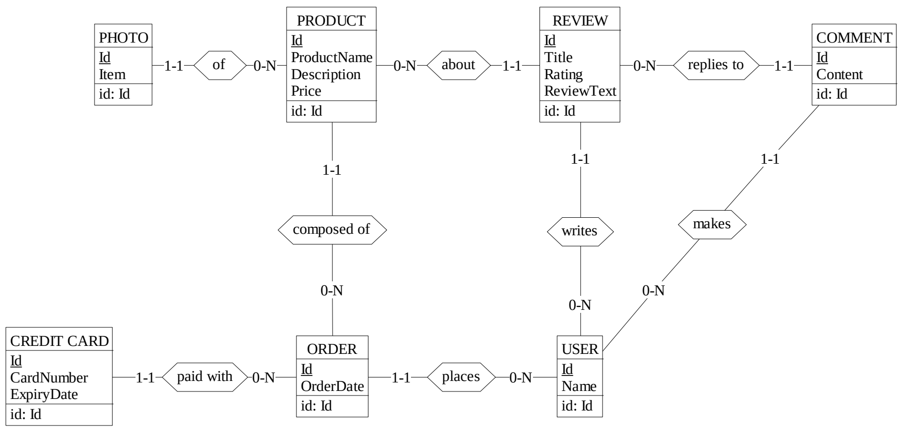


Figure 1.5: Conceptual schema of an e-commerce system

been separated into several tables. In our application domain we see that products, product reviews and comments are natural aggregates, *i.e.,* they are often retrieved together, a page displaying a product will also display reviews and comments. This is why in our e-commerce architecture, these three concepts have been stored in a single nested structure in a document database.

Finally, sensitive information concerning payments, which must respect strong Atomicity, Consistency, Isolation, Durability (ACID) constraints, have their place in relational databases where transactions allow for the protection of data.

Obviously the advantages brought by a hybrid architecture also bring their negative counterpart. Thus, the application code manipulating the data of this system will have to deal with the integration of several libraries and query languages specific to each of the databases used. As we have described in Section 1.2. In the physical schema and the databases described above, we can also see that the information concerning products is distributed over several databases, sometimes with a duplication of data. Managing this duplication and heterogeneity are the main challenges of **hybrid polystores**.

1.4 Database Evolution

Database evolution can be defined as a task where the ultimate goal is to change all impacted artefacts of a data intensive system to be compliant with an evolution concerning the database.

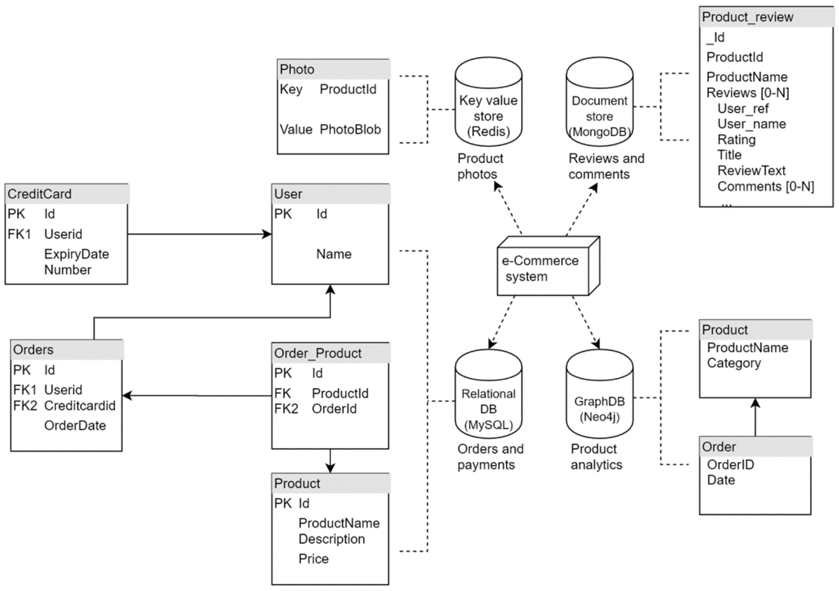


Figure 1.6: Databases and data representation (physical schema) of e-commerce system

Evolution Processes It is a complex process typically composed of a chain of sub processes which aim to :

(i) Understand the database and its artefacts.
(ii) Identify the impacted artefacts of a particular change.
(iii) Applying the evolution.
(iv) Adapt impacted artefacts.

Artefacts The three classical artefacts to evolve in a data-intensive system evolution process are :

- **Schemas**, that concerns the modification of the database structures (conceptual, logical and physical schemas);
- **Data**, which relates to the changes needed on stored data;
- **Client application**, addressing the replacement and adaptation of application code and queries.

These layers may evolve independently or jointly, depending on the specific requirements that drive the evolution. Evolving those layers in synchronization with each other is called **co-evolution**. Co-evolution involves significant work and is a risky and error-prone task. An empirical study by Qiu *et al.* [100] analysed open source projects and discovered that an atomic database schema change (*i.e.,* an attribute renaming) involve a mean of 10 to 100 lines of application code to modify.

Evolution classification Database evolution can be classified according to three dimensions, namely *structural, semantic, platform/language* dimensions [33]. Those dimensions respectively concerns evolutions about the schemas (conceptual, logical or physical modifications), the semantics of the schemas (adding, removing, or preserving the global semantics) or databases and manipulation languages.

The platform or language dimension intends to characterize a database evolution scenario in terms of platform/language change. In the light of polystore systems integrating both relational and NoSQL data models this dimension have an increased importance. The following possible cases can be distinguished:

- **Intra-platform database evolution**: the database evolution does not involve the replacement of the data management system.
- **Inter-platform database evolution**: the database evolution requires the replacement of the data management system with another one. Two sub-cases can occur:
  - **Intra-paradigm platform change**: the source and target database platforms belong to the same paradigm. A typical example is the migration of a relational database from MySQL to PostgreSQL.
  - **Inter-paradigm platform change**: the database evolution relies on a database paradigm switch, *i.e.,* the source and target database platforms belong to different paradigms. This is the case, for instance, when migrating a relational database (*e.g.,* MySQL) to a NoSQL platform (*e.g.,* MongoDB).

Illustrative example Let us consider a polystore representing an e-commerce system with product information stored for different reasons on multiple different data models databases, as described in Section 1.3 An illustrative atomic *structural* evolution at the conceptual schema level may materialize as a requirement to change the *ProductName* attribute of entity type *Product* to *ProductReference*, depicted in Figure 1.7.

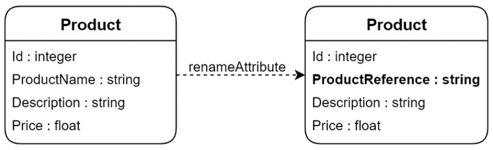


Figure 1.7: Atomic evolution

The *conceptual schema* of Figure 1.5 has to be changed to reflect the new name. Next, Figure 1.8 shows the impact of this change on *logical/physical schemas* artefacts. As this product information is stored in multiple databases, multiple *intra platform database evolution* are to be performed on the logical schema. On the physical level, as a relational table is identified as impacted artefact, an *ALTER TABLE* statement renaming a column name will have to be executed.

As NoSQL databases are schema-less, meaning that only data reflects the schema, *data evolution* has to be performed in order for the data to reflect the evolution scenario considered. Listing 1.11 shows a document that has to be modified accordingly. Lastly program code and queries accessing the old attribute name *ProductName* must also be adapted. Listing 1.12 shows the Java code needing code change to comply to the evolution. In line 2 it is code accessing the document database, in line 4 there is a graph database query and in line 6 there is a relational database query.

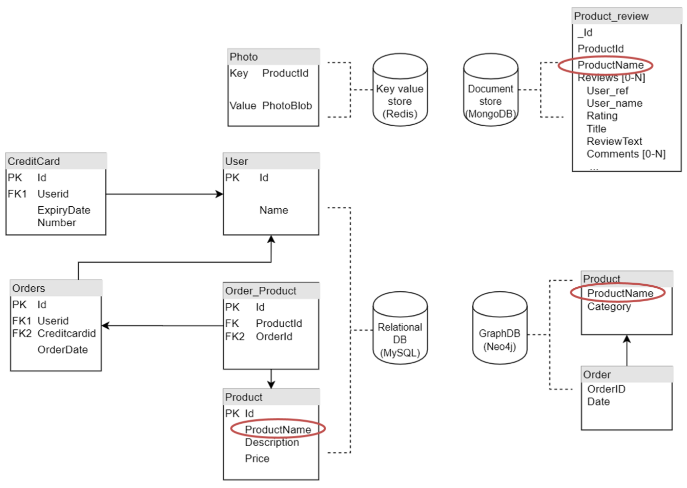


Figure 1.8: Impact of change in physical schema

```
{
  "_id": "5df8b60952264a01b6a8a2fc",
  "productId": "product0",
  "productName": "SuperCleaner3",
  "reviews": [
    {
      "review_id": "review1",
      "user_ref": "user55",
      "user_name": "Serena",
      "rating": 1,
      "title": "Dont buy!!!",
      "reviewText": "I dropped it from a 5 stories building and it broke!",
      "comments": []
    },
    ...
      ]
    }
}
```

Listing 1.11: Impact in data of a document database

```
// Application code
dbcollection.find({productName:"SuperCleaner3"});
// Native graph database query
match (n:Product{productName:"SuperCleaner3"})<--(o:Order)-->(otherProducts:
   Product) return n,o,otherProducts
// Native SQL query
Select userid from Orders O JOIN Order_Product OP ON O.Id=OP.OrderId JOIN
   Product P ON OP.ProductId=P.Id where P.productName="SuperClean3"
```

Listing 1.12: Impact on queries or application code

If those modifications are not done it exposes the system to multiple data inconsistency problems. Data migration and program adaptation may be executed using multiple strategies that will be presented in the following chapter.

1.5 Concluding Remarks

Throughout this chapter we described the well established process of database design in data intensive systems, specially for relational databases. We then described the four main data models of the so called NoSQL databases. We stated the importance of the database access code in a software system and the different choices that are available to the developers. They indeed need to determine the right technology to use and master very different way of accessing the data depending on the underlying databases. Finally we detailed the process of database evolution in data intensive systems. The next chapter will present the state of the art in terms of design, data manipulation and evolution of database-intensive systems.

CHAPTER 2

STATE OF THE ART

Contents

This chapter presents state-of-the-art approaches regarding the main axis this thesis, namely design, manipulation and evolution in the context of data intensive systems. We outline gaps in those works with regard to our research questions.

2.1 Introduction

Before developing new approaches, techniques and tools contributing to our research questions, we have reviewed the literature in the relevant research areas. Those include (1) relational and NoSQL database design, (2) database manipulation, and (3) evolution of databases, both relational and NoSQL.

This state of the art allowed us to better understand the challenges, to refine our research questions and to identify possible gaps and new contributions to be made.

2.2 Database Design

Database design for NoSQL applications is an emerging research area. Current state-of-the-art approaches mainly consist of technology, data model-specific design recommendations or best practices [1–3, 7]. NoSQL databases are often based on schema-less data models and are therefore oriented towards developers. The underlying goal is to develop applications faster, while taking less care of the underlying database structures, which otherwise are more rigid and harder to evolve in combination with the programs. However, those advantages can eventually lead to poor performance as well, since a class embedding, redundancy or bad design decisions may significantly affect scalability, performance and consistency [62]. With regard to those problems the importance of NoSQL database design is gathering more and more attention [107].

In Section 1.1.1 we presented the process of database design which have been the established standard and good practice for relational database design for decades. This section will present design methods for NoSQL systems or hybrid polystores starting with generic approaches then detailing research focused on a specific data model.

2.2.1 Generic approaches

Atzeni *et al.* [17, 18] introduce NoAM (NoSQL Abstract Model), an abstract data model and methodology for NoSQL databases, which exploits the commonalities of various NoSQL data models, namely key value stores, column and document databases. They propose a database design methodology for NoSQL systems [28] based on NoAM, which aims to be (partially) independent of the specific target NoSQL platform. NoAM is used to specify a system-independent representation of the application data. This intermediate, pivot representation can then be implemented in a specific NoSQL database platform. The proposed method consists of four main steps:

- **Aggregate design**. Following use cases different entities are grouped together. This is consistent with the Domain Driven Design methodology [47]. It is based on conceptual UML class diagram where identification of aggregates is added.
- **Aggregate partitioning**. Performance requirements guide the smaller partition of aggregates.
- **High level NoSQL database design**. Using the NoAM model. It provides high level design such as Entry per Aggregate Object (EAO) or Entry per Top-level Field (ETF) using new abstract model terminology such as *Blocks, collections, entries, data unit*
- **Implementation**. Manual implementation of NoAM structures to the target datastore.

NoAM conceptual design method defines is based on an extension of UML class diagram which is transformed into a custom logical language with new notions such as *Blocks, collections, entries, data unit* representing common concepts of different NoSQL data models (document, key value and column based). The logical data representation is based on predefined mapping between the high level representations (EAO and ETF) and physical structures. It is not possible for the designer to explicitly specify the logical schema via another mean than the NoAM model elements. The user have to master the new definition introduced in the conceptual level as well as the implicit mapping between those elements and the effective storage of data.

In Banerjee and Sarkar [22] they define their own conceptual language for NoSQL databases with a three layer architecture, each having specific construct types. Those elements are then linked together via specific semantic relationships such as *association*, *inheritance* or *containment*. The logical model language is also generic, and they provide formalized rules and extensive constraints to enforce data validation and comply with the Basically Available, Soft state, Eventual consistency (BASE) or Consitency Availability Partition tolerance (CAP) principles. On top of having to learn a new conceptual modelling language the logical language is complex, not intuitive and have to be manually written. Further work from this author [21] address this issue and propose a model to model transformation method. Starting from the same conceptual model they provide transformation rules to produce a JSON schema. They state that JSON schema can then be transformed in any NoSQL data models. However, this affirmation is not straightforward for all data models, and this logical schema does not give a clear view of how the data will be stored *i.e.,* in a Redis key value database.

TyphonML model [23], supports conceptual modelling of hybrid polystores as well as physical modelling of database structures that can be linked together through the usage of mapping rules. However, it imposes implicit restrictions on the way conceptual entities, attributes and relationships are physically translated in each different native backend. In other words, TyphonML does not leave developers the freedom to explicitly define the mappings between the conceptual schema elements and the physical schemas of the polystore.

Mortadello [41] is a complete approach from conceptual modelling to database code generation. They developed a custom conceptual metamodel, Generic Data Metamodel (GDM), integrating an Entity Relationship (ER) like language to declare the application domain objects (entities, relationships, attributes) and a language to specify the access queries that need to be executed on this domain. This metamodel is generic and independent of any data model. From this model a Model-to-Model (M2M) transformation translates to logical schema of NoSQL compliant with defined meta models of document and column based data models. A set of algorithms read the queries and merge them if needed and then generate database code (CQL code or JSON Schema)

2.2.2 Document oriented

Shin *et al.* [118] describes the different levels of database design established by Peter Chen's and how they relate to the NoSQL data models. Further they demonstrate that the entity relationship diagram and relational model can be used in NoSQL database design. They propose to use UML notation for conceptual modelling and provide high level mapping transformation rules to transform to document database data model. For this purpose they propose an extension of UML integrating foreign keys and embedding possibilities.

De Lima *et al.* [42] proposes a transformation rule based approach to create a NoSQL document logical schema based on an Extended Entity Relationship (EER) conceptual schema. Additional information about the workload of databases *i.e.,* volume, access frequency and application load is added and used as input parameter in the transformation rules.

2.2.3 Column based

Mior *et al.* [95] propose schema design and queries implementation plans for column based database Cassandra. As input, they use (1) a conceptual model, an *entity graph* which is a restricted ER model, (2) a workload and query information to perform on this model and (3) space constraints. From those they produce a logical schema syntax with *partitioning* and *clustering* keys and column names best fitting the workload given as input.

Another approach to designing and implementing NoSQL databases is proposed by Abdelhedi *et al.* [10], they use a model driven approach and transformation rules on a conceptual database schema in order to create a NoSQL (specifically a column oriented) logical schema and then a physical NoSQL schema. It is a two-step process, a first one is to create a model to conceptualize the data independently of all technical and data model aspects (PIM *Platform Independent Model*). After that is the PSM *Platform Specific Model* which represents the data with regard to a specific data model, in this case it is the column-oriented model. The transition between the two models is done via a set of transformation rules in QVT language (Query/View/Transformation) following the OMG specification. This work uses a well-known language *i.e.,* UML as the high level language. Usage of model to model transformation is more efficient as no manual writing of the mappings is required by the user. However, this approach is only applied to a single NoSQL data model, *i.e.,* Column based (HBase and Cassandra)

2.2.4 Graph based

Akoka and Comyn-Wattiau [11] use Roundtrip engineering (RTE), a facet of Model driven engineering (MDE) to design NoSQL databases. Roundtrip is a combination of forward and reverse engineering, transforming conceptual models to source code and vice versa. It relies on bidirectional transformation rules between the metamodels considered. The advantages of this approach is that it handles evolution by systematically propagating changes upward and downward. This work is based on a previous one [12] proposing a MDE approach to transform a conceptual ER diagram into a logical property graph. Their conceptual model is an Extended Entity Relationship (EER) which adds information on the 4V's of big data [49], *Volume, Variety, Velocity, Veracity*. This logical representation can then be transformed to a physical schema, namely a script generating a Neo4j database.

Daniel *et al.* [40] offer a framework to translate an UML conceptual schema to an abstract representation of a graph data model. Moreover, they handle conceptual constraints expressed in Object Constraint Language (OCL). They combine model-to-model transformation (expressed in ATL [76]), to produce their intermediate model, and then use model to text transformation to generate code compatible with a known API accessing graph databases (Blueprints API). While their approach is complete, from conceptual model to access code generation and even Eclipse plugin tool available, their focus is solely on graph databases.

2.2.5 Key value

Rossel and Manna [106] provides guidelines to transform a conceptual schema expressed in ER or UML to a logical schema fitting the key value model. It also take into account the query patterns. No concrete syntax or rules are given and this method is mainly a general recommendation than a real transformation solution.

2.2.6 Summary and gaps

Table 2.1 summarizes the works described above with their supported data models (*D* for document databases, *KV* for key value, *C* for column based, *G* for graph databases, *R* for relational) and the chosen conceptual modelling language. The lower part of the table describes the representation technique used to establish the logical schema, and lastly it states the way to go from the conceptual to the logical schema.

In the analysed approaches to conceptually and physically model NoSQL databases or polystores we notice that many parameters and choices are possible. There is the possibility to focus on a single data model and keep its specificities or to decide to cover several ones and have to add a new abstraction layer. For the choice of the conceptual model we notice three trends, there are the approaches that (1) use a known language such as UML or ER, (2) extend or restrict these languages and (3) define their own language.

The possibilities for the logical language are all the more numerous as it is at this level that the contribution in this field of research is generally located. There is thus a dominance of personalized languages. Others have chosen to use known notations such as JSON Schema, UML, or even do not use an intermediate logical level and directly translate the conceptual schema into code specific to a database.

The passage from a conceptual schema to a logical or physical schema (column *Transformation*) can be of several types, either it is a question of transformation rules, of model-to-model type, more or less well formalized, or this transformation is manual, also more or less well guided.

While a transformational approach offers the advantage of avoiding the task of manually writing mapping rules between conceptual and logical representation it removes the control and the flexibility of defining fine-grained custom data representation. Contrary to some work exposed we do not provide a method for building the logical/physical schema nor to choose the best fitting NoSQL model.

The previous approaches discussed above are either (1) database design methods for particular or limited series of data models, (2) abstraction-based approaches for the conceptual design of multiple NoSQL systems, or (3) polystore modelling

Paper	Data model supported	Conceptual Approach
Atzeni *et al.* [17, 18]	D, KV, C	UML + identification of aggregates
Banerjee and Sarka [22]	D, KV, G, C	Custom Language
Banerjee and Sarka [21]	D, KV, G, C	Custom Language
Mortadello [41]	D, C	Custom Language
TyphonML model [23]	R, D, G, KV	ER
Abdelhedi *et al.* [10]	C	UML
Duggan *et al.* [46]	C, R	Only object declaration
de Lima *et al.* [42]	D	Extended ER
Rossel and Manna [106]	KV	UML or ER
Shin *et al.* [118]	D	UML
Mior *et al.* [95]	C	Restricted ER
Akoka and Comyn-Wattiau [12] [11]	G	UML
Daniel *et al.* [40]	G	UML + OCL

Paper	Logical/Physical Approach	Transformation
Atzeni *et al.* [17, 18]	Custom language	Manual, not flexible, not explicit
Banerjee and Sarka [22]	Custom language	Manual, explicit
Banerjee and Sarka [21]	JSON schema	Transformation rules
Mortadello [41]	CQL code + JSON schema	Transformation rules + Algorithms
TyphonML model [23]	Custom language	Manual
Abdelhedi *et al.* [10]	Platform Independent Model	Transformation rules
Duggan *et al.* [46]	Database	Manual
de Lima *et al.* [42]	JSON schema	Transformation rules
Rossel and Manna [106]	Custom language	Not provided
Shin *et al.* [118]	UML with custom stereotypes	Transformation rules
Mior *et al.* [95]	Custom language	Algorithm
Akoka and Comyn-Wattiau [12] [11]	Logical graph representation	Model Driven Approach + Roundtrip
Daniel *et al.* [40]	GraphDB metamodel (Platform Specific Model) + Gremlin	Transformation rules

Table 2.1: NoSQL design approaches summary

approaches with limited control over the conceptual-to-physical mappings and no support to express cross-database overlapping within the polystore. In this thesis, we propose an approach to hybrid polystore modelling and manipulation, that (1) supports relational as well as the four main NoSQL data models, (2) provides users with a full and fine-grained control over the mapping between the conceptual schema and the underlying physical data structures and (3) supports overlapping across the polystore databases.

2.3 Database Manipulation

In this section, we present existing approaches, techniques and tools manipulating databases. Presented approaches bring one or more abstraction level higher to the manipulation code and are able to query *hybrid polystores.*

NotaQL [115] is a modular language used to transform, migrate or perform aggregate queries on data. Queries are constructed using transformation expressions between input data and output results. The language is platform independent however queries still rely on effective physical field names. While it works with an internal unified data model close to JSON there is no higher abstraction of data available. Evaluation of the language is done using an algorithm combining Spark *map, filter and reduce* operations where only *map* functions are platform dependent.

The SOS platform [20] provides a common interface to multiple NoSQL systems. It relies on a generic data model expressing data as object entities with hierarchical structures and attributes. Operations on data are possible using PUT, GET or DELETE with path like queries (*e.g.,* `get("users/281283/name")`) on defined entities in the meta layer. Path like queries are encapsulated and called on manager class. A manager is implemented for each underlying type of database. Application code is resistant to NoSQL infrastructure changes, only the initialisation constructor of the manager would have to change to use another database.

BigDAWG [46] is another polystore implementation focusing on query optimization and data placement. They created the concept of *island of information* which is the abstraction the user interacts with. It is defined as *a collection of storage engines accessed with a single query language.* An island is composed of a logical data model of the specific databases, a custom query language and interfaces to data management systems translating queries to native ones. It supports less common NoSQL systems *e.g.,* Accumulo (column based), SciDB (Multivalue), and PostgreSQL (relational) databases.

In Shah *et al.* [117] they access data using a RESTful API and data wrapper objects. A query is handled by a data mapper calling specific data wrappers based on configuration files. Each data wrapper is written to query the corresponding database storing the data. Extension to new data stores is possible thanks to an agnostic architecture, one just have to write a new wrapper implementation and adapt configuration files.

Liao *et al.* [86] propose a data adapter approach to query and migrate data in a hybrid architecture composed of relational database and HBase, a column oriented database. They offer a SQL interface which seamlessly access either database via a translator module to native databases. Application queries therefore do not need to be adapted. A migration, called data transformation functionality and three different query policies are also available.

Object-NoSQL Datastore Mapper (ONDM) [27, 29] is a framework to map data to different NoSQL data representations. It is a variant of Java Persistence API (JPA) in the sense that application data is organized using classes and annotations. They are used to declare classes as entities, specify the data representations (following NoAM logical representations 2.2.1) via *@DataRepresentation* or declaring the data source. Data representations annotations allow the developers to specify the mappings in a declarative manner. Its architecture guarantees that the application code is independent of the target data store and of the data representation.

ODBAPI [116] propose a streamlined and unified REST API to access key value (Riak) and document (CouchDB) NoSQL databases as well as relational databases.

They defined their own model and terminology to map to the different concepts of the other data models. They model a polystore in terms of *Environment* which contains *Databases*, in their turns contains *Entity Set*, with *Entities*. CRUD functionalities are asked on each of the aforementioned resources using GET, PUT, POST and DELETE HTTP operations of their API. The ODBAPI provides a unique and simple entry point to different databases however the user has to known and provide the type of database where the data of the wanted entity is stored via an HTTP header in the request. This breaks the data independence principle with regard to the target datastore.

Xu *et al.* [132] propose to build a middleware system that will integrate relational database management systems and NoSQL systems. They develop a new language, called ZQL, which is platform-independent and allows users to write queries that will eventually be translated and executed using native queries. The ZQL module translates the given query to the corresponding system and return the results with full transparency for the user. However, this research is theoretical and has only been tested on a MySQL and Hive databases, which are both relying on the relational data model.

NoSQL Layer, proposed by Rocha *et al.* [105] is a framework to migrate data from relational databases to MongoDB. It contains two modules: The *Data migration* module and the *Data Mapping Module*. The data migration module automatically builds an equivalent structure in MongoDB and then migrate the data. The data mapping module is a data access layer; its goal is to act as an interface between the databases and the application programs. Its main feature is the ability to translate each SQL query into an equivalent MongoDB query, and then convert the returned query results in a SQL-compatible format. Therefore, in principle, no program adaptation is required.

TyphonQL [83] is a language for querying data among heterogeneous databases. It is a compiler executing queries expressed on conceptual element declared using the TyphonML language [23]. Join like queries across heterogeneous databases can be written and the TyphonQL engine will break them into multiple native queries which will then be executed in the corresponding databases. Finally, a uniform representation of the results is sent back to the user.

2.3.1 Summary and gaps

Table 2.2 presents the different query approaches for a polystore system as well as the supported data models, whether the way of performing queries is at a conceptual level or not, and finally briefly describes the type of these queries.

The table then continues to characterize these approaches by mentioning whether it is possible to perform queries on several databases via a single query (*Cross database queries* column) or whether the queries are expressed in a way that is independent of the actual data storage technology. Finally, we characterise if those queries are resistant to changes in the physical structure of the conceptual data. Among these approaches several interesting aspects stand out and our proposal aims at combining these aspects in a single tool. Our goal is to guarantee **data**

independence of the application code towards the system hosting the data as well as towards the representation of the data. We argue that only queries manipulating objects at the conceptual level of abstraction can achieve this. The identified papers use several query strategies, either a custom language, or a SQL-like language or finally an interface to access the data via generic *get, put, delete* methods. Our solution to the problem of data manipulation makes the choice of access by API methods rather than a language based approach. We detail here the differences as well as the advantages and disadvantages of the two techniques, provided that they both manipulate data at the conceptual level (TyphonQL [83] is taken as the comparison tool given the overlap of the supported functionalities).

- Simplicity. The method call approach allows providing intuitive names describing the function behaviour to users. Moreover, calling a function in a known programming language is also simpler than having to learn and build a query in a new language. This aspect is confirmed in the user evaluation, presented in Chapter 7.
- Expressiveness. The advantage for this criterion goes to the language approach, indeed if it supports nested queries, more complex queries can be written by the user. Whereas the use of pre-built methods of an API requires potentially large application code using sequence of loops.
- Portability/Usability. The API proposed by our framework provides Java methods. Its usage then requires the import of the generated library and code calling its methods. While using the language version, provided that the interpreter is installed, the user queries data with a string.
- Error tolerance. The way to execute a query influences error detection. Indeed, an erroneous use of the API will result in a compilation error that can be easily detected and corrected by the developer. Whereas an error in a string query will only be discovered during its execution. This aspect is particularly interesting in the context of the co-evolution between the database schema and the application code. As raised by Meurice [92] a schema evolution is not necessarily linked query adaptations, the code being still compilable.
- Development time. Both approaches share the same underlying global operations to be performed in order to return conceptual results. Those include the construction of native queries, the retrieval and joining of results and the encapsulation in a conceptual object. However, offering a new language to compile/interpret can be considered as an additional layer to develop. Therefore, the development time of an API version is naturally shorter.

Lastly, given that the context of use of these *data manipulation* approaches is a *hybrid polystore*, we attach importance to the capacity of this manipulation to perform queries on several databases simultaneously. This is to allow duplication and heterogeneity of data, but also to avoid the user writing complex join code.

The listed approaches all have advantages on particular aspects but none of them combines an integrated approach allowing at the same time (1) the manipulation of data at the conceptual level, (2) the management of multiple databases, (3) the independence from physical structures as well as from their evolutions.

Paper	Data model supported	Conceptual level	Type of queries
NotaQL [115]	KV, C, D, CSV	✗	Custom language. Declarative state IN vs OUT
SOS platform [20]	KV, C, D	✓Entities and attributes only.	Java API implementing basic get, put, delete operations
BigDAWG [46]	R, C	✗	Custom language
Shah *et al.* [117]	R, KV, C, D	✓	RESTful interface (GET, PUT, *etc.*)
Liao *et al.* [86]	R, C	✗	SQL
ONDM [27]	KV, C, D	✓	JPA
ODBAPI [116]	R, KV, D	✓Entities and attributes only.	RESTful interface (GET, PUT,...)
ZQL [132]	R, NoSQL (not specified)	✗	Custom (SQL like)
NoSQL Layer *et al.* [105]	R, D	✗	SQL
TyphonQL [83]	R, D, KV	✓	Custom language

Paper	Cross database queries	Platform independent queries	Resistant to evolution
NotaQL [115]	✗	✓	✗
SOS platform [20]	Partial, combination of get or put mapped to different dbs (no single get on mutliple db)	✓	✓
BigDAWG [46]	✓	✓	✗
Shah *et al.* [117]	✗	✓	✓
Liao *et al.* [86]	✓	✓	✓
ONDM [27]	✓(no multiple datastore for a single entity)	✓	✓
ODBAPI [116]	✗	✗	✗
ZQL [132]	✓	✓	✗
NoSQL Layer *et al.* [105]	✗	✓	✓
TyphonQL [83]	✓	✓	✓

Table 2.2: Summary of polystore database manipulation approaches

2.4 Database Evolution

Database schemas are known to be change-prone [131]. Even small changes to the schema may have significant impact on the application code [39, 61, 87, 119]. Large software systems also suffer from the heterogeneity challenge, as they tend to use multiple database access technologies jointly [59] [35], some of which complement and reinforce one another [60]. Studies of co-evolution of the database schema and the application code have revealed that these are not always in sync [101]. Database schema and format changes have been shown to impact the application code, which may lead to potential problems when the application code is separated from its persistent data or database [88]. Several approaches have been proposed that usually rely on (the combination of) transformational and generative techniques.

This section lists the main works that deal with how to perform database schema evolution, then we present papers that aim to help perform schema co-evolution together with manipulation programs. And finally we gather the researches that deal with schema evolution in NoSQL systems.

Supporting schema evolution

Hick and Hainaut [73, 74] propose the DB-MAIN approach to database schema evolution. This approach relies on a generic database model, namely the GER model [69], and on transformational paradigm that states that database engineering processes can be modelled by (chains of) schema transformations. Indeed, a transformation provides both structural and instance mappings that formally define how to jointly modify database structures and related data instances. The authors describe both a complete and a simplified, more pragmatic version of their approach, and compare their respective merits and drawbacks.

Database evolutions may involve schema modifications that can in turn impact the data instances and the database queries. Adapting data and queries to evolving schemas may constitute a long, risky and often manual process for database administrators. In [38], Curino *et al.* present PRISM++, a system that supports the database evolution process by evaluating the impact of schema modifications on queries and on data. PRISM++ then helps developers with the rewriting of historical queries and the migration of related data, thereby reducing the downtime of the system by reducing manual effort. They achieve this by defining a set of *Schema Modification Operators* (SMOs) representing atomic schema changes, and they link each of these operators with modification functions for data and queries.

The evolution operators considered by PRISM++ are inspired by the operators defined by Ambler and Sadalage [14]. They defined six main categories of operators, namely *transformation*, *structure refactoring*, *referential integrity refactoring*, *architecture refactoring*, *data quality refactoring* and *method refactoring*. In order to be even more precise, it is possible to further classify those operators into finer-grained atomic operators, as done by Curino *et al.*

In the same spirit, Qiu *et al.* [100] propose an exhaustive list of 24 schema change operators, each corresponding to an atomic DDL query.

Supporting schema-program co-evolution

The adaptation of client application programs under database schema evolution is a complex process. Most existing tool-supported approaches to this process rely on transformational techniques, generative techniques or a combination of both. For instance, several authors attempt to contain the ripple effect of changes to the database schema *e.g.,* by generating *wrappers* [126], views or APIs that provide/enable backward compatibility and by transforming the programs in order to interface them with those intermediate layers.

A first step towards program adaptation has been explored by Grolinger and Capretz [67]. They propose the integration of database accesses in the unit tests. They add a layer which effectively accesses the database instead of mocking it. In this way the actual queries can be checked against the (evolving) schema. If needed they also modify the queries in order to query the structure of the schema instead of the actual data, with the aim to increase data access performance. By validating queries to the schema they can identify source code fragments in the programs that have become invalid and that would therefore fail, in order to help programmers when adapting programs to evolutions of the database.

Cleve *et al.* [34] presented a tool-supported approach that combines the automated derivation of a relational database from a conceptual schema, and the automated generation of a data manipulation API providing programs with a conceptual view of the relational database. The derivation of the database is achieved through a systematic transformation process, keeping track of the mapping between the successive versions of the schema. The generation of the conceptual API exploits the mapping between the conceptual and physical database schemas. Database schema changes are then propagated through API regeneration so that client applications are protected against changes that preserve the semantics of their view on the data.

In [50] the authors provide a tool and program slicing technique specifically designed to adapt the programs source code as well as related regression tests when a database schema change occurs. This two-folded impact analysis method aims to identify the source code statements affected by the schema changes and the affected test suites associated to these source code fragments. They implemented their approach for PL/SQL applications.

Chang *et al.* [32] propose a formal framework for database refactoring which is based on a logical model of changes, and that can automatically identify inconsistencies in the application code as well as database modelling problems.

Meurice *et al.* [92] present a tool-supported approach, that allows developers to simulate a database schema change and automatically determine the set of source code locations that would be impacted by this change. Developers are then provided with recommendations about what they should modify at those source code locations in order to avoid query-schema inconsistencies. The approach has been designed to deal with Java systems that use dynamic data access frameworks such as JDBC, Hibernate and JPA.

Evolution of NoSQL databases

Recent approaches and studies have focused on the evolution of NoSQL databases. One of the first attempts on this topic was proposed in [110] by Scherzinger *et al.* The authors have addressed the gaps between schema evolution and data migration for managing NoSQL data stores. On the one hand, they have proposed a declarative evolution language specifying basic evolution operations such as add, delete, rename, move and copy. On the other hand, they have specified a NoSQL database programming language implementing these evolution steps by manipulating an *application state* and a *datastore state*. They have also introduced the concept of *safe migration* which verifies that an operation does not produce more than one entity of the same key.

The same authors have later specified, (1) in [112, 113], a data manipulation language using Datalog, capable of implementing chains of schema changes. They offer a logical view that allows the data to be migrated eagerly or lazily, transparently for the user. (2) ControVol [109], a framework controlling schema evolution in

NoSQL applications. It statically checks types of object mapper class declarations against earlier versions in the code repository. ControVol is also capable of warning developers of risky cases of mismatched data and schema. The tool also suggests and performs automatic fixes to resolve possible schema migration problems.

Evolving data in a NoSQL schema evolution context was also investigated by Klettke *et al.* [79]. They analysed lazy and eager data migration for big data applications in order to reduce time and financial costs. To do so, they have introduced innovative migration techniques such as (1) composite migration, which consists in grouping together chains of schema changes or (2) lazy stepwise migration, which consists in migrating lazily thereby waiting for a certain number of releases before migrating to the latest version.

Works regarding this data migration aspect may also be data model specific. Saur *et al.* [108] have suggested an online lazy data migration tool (Kvolve) for the key value datastore Redis [1]. The authors offer a unique logical view of multiple data versions and offer query translation mechanisms. If a new version occurs, the developer must provide a transformer function that will automatically remap the old key to the new one. Therefore, no changes are required by the end user applications.

Cleager [111] a tool for document databases to translate the evolution language operation into MapReduce jobs in order to migrate data in an eagerly manner. They provide support of evolution operations such as splitting or merging through implementing them via custom Map Reduce jobs.

Meurice and Cleve [91] provide a method analysing the joint evolution of the application programs and their underlying NoSQL data store. They use the application code and its evolution history to identify implicit changes in the data structure and potential points of failure due to this schema evolution.

Schildgen *et al.* [115] have specified a transformation language, called NotaQL, that is used to reach multiple goals: (i) to evolve schemas, (ii) to perform big data analytics using aggregation functions, (iii) to migrate data within the same database platform or even towards another platform. The proposed language is based on the idea of mapping transformation rules between the input format and the output format. Besides, the authors decided to offer an engine-specific syntax, allowing the user to work with familiar concepts.

Haubold *et al.* [70] have implemented ControVol Flex, that analyses the source code repository to detect potential errors. It then warns the users and recommends them automatic fixes. They have used the *NotaQL* [114] declarative transformation language to migrate data eagerly and Object Mapper Annotations to migrate data lazily.

Storl *et al.* [124, 125] have developed a GUI integrated NoSQL schema evolution tool called Darwin. The users can apply changes and migrate the data eagerly or lazily. Darwin also analyses data and code repository in order to detect possible schema changes. Once those schema changes are detected the tool can rewrite queries or migrate data.

[1] https://redis.io/

Klettke *et al.* [78] give the algorithm underlying the schema detection of their previous work. This algorithm detects chains of schema changes instead of a global unique schema. They are capable of detecting single type evolution operations such as *add, delete, rename* and even multi-type operation *copy, move.* This detection needs as a prerequisite persisted data with timestamps versions. The derived evolution steps are proposed to the user in case of ambiguities. They further evaluated their algorithm to detect inclusion dependencies in MongoDB data sets.

2.4.1 Summary and gaps

From the papers presented allowing support for the evolution of database schemas, we retain the transformational approach. The evolution process is indeed characterized as a sequence of transformations on the different components of the system, thus favouring co-evolution. Moreover, several works propose to characterize and specify the evolutions via explicit operators.

Then we have seen that the evolution of NoSQL databases adds some particularities to the data management. Indeed, these systems are not constraining regarding the data structures, so it is possible to have several versions that co-exist in the same database. This brings a dimension of data version management in the application code, but also opens the door to multiple data migration strategies. Such considerations should be part of a future integrated evolutive solution including NoSQL systems.

This thesis proposes several contributions to the evolution of databases. These are mainly about the co-evolution of schema and application programs and queries. First, we aim at supporting the evolution of data structures without conceptual modifications. Thanks to the **generative approach** and the conceptual accesses of our proposed solution for data manipulation, we can guarantee an automatic co-evolution, via regeneration, of the user application code.

In a second step, we offer an **adaptation of queries** according to conceptual evolution expressed with the help of conceptual evolution operators.

Finally, we end with a proposal of a **theoretical framework** for the evolution of polystores based on a transformational approach using evolution operators. These operators characterize the evolution to be carried out and specify the modifications to be made on each of the components affected by the polystore.

2.5 Concluding Remarks

In this chapter we have reviewed the state of the art for the three main research axes of this thesis, *i.e.,* the design, the manipulation and the evolution of data intensive systems. This state of the art was oriented under a *polystore* angle integrating a *heterogeneous* dimension, consisting of a co-existence of databases relying on relational as well as NoSQL data models.

As for the design part, we have seen that the engineering of relational databases was a well-known and established research field since decades, which brings maturity of methods and tools that NoSQL systems do not currently possess. Indeed, because of their variety of data models (document, key value, column or graph data models) and their variety of use cases, NoSQL systems offer a great diversity of physical data design methods for a single conceptual schema. This variety is all the more complex to manage and integrate in a context of *hybrid polystore*. This is why our approach proposes a modelling language integrating the conceptual schema, based on a well-established model (*i.e.,* Entity Relationship) as well as a fine-grained physical representation of the data for several NoSQL models.

Concerning the data manipulation in a polystore context we have privileged a conceptual access approach in order to guarantee the independence from the database and its structures. None of the mentioned techniques allows the combination of this independence with a complete support of a polystore system heterogeneity.

The approach chosen for the data manipulation also brings us solutions to facilitate the co-evolution of the application code with respect to a database schema evolution. Of which we have established the main characteristics and the most important points through the research described in the last part of this chapter.

The rest of this thesis presents the contributions made in each of these three aspects.

Part II

Contributions

CHAPTER 3

CHALLENGES AND BEST PRACTICES OF TESTING DATABASE ACCESS CODE

Contents

This chapter presents an exploratory work conducted on the testing of database manipulation code. An empirical study of the questions and answers from the developer communities allowed us to highlight the problems encountered and the best practices recommanded.

This work has been published [56] at the 31st International Conference on Advanced Information Systems Engineering (CAiSE 2019) and has been invited to publish an extension in the Information Systems journal [58].

3.1 Introduction

Through the introduction and the state of the art we have highlighted the main processes that are involved in the life cycle of a data intensive system, namely design, manipulation and evolution. We have noted that these processes are already complex enough for a single database system and we hypothesize that this complexity increases in a multiple database context (possibly hybrid). This increasing difficulty will not spare the code accessing databases, namely **database manipulation code**. Indeed, the developer will have to know and use several data manipulation languages (as described in Section 1.2) depending on the number of different database types used in his project.

This code, being located between the program and the databases, depends on the modifications made both in the program and in the databases. Evolving requirements result in changes in the schema, which in turn require adjustments in the database manipulation code. This *dual role* of database manipulation code leads to software maintenance problems. Stonebraker *et al.* argue that it is the most significant factor of database or application decay [122]. Developers tend to minimise their effort to implement modifications, and the application or the database quality suffers the consequences.

Considering the crucial role that this data manipulation code has, we first analysed whether this importance was reflected in the software tests. This chapter unfolds in three studies. In a first step we seek to **quantify** the coverage of database manipulation code by tests. This will tell us if the systems are sufficiently protected from implementation errors that may occur on this part of the code.

In a second step, we will analyse the difficulties encountered by developers when writing these tests, and we will produce a **taxonomy of issues** that will highlight the most significant avenues for improvement in order to help in writing these tests.

And finally we produce a **taxonomy of best practices** gathering the recommendations of the community developers to write quality tests.

3.2 Test Coverage of Database Access Code in Open-source Systems

We first explore how developers test their database manipulation code in practice. Figure 3.1 depicts an overview of the three main steps we followed during this exploration: ① we selected a set of open-source projects using databases, ② we identified which part of their source code was involved in database communication and ③ we analysed how automated tests covered it.

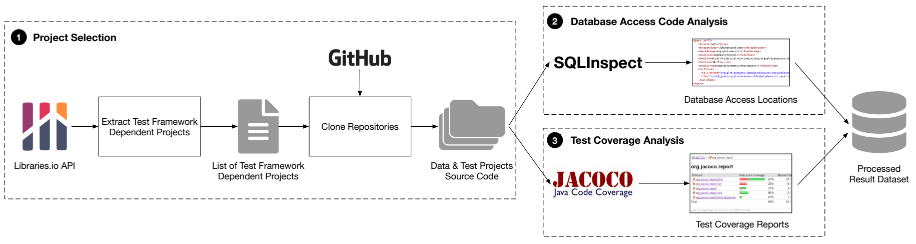

1 Project Selection
Libraries.io API
Extract Test Framework Dependent Projects
List of Test Framework Dependent Projects
GitHub
Clone Repositories
Data & Test Projects Source Code
2 Database Access Code Analysis
SQLInspect
Database Access Locations
3 Test Coverage Analysis
JACOCO Java Code Coverage
Test Coverage Reports
Processed Result Dataset

Figure 3.1: Overview of the main steps for test coverage analysis of database access code

During step ① *Project Selection*, we mined open-source systems from Libraries.io.[1] We chose it because they monitor a broad set of projects (not just libraries), and maintain an extensive database of dependencies among projects.[2] We specifically looked for applications using databases and automated testing technologies. Libraries.io provides us with the possibility of searching for such projects through their dependencies.

Selected projects had to satisfy four inclusion criteria: (i) *be written in Java*, since we rely on tools that support only Java (*i.e.,* to identify database code and measure test coverage); (ii) *use JUnit*[3] *or TestNG*,[4] *i.e.,* the top Java testing frameworks according to the usage statistics of Maven central;[5] (iii) *use database access technologies*, *e.g.,* `java.sql` or `javax.persistence`; (iv) *have executable test suites*, as required by JaCoCo,[6] the test coverage tool we rely on.

We relied on version 1.4.0 of the Libraries.io dataset published in December 2018, which was the most recent release at the time of conducting the survey. We cloned 6,626 systems satisfying a search query for Java projects with testing framework dependencies. Then we filtered them, looking for imports of database communication libraries. The list of imports can be found in our replication package [57]. At this stage, we identified 905 candidate projects.

In step ② *Database Access Code Analysis*, we identified the part of the source code involved in database communication. We used SQLInspect[7] for this purpose – a static code analyser for Java applications using JDBC, Hibernate, or JPA [96]. This tool looks for locations in the source code where queries are sent to a database, extracts these queries, and analyses them for further inspection, *e.g.,* smell detection. In the remaining of the chapter, we call *database access methods* all methods that construct or execute a DB query. We selected SQLInspect because (i) it supports popular database access technologies, (ii) it returns all the database access methods of the project under analysis, and (iii) it relies on a technique reaching a precision of 88% and a recall of 71.5% [93].

SQLInspect identified database access methods in 332 of the 905 projects selected at the first stage. It did not detect database accesses in the other projects. The reason is that SQLInspect looks for SQL, Hibernate, or JPA queries in the source code. An import does not necessarily imply query executions, and other DB communication means can be used (*e.g.,* an object-relational mapping; ORM), or the packages may not be used at all.

In step ③ *Test Coverage Analysis*, we looked at how tests cover the DB access methods. We used the JaCoCo Maven plugin that can be integrated with the tests of a project to collect coverage data at different granularity levels.

[1] https://libraries.io/data

[2] At the time of writing, Libraries.io had 2.7M unique packages, 33M repositories, and 235M interdependencies between them.

[3] https://junit.org/

[4] https://testng.org/

[5] `https://mvnrepository.com/open-source/testingframeworks`

[6] https://www.jacoco.org

[7] `https://bitbucket.org/csnagy/sqlinspect`

We implemented a script modifying the pom files of the 332 projects to execute tests with JaCoCo. Maven compilation or test execution failures prevented generating a test report file for 178 projects. For example, many projects (82) did not have a pom file or tests, despite dependency on a test framework. In the end, we collected test coverage data for 72 systems. Then we processed the reports along with the results of step ②.

Metric	Min	Q1	Med	Q3	Max
Java LOC (effective)	225	1,476	3,198	12,929	133,331
GitHub Stars	0	0	2	10	9,152
Methods	11	110	278	1,057	15,188
DB Access Methods (in prod. code)	1	2	4	7	80

Table 3.1: Overview of the projects (minimum, quartiles, median, and maximum values)

Table 3.1 summarises the main characteristics (with the minimum, quartiles, median, and maximum values) of the analysed projects. The projects are of various sizes ranging from 225 LOC to 133 kLOC. The biggest project is Speedment,[8] a Java Stream ORM. The most popular project is MyBatis[9] with 9,152 stars.

Regarding database access code, we only considered methods in production code, *i.e.,* we excluded test classes. We intentionally did not set a minimum threshold for the projects' size or database methods. Our goal was to see whether database access code is tested or not in real-life projects. If the project had only one method communicating with the DB, we wanted to see its tests.

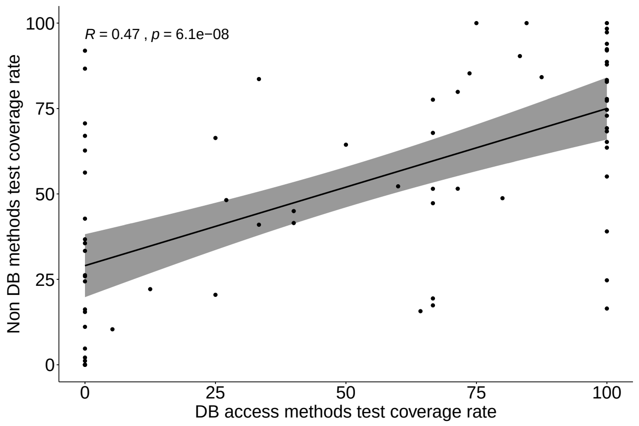


Figure 3.2: Test coverage rates of Non-DB access methods vs DB access methods

Figure 3.2 shows a scatter plot of all projects and their respective test coverage rates. In total, 24 projects do not test database access communication at all. A significant number of projects with the highest coverage rate had, in fact, full coverage. We

[8]https://github.com/speedment/speedment
[9]https://github.com/mybatis/mybatis-3

found a mean value of 2.8 database methods for projects with full coverage. There are slightly fewer projects (48.6%) in the figure with lower coverage for database methods. However, considering only the projects above the median (*i.e.,* with at least five database methods), there is a more significant difference: 59% have a smaller coverage for database methods than regular methods. Similarly, while 46% of the projects cover less than half of their database methods, this number increases to 53% for projects above the median. Moreover, 33% of the projects do not test the database code at all, and it rises again to 35% for projects with at least five database methods.

We assessed the relationship between the test coverage rates of DB access methods vs regular methods using the Kendall correlation, as the Shapiro-Wilk normality test showed a significant deviation from the normal distribution. The result was a moderate positive correlation with a high statistical significance ($\tau = 0.47$, $p < 0.0001$).

In summary, we found a statistically significant correlation between the test coverage of regular and database access methods, but it is a weak-moderate correlation, and there can be substantial differences between the two. As our closer look at the sample set showed, the coverage of database code is poor in general together with regular methods. But it is even more neglected when it comes to more complex database access code.

3.3 Challenges & Problems When Testing DB Access Code

Our first study aims to understand the difficulties of developers when considering database access code in their tests. We seek to answer the following research question (RQ):

> **RQ**$_1$: What are the main challenges/problems when testing database manipulation code?

We studied developers' most common problems on popular question-and-answer (Q&A) websites of the Stack Exchange network. The outcome of this qualitative study is a taxonomy of common issues faced by developers.

3.3.1 Context and data collection

We describe the method of building the taxonomy. We first present the data collection phase, then discuss the main steps of the manual labelling process.

Identification and extraction of questions

We targeted popular websites of the Stack Exchange network for data collection: Stack Overflow,[10] Software Engineering[11] and Code Review.[12] Stack Overflow is the

[10]http://stackoverflow.com
[11]http://softwareengineering.stackexchange.com
[12]http://codereview.stackexchange.com

largest Q&A website in software engineering, making it a popular target of mining studies. It included over 20M questions and 29M answers for software developers at the time of our analysis. Questions can be asked about specific programming problems, algorithms, tools used by programmers, and practical problems related to software development. Testing the database access code also falls into these categories. However, the guidelines say that *"the best Stack Overflow questions have a bit of source code in them."*[13] More generic questions, not closely related to source code, are often discouraged as out-of-scope or opinion-based. General discussions are preferred on Software Engineering. We included this site as we were interested in higher, conceptual-level problems as well, not only those related to the source code. Another valuable source for discussions in the Stack Exchange network is Code Review. There, developers can ask for suggestions on a given piece of code. As they often include test code, we considered questions from Code Review as well.

From these three Q&A websites, we selected our candidate questions according to the following criteria:

(a) *Scope.* We decided to select questions if (i) they explicitly mention testing in their title and (ii) they use database access terms in their description (*e.g.,* DAO, SQL). We loaded the dumps of Stack Exchange sites into a database for this filtering. We created full-text indices on both the titles and question bodies. Then we queried them, so the description had to match `(database | (data & access) | sql | dao | pdo) & test` and the title had to match `test`. The full-text search handled normalised text, so stemmed words were also considered (*e.g.,* test-*ing*, database-*s*). Notice that Stack Overflow has a tagging system for classification. However, using these tags is up to the user, who can easily omit them. Besides, the tagging system is different for the three sites considered, which led us to our alternative approach.

(b) *Impact and quality.* Due to the potentially large number of questions and limited resources, we targeted posts with higher impact and better quality. For this reason, we relied on the scoring system of Stack Exchange. No up-votes or a negative score may indicate problems, *e.g.,* an unclear or out-of-scope question. Therefore, we excluded posts with zero or negative ratings.

We used the Stack Overflow dump published by Stack Exchange in December 2019 and the dumps of Software Engineering and Code Review published in March 2020. A total number of 1,837 questions matched the criteria: 41 on Code Review, 174 on Software Engineering and 1,622 on Stack Overflow (see Table 3.2).

We did a first manual screening of questions on the different sites. We observed that Code Review and Software Engineering questions were closer to our scope. Therefore, we selected more questions from these sites and aimed for higher quality. To reach a 99% confidence level with a 5% margin of error, we set a threshold for a minimum score of 1 for Code Review, 3 for Software Engineering, and 13 for Stack Overflow.

[13]https://stackoverflow.com/help/on-topic

Source	**Candidate Questions**	**Selected Questions**	**False Positives**
Code Review	41	41	3
Software Engineering	174	140	25
Stack Overflow	1,622	351	86
Total	1,837	532	114

Table 3.2: Overview of the questions selected from Stack Exchange sites

Manual classification of database testing issues

After collecting the 532 questions, we manually inspected them. We followed an open coding process often applied to construct taxonomies or systematic mapping studies [99, 129]. In this approach, participants apply labels to concepts found in the text of artefacts. Then the tags are organised into an overall structure. During the process, labels and categories might be merged and renamed [99].

We performed the classification process in three rounds. First, we carried out a trial round with a random set of 100 questions, wherein two of the authors assigned labels to the artefacts. We wanted to test the classification platform and see whether we needed to apply changes to our selection criteria. After this trial round, we implemented a few adjustments to our platform. Then, we labelled the remaining questions in a second round by involving four authors. Each artefact was labelled by two authors, randomly assigned to them. In the last round, we resolved conflicts where needed.

We implemented a labelling platform for this purpose. It showed the question, its relevant metadata (score, timestamp, tags) and a link to the original discussion thread for further inspection. We followed a multi-label approach. Each participant could assign multiple labels to the artefact from the existing list in the database. If needed, they could also create new tags. In principle, existing labels should not be shown to participants. But as we expected a high number of tags, showing the existing ones could help us use consistent naming without introducing substantial bias. Indeed, the participants were not aware of the assignments.

After the second round, all 532 questions were labelled by two participants. At this point, one author reviewed all the tags and proposed merging those with identical meanings. This merging was discussed among authors and applied to the database.

We finally agreed and used identical tags for 147 questions; partially agreed for 77 posts (only a subset of identical labels) and used entirely different tags for 308 questions. The high number of unique tags explains this relatively high number of conflicts (72.37%). Indeed, at this point, the database had 290 different labels. Thus, participants took advantage of the multi-label classification and captured various aspects of questions.

To resolve conflicts, a third tagger was assigned to review each conflicting artefact. This third person was a randomly selected author who took part in the classification but did not label the same question beforehand. The system showed the labels of the previous taggers, and the reviewer could accept or discard them. Minor modifications were also allowed, if necessary.

At the end of the process, one author carefully reviewed all the tags and organised them into categories. This categorisation was then discussed among the authors in multiple rounds. As an outcome, a taxonomy was constructed with 83 database testing issues in 7 main categories. We present this taxonomy with qualitative examples in the rest of this section.

3.3.2 Taxonomy of database testing issues

Usman *et al.* reviewed taxonomies in software engineering and found the hierarchical form the most frequently used classification structure [129]. We adopted this representation as an efficient approach to organise our findings. In this form, there is a parent-child (*is-a*) relationship between categories, and one category has additional subcategories. Categories correspond to issues or problems raised in the question, and subcategories represent subtypes of a problem. Consider, *e.g., Mocking Persistence Layer* as a specialised type of *Mocking*-related issues.

Figure 3.3 shows the final structure of the taxonomy. There are a total number of 83 leaf issues organised in 7 main (root) categories. We indicate the total number of questions labelled with related problems for each root category. The distribution of the corresponding questions over the three sites is also provided. For example, the *Mocking* category had 54 questions, including 8 from Code Review, 17 from Stack Exchange, and 29 from Stack Overflow. Recall that we had a multi-label approach, so one question could represent mixed problems. Thus, a question can belong to more categories in the hierarchical taxonomy.

We observe intriguing technical and conceptual difficulties, differentiating between them in Figure 3.3 as follows. We mark the technical problems with ⑂ and the conceptual ones with 🎓. It is interesting to observe the origin of questions for those abstraction levels. Higher-level conceptual problems mainly originate from Software Engineering, especially for *Maintainability/Testability* or *Method*. Technical problems are closer to the source code and mostly originate from Stack Overflow, especially for the *Framework/Tool Usage* category. Questions from Code Review cover both abstraction levels, but most of them relate to the general *Best Practices* category. None of them deals with *Framework/Tool Usage*. Below, we describe and illustrate each main problem category.

⑂ DB Management

The most prevalent technical issues are related to database management: we found 145 questions in this category. Indeed, many have problems initialising the database before executing the tests. This includes starting the database, configuring it, and populating it with test data. The test database population was often mentioned as a root cause of performance issues. These initialisation steps are critical as they must be performed before test executions. As a developer complained: *"This whole thing takes quite some time [. . .]. Having this run as part of our CI [. . .] is not a*

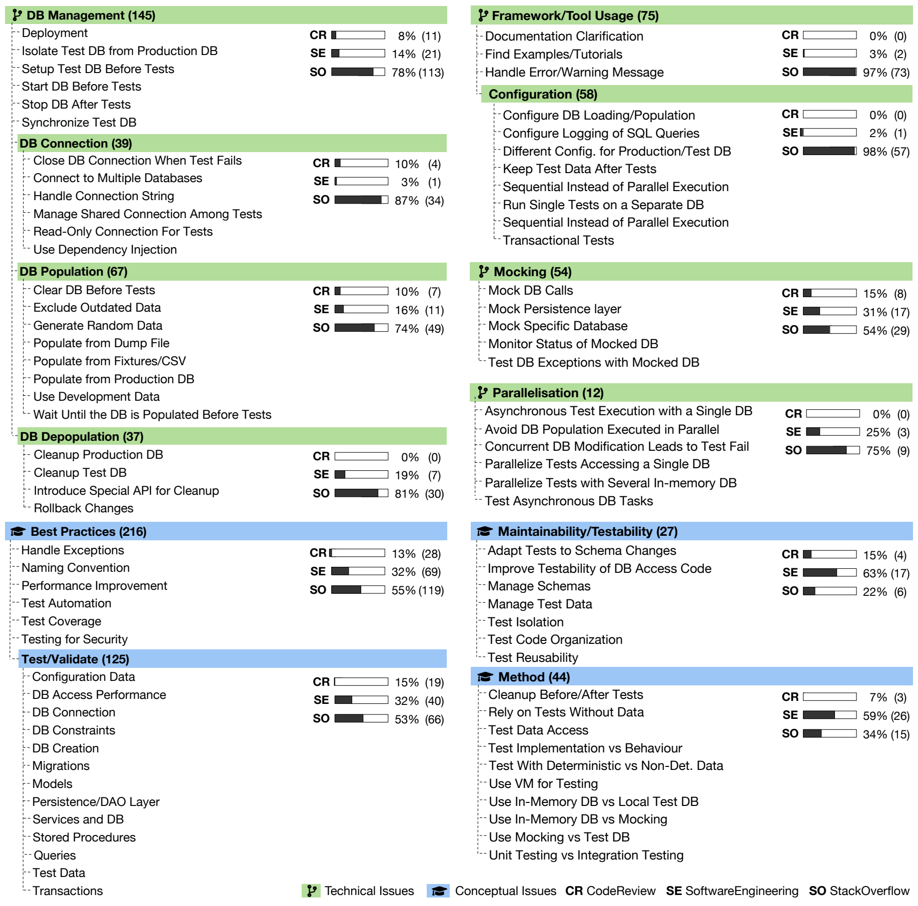


Figure 3.3: Taxonomy of issues faced by developers when testing database access code

problem, but running locally takes a long time and really prohibits running them before committing code" (*SE1*).[14]

Questions also came from situations when the design did not support data deletion (*SE2*). Others faced issues keeping a test DB in synch with a production or development DB (*SE3*), while many had problems handling the connection to a test DB (*SE4*, *SE5*, *SE6*).

Framework/Tool Usage

Many problems (75) concern using a concrete tool or framework. Most of them relate to configuring a framework for a dedicated database in a test/development or production environment (*SE7*, *SE8*). These questions have high scores suggesting

[14]We cite posts on Stack Exchange sites with *SE* notation. These references can be found in Table A.1 of Appendix A

that many developers suffer from such issues. A question to configure Django (*SE8*) was voted up 59 times and stared by 16 users.

Similarly, developers ask help for different DB initialisation (*e.g.,* running scripts, using dumps or fixtures) or cleanup configurations (*SE9*). Interestingly, in some cases, they want to keep the test database after running their tests for debugging purposes (*SE10*). Many also ask for guidance to solve a particular error message in the testing framework, *e.g.,* misusing transactional tests (*SE11*) or configuring in-memory databases (*SE12*).

Mocking

Mocks can help by isolating the tests (*i.e.,* cutting off dependencies) and avoiding the performance drawbacks of databases (*e.g.,* avoiding IO). Many questions indicate that developers need help in mocking the persistence layer. As a first step, an important design decision they have to make is the level at which they implement the mocks.

For example, a developer reasoned in a question as follows: *"I could either mock this object at a high level [...] so that there are no calls to the SQL at all [...]. Or I could do it at a very low level, by creating a MockSQLQueryFactory that instead of actually querying the database just provides mock data back"* (*SE13*). Recommendations depend on the objectives, as an answer says: *"Higher level approaches are more appropriate for unit testing. Lower-level approaches are more appropriate for integration testing."*

Broader questions were also about the benefits of mocking (*SE14*) or guidelines to mock the data access layer (*SE15*, *SE16*). Technical questions tackled, for example, emulating exceptions in a mocked database (*SE17*). When mocking is unfeasible, it can indicate poor software design (*SE18*). Stored procedures (*SE19*) and views (*SE20*) made mocking impossible in other systems.

Parallelisation

We observed some (12) technical problems related to parallel test executions. These were closely related, so we grouped them in this category. One of the highest-rated questions was about turning off the parallel execution of tests in *sbt* (a build tool for Scala and Java) (*SE21*). The developer complained that a project *"mutates state in a test database concurrently, leading to the test to fail."* Likewise, asynchronous or lazy calculations led to challenging bug hunts (*SE22*). They also asked for advice to parallel test execution, *e.g.,* to handle a dedicated in-memory database per thread (*SE23*).

Best Practices

The most frequently used labels were about testing best practices for DB applications. Developers either look for general advice or explicitly want to know about best practices. The highest-rated question has 331 up-votes entitled *"What's the best* *strategy for unit-testing database-driven applications?"* (*SE24*). It generates discussion on mocking vs testing against an actual database. In the answers, mocking is mainly recommended for unit testing, while a copy of the database is favoured for more complex databases. In other cases, a combined approach might be needed: *"Ideally I want to test the data access layer using mocking without the need to connect to a database and then unit test the store procedure in a separate set of tests"* (*SE15*).

Best practices are also sought for performance improvements (*SE25*, *SE26*, *SE27*). In particular, where mocking is not an option, solutions mainly advise using in-memory databases to reduce IO operations. Other topics include testing for security vulnerabilities, *e.g.,* looking for static analysers to spot SQL injection attacks (*SE28*). Likewise, some questions look for tools to measure test coverage. They want to know, for example, the coverage of executed queries in test cases (*SE29*). A majority of these questions were grouped under *Test/Validate*. These are looking for advice on testing or validating a specific code or DB entity, *e.g.,* SQL queries embedded in code (*SE30*), database migration (*SE31*) or transactions (*SE32*).

Maintainability/Testability

Several questions tried to address maintainability problems or the testability of the database access code. In a question, a developer struggled with a system that validated RESTful APIs with SQL queries in its integration tests (*SE33*). As he summed up his root problem: *"A small change in the DB structure often results in several man days wasted on updating the SQL and the SQL building logic in the integration tests."* The developer wanted to wipe out the SQL code from the tests entirely. In the answer, they discouraged him from doing so. They acknowledged that relying on the queries can be a good practice to verify the database state. Instead, it was recommended to improve the maintainability of the tests: (i) by reducing the coupling inside the codebase (one table per module), and (ii) by splitting the tests into smaller pieces.

In another question, a developer wanted to reduce the maintenance effort by omitting the tests of the ORM layer. He was, however, afraid of giving up on aiming for 100% coverage. As he wrote it, *"Our test databases are a bit messy and are never reseted, hence it's impossible to validate any data (and that is out of my control)."* In the answers, they supported him that balancing coverage and prioritising efforts is important, then suggested generating the tests for the ORM layer.

Other questions pointed out that preparing the environment of testing the database access code is also troublesome. For example, a developer complained: *"The problem I ran into was that I spent a lot of time maintaining the code to set up the test environment more than the tests"* (*SE34*).

Many questions were also related to the management of changing schema or test data. As a general guideline, a recommendation said: *"I would apply a single rule: keep your test data close to your test. Test is all about maintenance: they should be designed with maintenance in mind, hence, keep it simple"* (*SE35*).

Method

Many developers were concerned about the problems of their testing method. The most frequent arguments were whether DB-dependent code should be tested via unit or integration tests (*SE36*, *SE37*, *SE38*, *SE39*, *SE40*). A regular claim was that *"unit tests should not deal with the database, integration tests deal with the database"* (*SE37*). Recommendations target to maximise the isolation of unit tests and decouple the database, *e.g.,* through mocking. In contrast, integration tests aim to test more complex structures by relying on the database.

Interesting questions were related to populating a database before tests, *e.g.,* whether data should be dynamically generated or pre-populated beforehand (*SE41*).

A recurring discussion was on using an in-memory database versus a mocking strategy (*SE42*). When performance or decoupling the tests from the database was more critical, the choice was to mock. Otherwise, we could see cases where mocking was impossible (*e.g.,* because of stored procedures or views). The in-memory database was considered a good compromise to test the database access. It indeed solves the portability issues of testing against an actual DB and improves the performance. Compared to mocking, the testing can be more extensive, *e.g.,* it enables the tests to validate embedded SQL queries. In some cases, however, the in-memory database differs significantly from the production database. This can be a problem as some DB-specific features cannot be tested, *e.g.,* a special SQL syntax (*SE43*).

3.4 Best Practices When Testing DB Access Code

In the previous study, we investigated the challenges of testing database access code. Here, we study the solutions proposed by developers. We seek to answer the following research question:

> **RQ_2:** What are the best practices when testing database manipulation code?

We assessed RQ_2 by studying answers to the StackExchange sites' questions in our previous study. The outcome is a hierarchical taxonomy of best practices recommended by the developers.

3.4.1 Method

We conduct this research with an open coding process similar to the labelling in our first study (see Section 3.3.1). First, we describe this process by presenting our dataset preparation, and then we discuss the details of the manual labelling.

Dataset preparation

We took RQ_1's dataset of questions about database access code testing from Stack Overflow, Software Engineering, and Code Review. We loaded the same data dump versions into a database to remain consistent with our previous research question. We exported all the answers to the 418 questions labelled previously (532 questions excluding 114 false positives). Again, we filtered the answers with negative scores as they usually suffered from quality issues, *i.e.,* they could be wrong, incomplete, or irrelevant to the question. Next, we picked the top three highest-rated answers for each question. When the accepted answer was not among the top three, we included it as a fourth answer.

			Scores			
Site	**Questions**	**Answers**	**Min**	**Max**	**Avg**	**Median**
Code Review	38	50	0	19	3.22	2
Software Engineering	115	243	0	182	5.70	3
Stack Overflow	265	691	0	545	21.10	10
Total	418	984	0	545	16.39	6

Table 3.3: Overview of the selected answers and their scores

Table 3.3 presents an overview of the answers to the questions after the selection process. The table shows, for each Stack Exchange site, the number of questions, answers, and statistics (min, max, average, and median) of their scores. Overall, we had 984 answers to 418 questions in our database. The highest-rated answer had 545 upvotes. It responded to a Stack Overflow question about *"MySQL - force not to use cache for testing speed of query"* (*SE44*). Also interesting to notice an answer to a Software Engineering question, which had 182 likes. The question with the title *"Shouldn't unit tests use my own methods?"* addressed the unit testing of Data Access Objects' methods (*SE45*).

Manual labelling

We used a similar open coding process to RQ_1 and manually labelled the answers. However, we had some significant differences, which we explain below.

First, we needed to adapt the labelling platform to the answers, as it was designed for labelling questions. When someone started tagging, the platform randomly assigned a question to the user and showed it with the highest-rated answers. The platform also displayed the metadata of the question (score, time, URL) and its answers (score, URL, and whether it was accepted). It also presented the problem categories assigned to the question in the previous round.

We could assign multiple labels to each problem category for each question, covering all answers. For example, if a question had been labelled in RQ_1 as "*Best Practices > Test/Validate > Queries*" and "*Mocking,*" one could assign a "*Don't mock the connection*" tag to the mocking category and another "*Avoid In-Memory DB as it might not be fully compatible*" tag to the query validation.

This is a significant difference from RQ_1, where we assigned problem labels directly to the question.

Second, we added a feature to highlight relevant sentences in questions or answers. We needed this feature as the answers had many exciting ideas, guidelines, or takeaway messages, which could easily get lost in the longer texts and code examples. After our first trial round, we also anticipated that highlights would be helpful when reviewing each others' tags. We highlighted 835 text fragments, in the end, 3.27 on average per question. The highlighted sentences are available with our tagging in the replication package [57].

Third, we simplified the process, and each question was first tagged by an author then reviewed by another one. In RQ_1, two authors labelled each question independently, and then a third one reviewed it. We altered the process because we experienced many conflicts due to the large number of tags, and the reviewing typically meant merging the two authors' tags. In RQ_2, the second tagger had an explicit reviewer role instead of an independent tagger.

All five authors participated in the labelling. Like in the first study, we conducted the process in three main rounds. First, we had a trial round of 27 questions, adding the highlighting feature and minor fixes to the platform. Then, we performed the first labelling round, followed by the reviewing round. After each round, we held discussions among all authors, shared our experience, and renamed or merged tags where needed (*e.g.,* because of their identical meaning).

We did not label all the questions from RQ_1 because of limited time constraints. Instead, we focused on *Best Practices*, the most extensive and significant problem category (see Figure 3.3).

	Questions		**Answers**	
Issue Category	**Total**	**Labelled**	**Total**	**Labelled**
Best Practices	216	216	510	510
DB Management	145	57	351	142
Framework/Tool Usage	75	34	185	75
Maintainability/Testability	27	16	50	32
Method	44	21	106	56
Mocking	54	34	123	75
Parallelisation	12	3	29	7
Total	418	255	984	598

Table 3.4: Total and labelled questions per main issue categories

Table 3.4 presents an overview of the labelled questions and their answers per each main issue category. A question could belong to multiple problem categories in RQ_1. Thus, "Total" represents the union of the questions, and its figures do not necessarily equal the sum of all categories. We labelled 598 answers of 255 questions, 61% of all the answers and 100% of the **Best Practices** category. Among these, we found 4 questions without answers, and 18 with irrelevant answers, *e.g.,* they were unrelated to database manipulation code.

Finally, one author carefully reviewed all the tags and organised them into a hierarchical taxonomy. This categorisation was then discussed among the authors in multiple rounds. The final taxonomy had 363 tags in 9 main categories. We present this taxonomy through qualitative examples in Section 3.4.2.

3.4.2 Taxonomy of database testing best practices

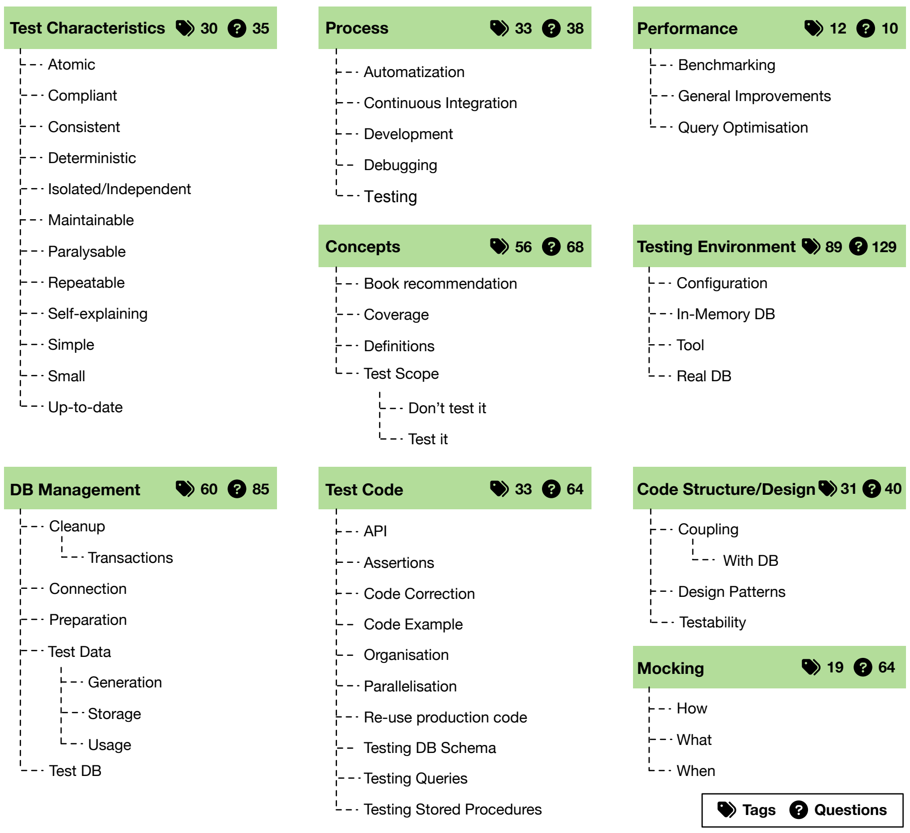


Figure 3.4: Taxonomy of best practices proposed by developers when testing database access code

Figure 3.4 shows the main categories in the taxonomy of database testing best practices. The taxonomy follows the same hierarchical structure as we described in Section 3.3.2. We had a total number of 363 tags at the end of the manual labelling process. Each tag represents a solution to a problem fitting into contexts like *"The developers recommend..."* or *"The solution to this problem is..."*. We organised the tags into 9 root categories following a similar classification to the taxonomy of issues. The figure presents the number of questions (❓ icon) and the number of tags (🏷 icon) for the root categories. The lower-level tags, that are not in the figure, are listed in the replication package [57]. In the rest of the section, we describe each category through intriguing examples.

Test characteristics

Many answers expressed that sound tests should adhere to specific characteristics and principles. Suggestions are mainly general, such as the *FIRST* (Fast, Independent, Repeatable, Self Validating, Thorough) principle (*SE46*), or that tests should be *atomic, small, simple, consistent* and *in compliance with requirements* (*SE47*). Many highlighted that tests should be *independent/isolated.* For example, an answer says that *"if the tests can not run independently, then they are not unit tests"* (*SE48*). This is especially important for database-centred applications where it can be tempting to write tests relying on a database state from a previous test, which would be against the rule of isolation. It can also affect *repeatability* and further complicate test failures. An answer points it out, *"relying on the order of your tests indicates that you are persisting state across tests. This is smelly"* (*SE49*). Leaving the database in a consistent state requires extra effort. Developers have various suggestions, *e.g.,* transactions, in-memory databases, mocks, fakes, stubs. We have more specific tags for these in the following taxonomy categories. It is also interesting to note the *up-to-date* test property, highlighting the importance of syncing tests with changes applied to the production database (*SE50*).

Code structure/design

This category contains recommendations targeting source code structure or design. We grouped tags into three subcategories: *Coupling, Design Patterns,* and *Testability*. Although they are interrelated, we differentiated them based on the primary aim of the suggestion.

Coupling was a primary concern for testing: 15 questions had answers falling under this category. We identified tags such as *design with loose coupling for better mocking* or *decouple tests from implementation.* An answer noted: *"Make sure you design them [service classes] with loose coupling in mind so you can mock out each dependency"* (*SE51*).

We separated a subgroup where they specifically targeted coupling *With DB*. Common recommendations were to *keep logic out of the database*, have a *separate data access layer*, and *decouple the data layer.* As a developer said, *"I would strongly suggest decoupling it [the data access layer] from both the web and from the DB"* (*SE52*). Others added, *"if your objects are tightly coupled to your data layer, it is difficult to do proper unit testing"* (*SE53*); *"The test knows too much about intimate details of the implementation"* (*SE54*). Developers also proposed keeping logic out of the database (*SE19*, *SE55*). An answer stressed that *"too much business logic is making its way into databases these days"* (*SE56*).

Recommendations also aimed for general *testability* improvements, such as a *design with single responsibility* and *break the code down to smaller testable units* – for stored procedures too (*SE57*).

Many suggestions mentioned *design patterns.* For example, developers recommended *dependency injection* in the answers to 14 questions. They employed it to automatically inject a connection to a test database instead of the production database.

Developers said, *“using dependency injection, have the unit tests select a different database than what the production (or test, or local) builds use”* (*SE58*); *“I’ve found that dependency injection is the design pattern that most helps make my code testable”* (*SE59*). A design recommendation was also to have a dedicated *base class* or *template interface* for tests involving database access code, *e.g.,* simplifying database initialisation, cleaning up, or mocking (*SE60*).

Concepts

We found valuable discussion threads about definitions, various aspects of tests, and coverage. We grouped them under the *Concepts* category. In particular, 34 questions were tagged as *explanation of levels of tests: unit, interaction, integration, and acceptance tests*, the second most used tag.

Many answers state the differences between unit and integration testing when databases are involved. A common argument is that unit tests should test their units in isolation; hence, they should mock, fake, or stub the database. In contrast, integration tests consist of an actual test database with test data reset in a known state before and after each test.

An answer summarises it as follows: *“A unit test deals with a part of the code which is granular enough to be able to narrow the search of a bug if the unit test fails. There is no long polling here. No REST calls. No AJAX. No database access. REST, access to files, database calls, and all those operations which are exterior to the tested code are mocked, i.e. a mock or a stub is created for everything your tested unit needs. [...] Once you have unit tests covering the critical parts of the application, you can start assembling the parts. Interfaces between different components of a system are good places for mistakes, so the integration of components needs to be tested as well. This is what integration tests are about”* (*SE61*).

We found interesting discussions about *Coverage*. They generally agreed that border and corner cases must be covered (*SE62*). They also argued the importance of testing data access code: *“If you don’t test your database operations, how do you know that your data access component works?”* (*SE63*)

The *Scope* subcategory collects answers about *what to* or *what not to* test. Developers suggested *not to test* (i) anything that can’t fail (*SE64*), (ii) third-party code (*SE65*) and (iii) code without logic (*SE66*). For example, an answer said, *“there are many purists who say that you shouldn’t test technologies such as EF and NHibernate. They are right, they’re already very stringently tested and [...] it’s often pointless to spend vast amounts of time testing what you don’t own”* (*SE65*). Another argued, *“if it’s really thin and there’s no interesting code there, don’t bother unit testing it. Don’t be afraid to not unit test something if there’s no real code there”* (*SE66*). The data access layer has a special role in this case, as noted by a developer: *“Some people [...] say you should only test code which has conditional logic (IF statements etc.), which may or may not include your DAL [Data Access Layer]. Some people (often those doing TDD [Test-Driven Development]) will say you should test everything, including the DAL, and aim for 100% code coverage”* (*SE67*).

What to test was more sparse with suggestions such as *check both entities and queries, one test for each type of output resultset (one row, multiple rows, empty resultset), test DAO[Data Access Object]/Repository normally if it performs any logic*. Developers suggested tests for pre- and postconditions (*SE68*), query correctness (*SE69*, *SE70*, *SE71*), the connection string (*SE5*), database schema (*SE15*), UI (*SE72*), and CRUD operations. The importance of the latter one was highlighted in an answer as follows: *"To really test your service layer, I think your layer needs to go down to DLLs and the database and write at least CRUD test"* (*SE66*).

We also found book recommendations, *e.g., Growing Object-Oriented Software, Guided by Tests* (*SE73*, *SE74*), *The Art of Unit Testing* (*SE75 SE76*), and *xUnit Test Patterns* (*SE76*).

Database management

The second most prevalent category was *Database Management*. The same problem category was also significant in the taxonomy of issues. We divided it into subcategories, *i.e.,* the preparation or clean up of a test database, the generation, storage, and usage of test data, or handling the connection to the database.

The tags revolve around reaching a known state before each test execution (*e.g., SE77*, *SE78*). Indeed, the recommendation to *clean up before each test (known state)* was the most prevalent, with 20 occurrences. We have seen mainly two best practices as follows. (i) *Using an actual database*, loading the schema and test data before each test, then cleaning the database before the next test case. There are various tuning practices. For example, one can optimise database initialisation by loading the schema once then populating only with necessary minimal data. Then clean up only the modified records if there were any. Many also proposed in-memory databases for performance reasons or simply because they are easily destroyable after each test run. (ii) The other thread was about *transactions*, *i.e.,* load the schema and test data, run the tests in transactions, then rollup changes. We tagged these as *use transaction scopes (which revert the state of DB after each test)*.

Interestingly, many frameworks provided support for these techniques. We counted answers recommending *setUp() & tearDown() methods* for database initialisation and clean up in unit tests. *Spring* also supports test execution through *transaction management*.

Developers had intriguing arguments for these approaches as follows. *"Just remember: At the start of the test, everything is created, at the end of the test everything is destroyed"* (*SE79*). *"I suggest either connecting to an empty DB and filling with data in the test set-up phase, then either emptying it or deleting it in the test clean up phase or creating a copy of a constant DB, connecting to it in the test set-up phase, then deleting in the test clean up phase. It is important to do this per test, so that the tests are truly independent"* (*SE55*). *"Using setUp() and tearDown() to get a consistent state for your data before running your tests is (imho) a fine way to write DB-driven unit tests"* (*SE80*).

We also grouped generic DB-related recommendations in the *Test DB* subcategory. For instance, an answer advised avoiding a different type of DB than in production to prevent cross-platform issues (*SE81*).

Finally, the *Test data* subcategory consists of recommendations to generate, store, and use testing data. For example, *use the ORM [Object-Relational Mapping] to initialise test data* (*SE65*) or *generate random but valid data entries* (*SE82*).

Mocking

We separated a *Mocking* category in the taxonomy due to the prevalence of these tags. The *use Mocking* tag was assigned to the most questions; we encountered it 46 times. A recurring discussion of *Concepts* argued unit tests should imply mocking. An answer stated it as follows: *"If you test class B, which is a client of A, then usually you mock the entire A object with something else, [...]. Likewise, when you write a unit test for class C, which is a client of B, you would mock something that takes the role of B"* (*SE5*).

Answers proposed *what* to mock, *e.g., "You shouldn't mock calls to the database because that would defeat the purpose. What you SHOULD mock are, for example, calls to your DAO from, say, a service layer. Mocking allows you to test methods in isolation."* (*SE83*). Others elaborated on *how* to mock and presented complete code examples (*SE84*). Mocking frameworks (*e.g.,* Easymock, Mockito, Moq, Rhino Mocks) were often recommended. Many threads also discussed the differences between mocks, stubs and fake objects (*SE85*, *SE86*, *SE27*).

Interestingly, a few answers hinted that caution is needed in using mocks (*SE87*, *SE88*). An answer argues that *"I'd try to use them [mocking] sparingly in unit tests since by using them you actually try to test the function implementation and not the adherence to its interface"* (*SE88*).

Performance

We separated a group for performance tuning or optimisation recommendations. These were often context-specific or fine-tuning alternatives. For example, an answer (*SE81*) recommended using *tmpfs* (*i.e.,* run the database on an in-memory filesystem) when a *per se* in-memory database was not a viable option (*e.g.,* because of a different SQL dialect). Another answer suggested using prepared statements in loops (*SE82*) or deferring garbage collection for the tests (*SE27*). A recurring recommendation was using a lightweight database (in-memory) to optimise performance (*SE89*, *SE90*).

Process

Process category groups tags about automatisation, continuous integration, test failure debugging, or the testing process in general (*e.g., SE78 SE50*). The most recurring discussions revolved around integration and unit tests. In particular, they suggested separating integration from unit tests (*SE5*), highlighted the importance of automated UI tests (*SE91*), or proposed testing the persistence layer manually (*SE74*).

Recommendations were also related to debugging, *e.g.,* logging failing or slow queries (*SE92*). They also suggested integration testing in certain situations (*SE93*, *SE31*). As an example, an answer says, *"there's no way to unit test Spring Data JPA [Java Persistence API] repositories reasonably for a simple reason: it's way too cumbersome to mock all the parts of the JPA API we invoke to bootstrap the repositories. Unit tests don't make too much sense here anyway, as you're usually not writing any implementation code yourself [...] so that integration testing is the most reasonable approach"* (*SE94*).

Test code

Recommendations also focused on the source code of the tests, *e.g.,* recommended specific APIs. We group these under the *Test code* category. The answers include configuration fixes, general how-tos of APIs, code examples, or asserts in tests.

For example, developers often stress the importance of *single asserts*. An answer says, *"The tests are far more granular, each test verifies one property [...] single asserts are good"* (*SE65*). Another developer argues that *"there should only be one reason for a test to fail"* (*SE95*).

We learned that many testing frameworks actually provide APIs to support database testing. In particular, we found API recommendations for *Android*, *Entity Framework*, *Django*, *LINQ*, *Spring Framework*, and *NUnit*.

Testing environment

We separated a group of *Testing environment* recommendations. This group has the highest number of tags and questions due to the several tools named in the answers. It consists of advice about the application or build frameworks, configurations, database management system, or various tool proposals.

Many answers recommended in-memory databases, *e.g.,* H2, HSQLBD, HyperSQL, Ephemeral PG, and SQLite. These are not exclusively relational. For example, a thread talks about using MongoDB in memory mode (*SE12*).

Tool recommendation includes also mocking libraries (*e.g.,* Easymock, Mockito, Moq, or Spring Data Mock). Database configuration concerned various technologies such as Laravel (*SE96*), JUnit (*SE97*), Spring Boot (*SE98*), Django (*SE99*, *SE100*). These were mainly related to configuring a test database, fixtures, or migrations. Flyway and Liquibase were also notable tools in this context (*SE81*). They were mentioned to manage (track, version, and deploy) database schema changes.

We also found tool recommendations for vulnerability testing, *e.g.,* SQL injections (*SE101*, *SE102*), and tools for virtualisation or container environments (*SE81*).

Finally, it is interesting to notice that 24 answers proposed DBUnit,[15] a testing framework for database-centred applications (*SE103*, *SE89*, *SE77*, *SE81*).

An answer remarks that *"the DbUnit framework (a testing framework allowing to put a database in a known state and to perform assertion against its content) has a page listing database testing best practices that, to my experience, are true"* (*SE78*).

[15] https://www.dbunit.org/

3.5 Discussion and Implications

Below, we discuss the main observations we made in our investigation, together with future directions for researchers and practitioners.

Maintainability of database tests Test maintenance was a frequent issue. A developer aptly outlined, *"if it is hard to maintain, you're doing it wrong"* (*SE39*). Many answers recommended following sound characteristics or principles. However, their implementation guidelines were often unclear. A common challenge was to isolate tests. Best practices suggested mocking in unit tests and a separate testing database for integration tests. They preferred in-memory databases for this purpose. A well-designed source code where the database access code is loosely coupled to other parts also played a crucial role in maintainability. Many struggled to keep tests in sync with database schema changes. Indeed, developers hardly get any support for this task.

Our study is exploratory by nature. More studies are needed to understand the factors affecting the maintainability of database-related test code. Understanding more from the practices of the developers and good, maintainable database test code [13, 104] is a promising direction. Alternatively, automated approaches could help in regular tasks of developers. Some approaches aim to identify the system fragments impacted by schema changes [90, 94]. Such methods could be extended to the testing context, *e.g.,* to maintain a mapping between schema elements and mocks.

In-memory database vs actual database vs mocking We have seen many arguments for and against mocking, in-memory databases, or the actual database. In our motivational study, we found that 19 out of the 72 projects (26%) used mocks: 17 had Mockito,[16] and 2 had EasyMock tests.[17] This low number surprised us, as mocking was *the* recommended approach for unit tests to decouple them from the database. This is in line with the findings of Trautsch and Grabowski [127], who observed only a small amount of unit tests in open source Python projects, especially with mocks. A potential explanation is that it is easier to set up an in-memory database and rely on integration tests; instead of bothering with the implementation of mocks, despite its advantages.

We also found positive examples when manually inspected top-starred projects from the motivational study.

For instance, MyBatis,[18] a popular project with 17k stars and 11.4k forks, had a 74% of its database access methods covered with tests. It is a persistence framework, hence the high coverage. They use mocking for unit testing[19] and a test database with test data in scripts for integration testing.[20] The test database is initialised

[16]https://site.mockito.org/
[17]https://easymock.org/
[18]https://github.com/mybatis/mybatis-3
[19]https://tinyurl.com/2dm37hv5
[20]https://tinyurl.com/39vkk8j2

before executing all tests, using the *@BeforeAll* annotation. Finally, each test ends with a rollback function to get back to the initial state of the database.[21]

As another example, AxonFramework,[22] a framework for building event-driven microsystems, has 2.6k stars and 699 forks. They use dependency injection in the Spring framework to mock the data source of tests.[23] Their tests flush the database before each test using the *@BeforeEach* annotation.[24]

In any respect, developers need help in the implementation of database-related tests. They use frameworks' features when available, but they would benefit from more automated support in this context. Researchers have already explored generating tests with mocks [16, 98]. Such tools' emergence and initial success (*e.g.,* EasyMock,[17] MockNeat[25]) encourages similar approaches.

Database support in testing frameworks In our motivational study, we excluded projects with failing tests. Many failures were due to misconfigured testing environments. The systems either (i) relied on an external database for their tests or (ii) used in-memory databases but did not set them up correctly. We observed related problems in our qualitative study: many developers struggled to configure frameworks with multiple database connections. Consequently, our most extensive category among the solutions was the testing environment. Almost half of the questions and a third of the tags were related to this category.

Testing frameworks could provide more support to developers with database-dedicated features. Especially if these are configurable from the build systems. Some frameworks already offer similar functionalities. For example, *Spring Test* has *JdbcTestUtils*,[26] a collection of JDBC-related functions. It also supports test fixtures and transactional tests. Rails and Django offer similar features.

Answers also mentioned dedicated tools to support databases in unit or integration tests. For example, DBUnit is a JUnit extension targeted at database-driven projects, *"an excellent way to avoid the myriad of problems that can occur when one test case corrupts the database and causes subsequent tests to fail or exacerbate the damage."*[15] PHPUnit's database extension has similar features.[27]

We observed that the most desired features pertained to the initial configuration of databases and the efficient recovery of the database state between successive tests. The high demand and many problems related to such features indicate that developers' needs remain unexplored in this field. Moreover, only a few answers tackled trending technologies such as clouds and virtualisation or docker containers. Such technologies could offer robust solutions, but they appear to be unexploited.

Further research is necessary to improve testing practices as far as database access is concerned. As an answer notes, *"the database is the bread and butter of most business"* (*SE104*).

[21]https://tinyurl.com/2p9ftswk
[22]https://github.com/AxonFramework/AxonFramework
[23]https://tinyurl.com/yckjv4h5
[24]https://tinyurl.com/5bs4nbzp
[25]https://github.com/nomemory/mockneat
[26]https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html
[27]https://phpunit.de/manual/6.5/en/database.html

3.6 Threats to Validity

In this section, we discuss threats to the validity of our motivational study and two research questions.

Construct validity In our motivational study, we rely on SQLInspect to identify the database access methods of projects, *i.e.,* methods involved in querying the database. As a static tool, it may miss some DB methods, particularly in the case of highly dynamic query construction.

For test coverage, we rely on JaCoCo, a state-of-the-art tool used in industry and academia [75]. It might miss execution paths, and its configuration can influence the coverage results (*e.g.,* missing classes from the classpath). To avoid this, we executed tests according to Maven standards and excluded projects with failing tests.

Internal validity In our qualitative analysis, the manual classification of Stack Exchange questions is exposed to subjectiveness. To mitigate this risk, two authors examined each post independently, and a third author resolved conflicts in the first study. In the second study, one author assessed a question first, and then a second author reviewed it. Already assigned tags could influence a reviewer. However, the statistics confirmed that reviewers did not simply accept the taggings but also removed or added new tags when needed. The questions had 2.51 tags on average in the first round and 3.25 after the reviews. The total number of tags has also increased 21% from 300 to 365.

External validity Our motivational study is exploratory by nature. It considers various types of projects in terms of their application domain, size, and intensity of DB interactions. They are, however, all from Libraries.io and limited to the Java programming language. Projects not considered in our study might lead to other results.

In our first qualitative study, we extracted questions from three different Stack Exchange sites, intending to reach a higher level of diversity. We selected higher-ranked questions which are likely to influence more developers. Similarly, we labelled only the top three answers in the second study. This might introduce a bias towards the posts we selected. In reality, developers might face even more diverse challenges when (not) testing database code.

3.7 Related Work

In this section, we overview the related work of our study. First, we present empirical studies inspiring our research. Then, we discuss approaches to support testing database applications and present studies mining Stack Overflow.

3.7.1 Empirical studies on software testing

Our research got motivated and inspired by more general studies analysing testing practices and maintainability issues.

In this context, Kochhar *et al.* [81] investigated the adoption of testing in open source projects. They studied more than 20 thousand projects and explored the correlation of test cases with project development characteristics, including project size, development team size, number of bugs, number of bug reporters, and the underlying programming languages.

Greiler *et al.* [66] conducted a qualitative study about testing practices of plug-in based applications. They interviewed 25 senior practitioners and surveyed more than 150 professionals. As an outcome, they provide an overview of testing practices. They identified obstacles limiting the adoption of automated tests and proposed recommendations and areas for future research.

Beller *et al.* [25] conducted a large-scale field study on testing practices, monitoring five months of activities from 416 software engineers. They observed, among others, that (i) developers rarely run tests in the IDE, (ii) test-driven development is not widely spread among the participants, and (iii) developers usually spend 25% of their time on testing.

Gonzalez *et al.* [63] analysed over 80k open-source projects. They found that only 17% of those projects included test cases, and 76% did not implement testing patterns that would ease maintainability.

Trautsch and Grabowski [127] analysed more than 70k revisions of 10 Python projects. They observed that most projects had minimal unit tests, resulting in poor test coverage. They also showed that developers tended to overestimate the coverage of their tests and that mocks did not significantly influence the number of unit tests.

Our qualitative analysis revealed that many Stack Exchange questions concerned mocking, a testing technique often used to isolate the component under test. Spadini *et al.* [120] empirically analysed the usage of mocking dependencies on testing. Their goal was to understand how and why developers used mocking. They explored four projects with 2,178 test dependencies and surveyed 105 developers. Their results indicate that mocking is often used on dependencies, complicating tests dependent on external resources.

Alsharif *et al.* [13] studied the understandability of auto-generated database tests. They argued that studies on creating database tests did not consider the human cost to understand such tests. They used five database test generators and asked participants to explain the results. The authors highlighted two main findings: (i) the values in *insert* statements affected understandability, and (ii) using *null* values with integrity constraints may confuse human subjects on the outcome of tests.

3.7.2 Support for testing database applications

Several researchers have proposed approaches to *support* testing database applications.

In this regard, Deng *et al.* [43] proposed a white-box testing approach for web applications. They extracted URLs from the application source code to create a path graph and generate test cases.

Ran *et al.* [102] proposed a similar framework for black-box testing of web applications. They used a directed graph of web page transitions and database interactions to generate test sequences and capture how the database gets updated with the test cases.

Marcozzi *et al.* [89] proposed an approach to symbolic execution of SQL statements integrated with the traditional symbolic execution of the application source code. Their approach handled interdependent interactions between the application and the database. They also presented a symbolic execution algorithm for a subset of Java and SQL, implemented as a testing tool for generating test cases.

Another important aspect of testing database applications is *specialised coverage* since standard coverage techniques appear unsuitable for preserving all database constraints.

In this sense, Kapfhammer and Soffa [77] presented a test coverage technique to monitor interactions with database elements. They employed instrumentation of the application and test cases to capture SQL statement usage. Then they collected database-aware coverage reports of a test suite. Their coverage results also considered database interactions from the test cases and the program methods. They used six database-centric applications as case studies and observed a testing time increase from 13% to 54% as a drawback.

Tuya *et al.* [128] presented an approach to measure SQL query coverage. They argued that SQL queries embedded in code are not considered for test design, although queries implement an important part of the business logic. Their approach identified test data requirements for SQL statements and expressed them as a set of predicate rules. They demonstrated it on an open-source ERP application as a case study.

3.7.3 Mining stack exchange discussions

We collected and classified questions in Stack Exchange sites through a multi-label approach, inspired by previous work in our field.

Vasilescu *et al.* [130] investigated relationships between StackOverflow questions/answers and GitHub commits. They argued that developers could find suitable technical solutions in StackOverflow, affecting their commit productivity on GitHub. Their study showed a positive correlation indicating that developers' activity on StackOverflow affected their commit activity on GitHub.

Finally, Gonzalez *et al.* [64] proposed a five-way classifier approach assigning multiple tags to StackOverflow questions. They used a dataset of over 3 million questions.

3.7.4 Summary

The analysis of related research shows that database access code is sufficiently different from regular code to warrant specialised approaches. Several research works proposed approaches to *support* testing database access code. Nevertheless, no research has investigated *how* developers test database access code *in practice*, the main *issues* they face in this context, and the *best practices* recommended by the developer community.

Our work tried to fill this gap by analysing how tests in open-source systems cover the database access code and investigating the challenges and best practices of testing database access code.

3.8 Conclusion

In this chapter we studied developers' **challenges and best practices in testing database access code**. In our first motivational study, we analysed 72 open-source Java projects and investigated how their tests cover database access code. We found that 46% of those projects did not test half of their database methods, and 33% of them did not test the database communication at all.

We then conducted **two qualitative studies**[28]. (i) First, we analysed 532 StackExchange questions about database code testing and identified 83 issues, classified in a taxonomy of 7 main categories. We found that developers mostly look for general best practices to test DB access code. Concerning technical issues, they ask mostly about DB handling, mocking, parallelisation, or framework/tool usage. (ii) Next, we examined the answers to the questions. We distinguished 363 best practices and organised them with 9 main categories in a taxonomy. Most of the tags and questions were related to the testing environment and proposed various tools or configurations. The second most significant category was about database management best practices for initialising and cleaning a test database. The remaining categories were code structure or design, concepts, performance, processes, test characteristics, test code, and mocking.

Through the identification of these **main difficulties and best practices** we propose several paths of future research in order to help developers in the writing of quality tests of database manipulation code and thus improve the general quality of data intensive systems' software.

In the following chapters of this thesis we propose a database access code generator that can integrate the identified recommendations and facilitate the writing of tests.

[28]All data, scripts, and detailed results of our study publicly are available in a replication package [57].

CHAPTER 4

HyDRa Polystore Modelling Language

Contents

This chapter presents the modelling language we developed to model hybrid polystores. We will go through all the main sections of the language, detailing the grammar and show applied examples. We finally illustrate its benefits.

It is an extension of the work [54] published and presented at the 40th International Conference on Conceptual Modeling (ER 2021).

4.1 Introduction

As we have seen in Chapter 2, approaches to NoSQL databases design, as well as hybrid polystores, suffer from shortcomings because they (1) are specific to particular types of databases, or (2) are approaches based on the abstraction of NoSQL models erasing their specificities, or (3) are approaches combining several types of databases but allowing limited control over mappings between the conceptual and the physical representation.

This chapter presents the HyDRa modelling language allowing the user to (1) conceptually model the domain of a multi database system, (2) model the physical schema of data on several types of databases, (3) finely control the mappings between the conceptual and physical levels, (4) represent overlapping data on several types of databases.

The remaining of this chapter illustrates the HyDRa language, based on an example polystore schema built on the *IMDB* dataset[1], a dataset about movies. Figure 4.1 shows the complete case study of our running example. At the top of the figure is the conceptual schema of this application domain represented using Entity Relationship (ER) model. It consists of *Movie*, *Actor* and *Director* entity types. Those entity types are linked together with many-to-many or one-to-many relationship types *play* and *direct*. Alongside the cardinalities are the names of the role played by the entity type in the relationships. For example, the *Movie* entity type plays the role of *directed_movie* in the relationship *direct*. Those role names are facultative in an entity relationship schema.

In the bottom are represented our illustrated polystore's four database backends, two document databases (*mymongo, mymongo2*), a key-value (*myredis*) and a relational (*mydb*) database.

In the middle is represented the data stored on those databases that reflect the conceptual schema. Data is illustrated using a custom graphic notation to conform to their respective data models.

- *actorCollection* is a collection of JSON-like documents with several nested levels of attributes.
- *movieKV* is a set of key value pair sharing the same key pattern (*movie:[ID]*, *i.e.,* a constant string component *movie:* and a dynamic component *ID* being the movie identifier), the value is composed of a hash structure[2]
- *directed & directorTable* are two classical relational tables. *directed* having three *foreign keys* represented.

Each part of the designed language is detailed in the rest of this chapter and illustrated using this running example.

4.2 Language General Structure

The HyDRa polystore model language is composed of four main parts, each having its specific purpose. The next sections describe the abstract syntax of the language, in Appendix B is the concrete grammar expressed in the Xtext [9] language. A valid HyDRa schema, depicted in Figure 4.2 at lines (4.1) and (4.2), must be composed of a **conceptual schema**, a number of **physical schemas**, a **mapping rules** section and finally **databases** section.

- **Conceptual schema** specifies the domain-specific data model of the complete system.

[1]https://www.imdb.com/interfaces/
[2]https://redis.io/docs/data-types/hashes/

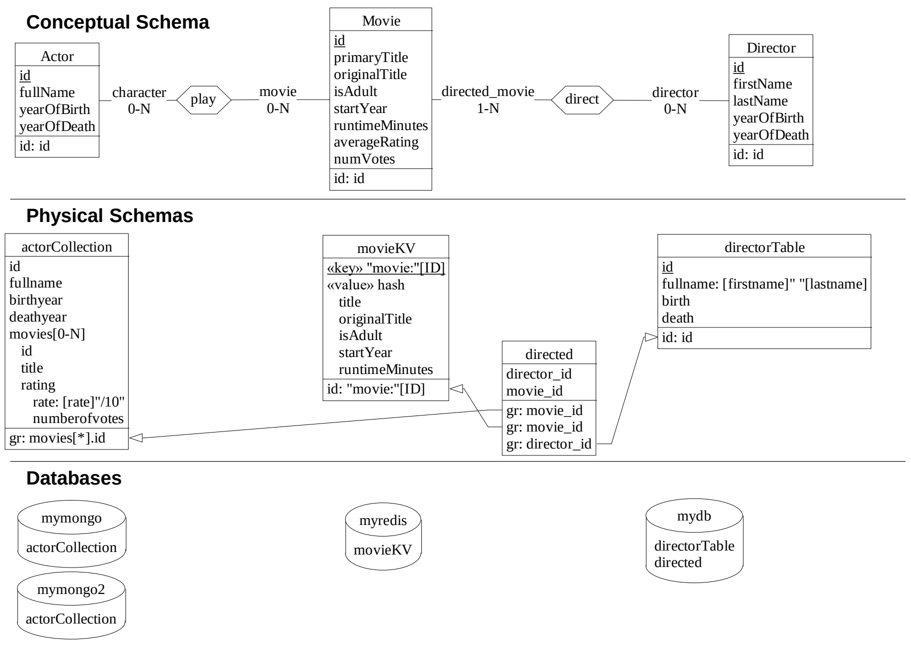


Figure 4.1: Schemas and databases of running IMDB example

- **Physical schema** describes the data structures in the underlying physical databases. It uses data model specific notation elements.
- **Mapping rules** is where the mappings between conceptual schema elements and physical schema elements are expressed.
- **Databases** declares the physical databases and their respective configurations.

In the remaining lines (4.3) (4.4) and (4.5) physical schema element is detailed. A physical schema represents a schema containing representation of the structure of data stored in a specific database of a specific data model. There are five types of data model supported and therefore five types of physical schemas available.

$$
\begin{aligned}
\langle\text{HyDRa schema}\rangle \quad &\models \quad \langle\text{ConceptualSchema}\rangle\langle\text{PhysicalSchema}\rangle^* && (4.1)\\
& \qquad \langle\text{Mapping Rules}\rangle\langle\text{Databases}\rangle && (4.2)\\
\langle\text{PhysicalSchema}\rangle \quad &\models \quad \langle\text{RelationalSchema}\rangle \mid \langle\text{DocumentSchema}\rangle \mid && (4.3)\\
& \qquad \langle\text{KeyValueSchema}\rangle \mid \langle\text{GraphSchema}\rangle \mid && (4.4)\\
& \qquad \langle\text{ColumnSchema}\rangle && (4.5)
\end{aligned}
$$

Figure 4.2: Abstract syntax of HyDRa language main components

4.3 Conceptual Schema

The **conceptual schema** represents the entities and the links between them that the polystore manipulates. As in standard database engineering methods, at the conceptual level, the user specifies the domain model [30] based on **Entity-Relationship** model constructions. The domain is described by means of *entity types, attributes, binary relationship types, conceptual identifiers, n-ary relationship types* or *relationship types with attributes*. Table 4.1 details the constructs of the *Entity-Relationship* model supported by the HyDRa conceptual language.

Entity Relationship Construction	Supported by HyDRa
Entity type	✓
Attributes	✓
Complex attributes	✗
Facultative attributes	✓
Binary relationship types	✓
N-ary relationship types	✗
Relationship with attributes	✓
Multi type roles	✗
Is-a relationship	✗

Table 4.1: Supported Entity Relationship model constructions by HyDRa conceptual language

The conceptual schema of our *IMDB* movie database example is illustrated in the top of Figure 4.1 and was described in Section 4.1. In Figure 4.3 is the equivalent model in the textual HyDRa modelling language. Entity types have attributes and declare one of several identifier(s) in the **identifier** block using specified attributes. Next we specify the relationship types, the cardinalities and mandatory role names played by the entity types within them.

4.4 Physical Databases

In the context of the development or reverse engineering of a polystore (section 1.3) several databases, possibly of different types, are involved. This is why our modelling language integrates a part allowing to declare the existence of these databases. Figure 4.4 shows the four databases, one relational (MariaDB), one key value (Redis) and two document databases (MongoDB) declared for our running example. The declaration of a database starts with its type that can be chosen between several technologies such as MariaDB, MySQL or SQLite for relational databases, Redis for key value, MongoDB for document, Cassandra or HBase for column based databases and Neo4j for graph data. Next is the specified name of database that will be referenced in the other parts of the HyDRa schema. Inside the block of a specific database we can find the server address (in our example three databases are run locally and there is a second MongoDB database $mymongo2$ on Line 16 running on an external

```
conceptual schema cs{
  entity type Actor {
    id : string,
    fullName : string,
    yearOfBirth : int,
    yearOfDeath : int
    identifier { id }
  }
  entity type Director {
    id : string,
    firstName : string,
    lastName : string,
    yearOfBirth : int,
    yearOfDeath : int
    identifier { id }
  }
```

```
  entity type Movie {
    id : string,
    primaryTitle : string,
    originalTitle : string,
    isAdult : bool,
    startYear : int,
    runtimeMinutes: int,
    averageRating : string,
    numVotes : int
    identifier { id }
  }
  relationship type direct{
    directed_movie[1-N]: Movie,
    director[0-N] : Director
  }
  relationship type play{
    character[0-N]: Actor,
    movie[0-N] : Movie
  }
}
```

Figure 4.3: Conceptual section of a HyDRa schema.

server) and the port, note that for relational databases we also find *dbname* which indicates the name of the database which is an additional information necessary for the correct establishment of the connection during the generation of the code.

These connections' information are read by the code generator to provide access classes with required connection string to the modelled data. Finally, they are used in the physical part of the language, allowing to link the concrete data structures to real databases. Moreover, duplication is allowed by mapping a physical schema to more than one declared databases. This can be particularly useful in a data loading use case, *e.g.,* to populate both production and test environment databases using a single application, which is a problem that has been identified as one of the obstacles to better test coverage in Chapter 3.

```
databases {
  mariadb mydb {
    host: "localhost"
    port: 3307
    dbname : "mydb"
  }
  redis myredis {
    host:"localhost"
    port:6379
  }
```

```
  mongodb mymongo{
    host : "localhost"
    port: 27100
  }
  mongodb mymongo2 {
    host : "hydra.unamurcs.be"
    port: 27000
  }
}
```

Figure 4.4: Databases declaration section

4.5 Physical Schemas

The **physical schema** section of our model allows the designer to specify how the data is actually persisted in native databases. We support the relational data model as well as the four most popular NoSQL data models [71], namely document, key-

$$\begin{aligned}
\langle\text{PhysicalStructure}\rangle &\models \langle\text{Table}\rangle \mid \langle\text{Collection}\rangle \mid \langle\text{TableColumn}\rangle \mid && (4.6)\\
&\ \langle\text{Node}\rangle \mid \langle\text{Edge}\rangle \mid \langle\text{KeyValuePair}\rangle && (4.7)\\
\langle\text{PhysicalField}\rangle &\models \langle\text{ShortField}\rangle \mid \langle\text{LongField}\rangle \mid \langle\text{ComplexField}\rangle && (4.8)\\
\langle\text{Reference}\rangle &\models \langle\text{PhysicalField}\rangle \rightarrow \langle\text{PhysicalField}\rangle && (4.9)\\
& && (4.10)
\end{aligned}$$

Figure 4.5: Abstract syntax of HyDRa physical structures

value, column wide and graph-based representations. One of the key advantage of the physical section is the ability to represent each design technique of each data model, by providing the designer with full control on physical data structures. Below, we illustrate how those common design strategies fit in HyDRa language and how the different physical data models are supported.

In our running example, the *physical schemas* are depicted in Figure 4.6. It represents three schemas based on three different data models. The first one, *movieRedis* (line 3) is a key-value schema stored on the key-value database *myredis* declared in the **physical databases** section (see Section 4.4), the second one is a schema based on the document database model, *IMDB_Mongo* will be stored on the two MongoDB databases, one local and the other on a remote server. This line allows, in a simple way, to duplicate the data of a schema (line 16). The third schema is a relational schema *myRelSchema* stored on the MariaDB *mydb* (line 35). The contents of each schema sections are described next.

In Figure 4.5 is specified the abstract grammar of a physical schema. As **Physical Schemas** may represent five different types of data models, we had to define common terms across them to refer to data structures, each data model specific data structures are grouped under the term **Physical Structure**. In their turn, those structures are composed of more detailed elements called **Physical Fields**, that can be of different types and complexity that we explain further next. Finally, a structure may contain a **Reference** towards another structure, allowing cross-database referencing.

Below we further define the chosen terms of *physical structures*, *physical fields* as well as *references*. A *physical structure* is an abstraction of technology-specific structures able to receive multiple data units. They contain multiple *physical fields*. Typical physical structures include **Table** in a relational database, **Collection** in a document database, and **Tablecolumn** in column oriented databases. For graph databases **Nodes** as well as **Edges** are considered as physical structures as each of them can declare fields. For key-value databases, we introduce the **KeyValuePair** concept. It reflects a set of key-value pairs sharing the same pattern of keys and values (see lines 3-14 of Figure 4.6).

A *physical field* is the term we use for data units in the corresponding technology-specific databases: **Columns** in relational, **Fields** in document, **Properties** in graph, **Columns** in column-oriented and **Value Properties** in key-value data models. If the field declared is composed of a single element we call it a **ShortField**. In our

```
1   physical schemas {
2
3     key value schema movieRedis :
         myredis {
4       kvpairs movieKV {
5         key : "movie:"[id],
6         value : hash{
7           title,
8           originalTitle,
9           isAdult,
10          startYear,
11          runtimeMinutes
12        }
13      }
14    }
15
16    document schema IMDB_Mongo :
         mymongo, mymongo2{
17
18      collection actorCollection {
19        fields {
20          id,
21          name:[fullname],
22          birthyear,
23          deathyear,
24          movies[0-N]{
25            id,
26            title,
27            rating[1]{
28              rate: [rate] "/10" ,
29              numberofvotes
30            }
31          }
32        }
33      }
34    }
35
```

```
35    relational schema myRelSchema :
         mydb {
36
37      table directorTable{
38        columns{
39          id,
40          fullname:
41            [firstname]" "[lastname],
42          birth,
43          death
44        }
45      }
46
47      table directed {
48        columns{
49          director_id,
50          movie_id
51        }
52
53        references {
54          directed_by : director_id
       -> directorTable.id
55          has_directed : movie_id ->
       movieRedis.movieKV.id
56          movie_info : movie_id ->
       IMDB_Mongo.actorCollection.
       movies.id
57        }
58      }
59    }
60  }
```

Figure 4.6: Example of a HyDRa schema physical section.

language we allow the declaration of fields which are composition of multiple simple elements, being **ShortField**s or string component. Those are called **LongField** and are detailed further in section 4.7. For NoSQL complex, potentially multiply nested, data types such as arrays or objects fields, we use a different word by calling them **ComplexField**s. This object type is recursive and it can contain again the aforementioned type of physical fields.

A **Reference** block expresses a link between two physical fields. In a polystore, a source field can reference a target field declared in a different database, and relying on a different data model. In other words, HyDRa offers the possibility to express cross-references between heterogeneous databases.

For instance, lines 53-57 in Figure 4.6 declare three references. Reference **directed_by** indicates that physical field *director_id* values are also stored in the *id* field of *directedTable*. This reference is the expression of a foreign key of the many-to-many table *directed*. The other column of this join table, *movie_id*, is a multiple hybrid reference, as *has_directed* targets *id* in the *movieRedis* key-value database and *movie_info* targets document database *IMDB_Mongo*.

4.5.1 Relational schemas

Relational schemas are composed of tables and columns where columns may only contain simple values as established by the relational data model. The abstract syntax depicted in Figure 4.7 describes the syntax of a table declaration in HyDRa. The declaration starts with *table* keyword followed by a name identifier. *columns* keyword comes before the columns' declaration which may be **ShortField** or **LongField** (as described in Section 4.5). Then it can possibly be followed by the definition of references. Source fields of the declared references are part of the current relational schema, while target fields may belong to a different physical structure.

$$\langle\text{Table}\rangle \models \texttt{table ID columns \{ } \langle\text{ShortField}\rangle\texttt{*}\ \texttt{|}\ \langle\text{LongField}\rangle\texttt{*\}}$$
$$\langle\text{Reference}\rangle\texttt{*}$$

Figure 4.7: Syntax of relational schema table structure

In our running example at lines 35-59 of Figure 4.6 we declare the relational schema structures. For instance Table *directed* (line 47) is a join table containing foreign keys declared in the reference block 53-57. *director_id* is a classical foreign key to another table *directorTable* of the same database (line 54). While the *movie_id* attribute is a foreign key to two structures, the *movieRedis.movieKV* key value database (line 55) and the *IMDB_mongo* document database (line 56).

4.5.2 Document schemas

Document schemas follow a JSON-like data model. It consists of pairs of field names and values organized by documents, each document field may in turn be a document, allowing embedded structures at any levels of depth. The syntax for document structures in HyDRa is shown in Figure 4.8, we start by mentioning the keyword *collection* followed by its identifier, then we declare the different fields, of any type (**ShortField**, **LongField** or **ComplexField**) after the keyword *fields*. This physical structure can also host explicit references (**Reference** blocks). Embedding data structures, referred to as *one-to-few*, as we describe in the corresponding data model section (Section 1.1.4) is illustrated in the running examples at Lines 24-31. The nested structure is represented using a **ComplexField** named *movies* as an array of movie objects. If, for design purposes or for technical reasons, embedding documents is not possible, the user can choose to use referencing values across collections of documents, using **References** blocks.

4.5.3 Key-Value schemas

Key-value schemas simply consist of key-value pairs, with no constructs allowing explicit references between data instances. Keys are made of string characters while values may be of different data types such as lists, sets, hash, *etc.* (see the Redis

$$\langle\text{Collection}\rangle \models \texttt{collection}\,\texttt{ID}\,\texttt{fields}\ \{\ \langle\text{PhysicalField}\rangle^*\}$$
$$\langle\text{Reference}\rangle^*$$

Figure 4.8: Syntax of a document schema collection structure

documentation [3] for a complete list). However HyDRa currently supports *lists* and *hashes*. As this database does not contain a pre-built physical structure of its own, we have created our own concept, named **KvPair**, which allows us to gather key-values that share the same key pattern. This structure is the equivalent, a **PhysicalStructure** of tables or collections in previous data models, so a key-value schema consists of a set of **KvPair** structures. Figure 4.9 shows the syntax **KvPair** in HyDRa. It consists first of the declaration of the identifier of the structure after the keyword *kvpair*. Then the *key* keyword indicates the specification of the key pattern, it can be a series of constant string and variables (**ShortField**). Afterward comes the value declaration which are **PhysicalField**. Finally, it is also possible to explicitly declare references between structures using the **Reference** declaration block.

The two main design patterns are described in Section 1.1.3. The first one, called *key value per field*, creates a key-value pair for each atomic field. The key is composed of different elements identifying a particular atomic instance. Examples of key patterns for this design includes **ENTITY:[identifier]:FIELD**. It results in data such as *MOVIE:tt0118715:TITLE* as key, and a binary object *The Big Lebowski* as value. The second design type, *key value per object*, uses complex data types instead of simple atomic values, this allows the grouping of multiple fields under the same key. Lines 3-14 in our running example illustrate this pattern.

$$\langle\text{KeyValuePair}\rangle \models \texttt{kvpair}\,\texttt{ID}\,\{\ \texttt{key}\ :\ \ (\texttt{string} \mid \langle\text{ShortField}\rangle)^*$$
$$\texttt{value}\ :\ \ \langle\text{PhysicalField}\rangle^*$$
$$\langle\text{Reference}\rangle^*\}$$

Figure 4.9: Syntax of key value schema kvpair structure

4.5.4 Column-oriented schemas

Column-oriented schemas are based on a model similar to the relational model in that they use tables and columns. However, here there is the possibility of using complex-valued columns, *i.e.*, multivalued columns or columns containing objects. Two different syntaxes exist depending on whether modelling a Cassandra or a

[3]https://redis.io/docs/data-types/

HBase database. Indeed, these two models have a slightly different functioning. The concept of *column family* is obsolete and is no longer used in Cassandra, which means that the syntax is very similar to relational tables. Only HBase keeps the concept of column family as a central element of its modelling, its syntax is depicted in Figure 4.10. HBase schemas rely on row identifiers (*rowkey*), and each row is composed of groups of key-value pairs (*column families*). Design methods identified in [2,3], such as *row per object representation* or *single cell per object* (see Section 1.1.5) are also supported in the HyDRa language. As this model is not represented in our running example the Listing 4.1 illustrates this modelling pattern.

$$\begin{array}{rcl} \langle\text{TableColumn}\rangle & \models & \texttt{table ID rowkey \{ } \langle\text{LongField}\rangle \mid \langle\text{ShortField}\rangle\texttt{\}} \\ & & \langle\text{ColumnFamily}\rangle^{*}\langle\text{Reference}\rangle^{*}\texttt{\}} \\ \langle\text{ColumnFamily}\rangle & \models & \texttt{ID}\langle\text{ShortField}\rangle^{*} \end{array}$$

Figure 4.10: Syntax of column schema table column structure

```
column schema colSchema {

  table Client {
    rowkey{
      clientnumber
    }

    columnfamilies {
      personnal {
        name:[first]"_"[last]
        }
      address{
        street,
        number,
        zipcode
      }
    }
  }
}
```

Listing 4.1: Example of a physical schema of a column based data model

4.5.5 Graph schemas

Graph schemas represent data that are heavily linked together and where relationships plays a central role. The data model is composed of **Nodes** and **Edges** that may contain **Properties** (*i.e.,* **ShortField**). This is why graph data are also called *property graphs*. The common way to design graph databases is described by the leading technology of graph databases, namely Neo4j [97]. Nodes usually represent entities and relationships between data are expressed using edges. The syntax for graph schemas in HyDRa is illustrated in Figure 4.11. It states that such a schema consists of any number of nodes, edges and references. A node consists of the keyword *node*, an identifier, and simple fields of type **ShortField**. An edge also consists of an identifier and any **ShortField** in addition to the keyword *edge* and a line establishing the two nodes linked by the current edge. *ID* is in this case the name of a previously specified node identifier.

$$
\begin{array}{rcl}
\langle\text{GraphSchema}\rangle & \models & \langle\text{Node}\rangle^*\langle\text{Edge}\rangle^*\langle\text{Reference}\rangle^* \\
\langle\text{Node}\rangle & \models & \texttt{NodeID}\ \langle\text{ShortField}\rangle^* \\
\langle\text{Edge}\rangle & \models & \texttt{Edge}\langle\text{ID}\rangle\texttt{->}\langle\text{ID}\rangle\ \langle\text{ShortField}\rangle^*
\end{array}
$$

Figure 4.11: Syntax of graph schema

Again, as the running example does not integrate graph databases, Listing 4.2 illustrates a design of a graph schema in HyDRa. In this representation there are two types of nodes, *Products* and *Orders* each with their respective attributes. These node types can be linked via *CONTAINS* edges, which also contain an attribute value *quantity*.

```
graph schema myGraphSchema {
  Node Product {
    product_id,
    Name,
    Description
  }

  Node Order {
    orderid
  }

  Edge CONTAINS {
    Order -> Product,
    quantity
  }
}
```

Listing 4.2: Example of a graph schema

4.6 Mapping Rules

The mapping rules section of an HyDRa polystore schema specifies links between the conceptual schema elements and the physical structures. Exploiting the possibly hybrid nature of those mapping rules, the designer can specify complex constructions such as **data structure split**, **data instance partitioning**, **data heterogeneity** and **data duplication** (see Section 4.7).

Figure 4.12 exposes the abstract syntax of mapping rules types and of their components. The left-hand side of the rule (before the arrow) is the **conceptual** component and the right-hand side corresponds to the **physical** component. Two types of mapping rules are supported: **entity mapping rules** and **role mapping rules**.

$$
\begin{array}{rcl}
\langle \text{EntityMappingRule} \rangle & \models & \langle \text{EntityType} \rangle \, (\langle \text{Attribute} \rangle +) \rightarrow \\
 & & \langle \text{PhysicalStructure} \rangle \, (\langle \text{PhysicalField} + \rangle) \\
\langle \text{RoleMappingRule} \rangle & \models & \langle \text{Role} \rangle \rightarrow \langle \text{Reference} \rangle \mid \langle \text{ComplexField} \rangle
\end{array}
$$

Figure 4.12: Abstract syntax of mapping rules

EntityMappingRule is a type of rule used to map **conceptual entity types** to **physical structures**. A conceptual entity type can be mapped to one or more heterogeneous physical structures. Mapping rules of our *IMDB* running example linking conceptual schema of Figure 4.3 and physical schema of Figure 4.6 are in Listing 4.3. First, at line 2, entity type *Actor* and its attributes are mapped to collection *actorCollection* in document database schema *IMDB_Mongo* (schema at lines 16-34). Second, at line 4, entity type *Director* is mapped to table *directorTable*, belonging to relational database schema *myRelSchema* (lines 35-59). Last, entity type *Movie* is mapped to three physical structures and one complex field (lines 8,9, 10, 11). Line 9 maps attributes *averageRating* and *numVotes* to physical fields contained into a third-level embedded structure *rating* in the *movies* array of *actorCollection*. Similarly, line 11 maps attributes of *Movie* to a key-value hash structure.

RoleMappingRule is another mapping rule type that maps **Roles** of **Relationship types** to **Reference** blocks or to **ComplexFields**. Line 3 maps the role *character* played by the entity type *Actor* of the relationship *play* to the physical complex field *movies* in the document schema. This field is an array type with cardinality 0-N, given that the other fields it contains are mapped to conceptual attributes of *Movie* (lines 8 and 9) and that the fields located at the same level as the array are linked to conceptual attributes of *Actor* (line 2), we can deduce that the array *movies* contains the movies in which the actor of the root document played. This is why we can map the role to the complex array field.

The lines 6 and 7 are role mappings on references and not on a complex object like the previous rule. The two references contained in the *directed* table, *has_directed* and *movie_info* (lines 54 and 55 in Figure 4.6) have been declared as foreign keys to the respective identifying fields of the physical structures mapped to *Movie* and *Director*. This explicit physical link expressed by the references is a reflection of an existing conceptual link represented by the *direct* relationship type between the *Movie* and *Director* conceptual entity types. The two role mapping rules make this link between physical references and conceptual relationship explicit.

Role mapping is important for the access code generator, as it provides various methods of manipulating data at a conceptual level both via the entity types and via the roles they play (we detail this in Chapter 5).

```
1   mapping rules{
2     cs.Actor(id,fullName,yearOfBirth,yearOfDeath) -> IMDB_Mongo.
        actorCollection(id,fullname,birthyear,deathyear),
3     cs.play.character-> IMDB_Mongo.actorCollection.movies(),
4     cs.Director(id,firstName,lastName, yearOfBirth,yearOfDeath) -> myRelSchema
        .directorTable(id,firstname,lastname,birth,death),
5     cs.direct.director -> myRelSchema.directed.directed_by,
6     cs.direct.directed_movie -> myRelSchema.directed.has_directed,
7     cs.direct.directed_movie -> myRelSchema.directed.movie_info,
8     cs.Movie(id, primaryTitle) -> IMDB_Mongo.actorCollection.movies(id,title),
9     cs.Movie(averageRating,numVotes) -> IMDB_Mongo.actorCollection.movies.
        rating(rate,numberofvotes),
10    cs.Movie(id) -> movieRedis.movieKV(id),
11    cs.Movie(primaryTitle,originalTitle,isAdult,startYear,runtimeMinutes) ->
        movieRedis.movieKV(title,originalTitle,isAdult,startYear,runtimeMinutes
        )
12  }
```

Listing 4.3: Example of a HyDRa schema, mapping rules section.

4.7 Benefits of HyDRa Modelling Language

Data duplication & heterogeneity HyDRa allows data duplication at the level of conceptual objects as well as at the physical schema level. Data duplication at the level of entity types can be expressed through multiple entity mapping rules, with the same entity type as left-hand side, but mapping it to several physical structures. An example was given above with the mappings of attribute *primaryTitle* of *Movie* entity type (lines 8 and 11) which is mapped to both a document database and a key-value database. HyDRa also allows one to duplicate an entire physical schema into several databases. For instance, line 16 in Figure 4.6 declares that physical schema *IMDB_Mongo* is stored in both databases *mymongo* and *mymongo2*.

Long fields The physical fields of an HyDRa schema may be a composition of several elements, namely mapped conceptual attributes and constant element. We refer so far to those field as **LongFields**. This is made possible by means of complex physical field declarations and related mapping rules. For instance, line 41 of Figure 4.6 specifies that the value of column *fullname* in relational table *directorTable* results from the concatenation of conceptual attributes *firstname* and *lastname*. This is expressed using the entity mapping rule at line 4. As another example, physical field *rate* at line 28 concatenates the *rate* conceptual attribute value with the "/10" string constant.

Data structure split Conceptual entity types can be split and stored in multiple and heterogeneous databases. Multiple entity mapping rules can be expressed for distinct fragments of a single entity type, e.g., by splitting its attributes into multiple and possibly heterogeneous databases. For instance, conceptual entity type *Movie* is composed of eight attributes, but those attributes are stored either in the *IMDB_Mongo* schema or in *movieRedis* schema, or in both physical schemas. As expressed by the mapping rules of lines 8, 9, 10 and 11, some movie attributes are subject to data duplication across several physical schemas, while attributes *isAdult, startYear, runtimeMinutes* are only present in the *movieRedis* schema.

Data instance partitioning Using data instance partitioning, an HyDRa polystore schema can map only a *subset* of the data instances of a given entity type to a particular physical structure. The data instances are discriminated based on user-defined conditions on the value of a particular entity type attribute. For instance, in Listing 4.4, a mapping rule expresses that the instances of entity type *Movie* that have an *averageRating* value greater than 9 must be stored in the *topMovies* physical structure.

```
	cs.Movie(id,primaryTitle,averageRating,numVotes) -(averageRating > 9)->
	 IMDB_Mongo.topMovies(id,title,rate,numberofvotes),
```

Listing 4.4: Mapping rule data instance partitioning

4.8 Conclusion

In this chapter we have detailed the HyDRa (Hybrid Data Representation and Access) modelling language that we have developed to **enable the modelling of multiple and possibly hybrid database systems**. The language is composed of four main parts, each with its own distinct prerogatives. Firstly, it is possible to establish a textual **conceptual schema** of the application domain using a subset of the Entity Relationship model. Secondly, the user specifies the concrete databases of which the polystore is made up, by giving their connection information. Then it is a question of specifying the format of the data structures hosted in each of these databases. To this purpose, **physical modelling** is possible for the *relational, document, key-value, column-oriented* and *graphs* data models. Each of these models having different terminologies and specific constructions, they have been grouped under concepts specific to HyDRa, such as **Physical Structures** or **Physical Fields**. The explicit declaration of foreign keys between several databases of different types is also a specificity of the declaration of physical schemas in HyDRa. And finally the last part of the modelling language consists of the establishment of the **rules of correspondence** between the two defined schemas, the conceptual and the physical. The mapping rules section allows mapping entity types as well as their roles played in relationships to references or complex objects in the physical schema.

This language allows several advantages among which are the explicit modelling of links between heterogeneous databases, the duplication of data, the flexibility of physical modelling with regard to the conceptual schema. In the following chapter we will see that this model brings other benefits, *i.e.,* in terms of data manipulation, via the implementation of a code generator based on the HyDRa modelling language.

CHAPTER 5

HyDRa Polystore Data Manipulation API

Contents

In this chapter we present the generated conceptual data access library of the HyDRa framework. We will go through why we chose this approach, what is actually produced, and how it is build. Finally, we summarise the advantages of such generated library.

This work has been published [55] at the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022) and has been invited to publish in Original Software Publications track in Science of Computer Programming journal (accepted, publication in 2023).

5.1 Introduction

When a new or existing polystore system has been modelled using the HyDRa modelling language presented in the previous chapter, the developer is still confronted with the data access problems identified in Section 1.2. Data manipulation in a polystore system is complex given the possible duplication of data across databases, as well as the heterogeneity of data models. Moreover, this data manipulation code, as described in Chapter 3, is not well tested and therefore more subject to bugs. It is to mitigate those data manipulation and evolution challenges that in addition to polystore data modelling, HyDRa also supports conceptual API code generation. It provides developers with a ready-to-use library, allowing polystore data manipulation operations on the conceptual level.

Before detailing this API in this chapter we will look back at the overview of the complete HyDRa framework through Figure 5.1.

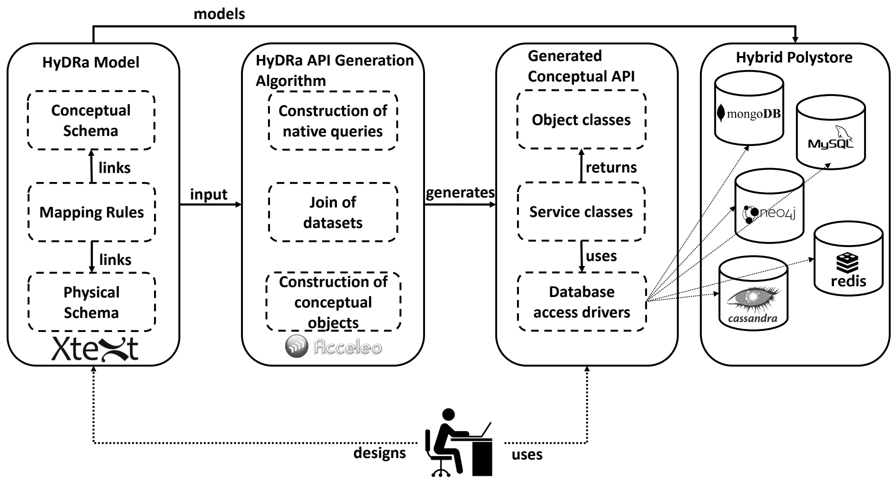


Figure 5.1: HyDRa framework

HyDRa Model HyDRa provides a textual modelling language, detailed in Chapter 4, to specify (1) the conceptual schema of the polystore, expressed in the **Entity-Relationship** model; (2) the physical schemas of each of its databases (NoSQL or relational), specifying data structures and their fields; and (3) a set of mapping rules to express possibly complex correspondences between the conceptual elements and the physical databases. Mapping rules enable possibly complex design choices, such as **data structure split**, **data instance partitioning**, **data heterogeneity** and **data duplication**.

HyDRa API Generation HyDRa relies on the automatic generation of a conceptual data manipulation API, derived from the polystore schema. This generation algorithm must read the model and produce classes capable of (1) **building native queries** for each of the supported database types and according to the physical mappings mentioned in the mapping rules, (2) **performing joins** of the collected data in case several databases are involved in a conceptual query, and (3) **reconstructing conceptual objects** based on the native database results.

Generated Conceptual API The generated API provides developers with a **ready-to-use library**, allowing polystore data manipulation operations on the conceptual level. The code generated contains (1) object classes which represent of the declared conceptual elements in the schema, (2) service classes which provide methods for manipulating the data objects and (3) classes responsible for handling the different database connections via native access drivers. Using the API to manipulate polystore data enforces the data heterogeneity, duplication and overlapping constraints declared in the mapping rules.

Hybrid Polystore The set of databases of the considered system which may be of different data models and also have data with implicit link between them.

The user He writes and designs the HyDRa model, generates the API using the code generator and then uses it in his own application code.

The rest of the chapter motivates the interest of a conceptual API with respect to the different physical variations possible in a polystore system. Then a generic specification and an example of the library produced is described. A description of some code generator algorithms follows with use cases illustrating them. Some technical considerations and a summary of the advantages of the generation approach finally conclude this chapter.

5.2 Conceptual Schema and Physical Schema Correspondences

To illustrate the functionalities of the generated API, we will first review a particular specificity that appears when modelling systems with multiple databases, moreover with NoSQL models. For the same conceptual schema it is possible to have several physical representations according to the user's needs, via the use of several databases, (*e.g., Products* are stored in a document database while *Orders* data are stored in a relational database as described in Section 1.3).

Tables 5.1 & 5.2 illustrate the variety of physical structures that can be chosen to implement conceptual constructs. All those physical structures can be expressed in HyDRa, via a combination of mapping rules. The databases depicted in *physical schema* column may be either of relational or NoSQL types.

Row **CS (1)** represents the design construct of **entity split** in multiple databases; an entity type can be entirely stored in a single database (a), or its attributes may be split among two databases (b). **Entity data duplication** is also possible, by declaring the same attribute in more than one database.

Row **CS (2)** shows **foreign keys**, **cross-database foreign keys** or **embedding structures** design choices. A one-to-many relationship type can be stored in a single database (a) and (d), or in different databases (b) and (c). In a single database, one can use an inner database foreign key reference (a), or an embedded structure (d). Physical schemas (b) and (c) implement **cross-databases** references, which can be **hybrid** *i.e.,* when the databases are of different types. In (b), a **mono-valued** foreign key is established between *E1* and *E2*, *i.e.,* the value in *E1.e2* references a particular

E2.id. In (c), a **multivalued** foreign key is established, in the opposite direction, *i.e.,* each value in *E2.E1* list references a particular *E1.id*.

Row **CS (3)** shows how a many-to-many relationship type (with attributes) can be physically represented in one (a), two (b) or three databases (c). A physical **join structure** *R* stores the relationship type attributes and references particular *E1* and *E2* instances (two foreign keys within *R*).

Table 5.1: Conceptual construction and physical correspondences

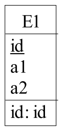
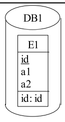
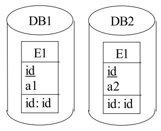
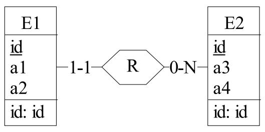
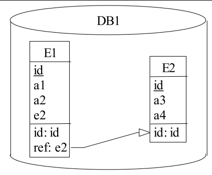
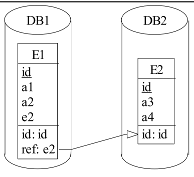
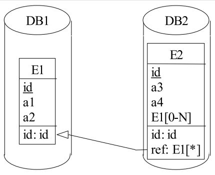
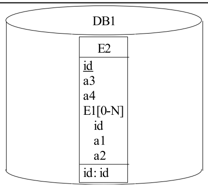
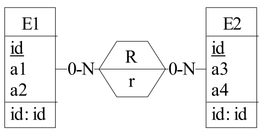
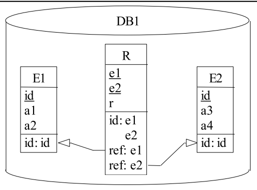
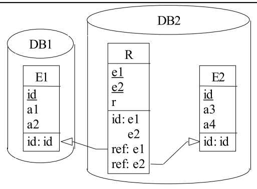
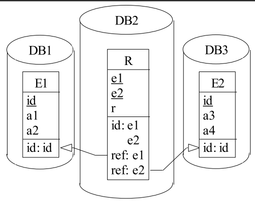
CS#	Conceptual Schema	Physical Schema
(1)		(a) 
		(b) 
(2)		(a) 
		(b) 
		(c)  DB1 E1 id a1 a2 id: id DB2 E2 id a3 a4 E1[0-N] id: id ref: E1[*]
		(d) 

Table 5.2: Conceptual construction and physical correspondences (continued)

CS#	Conceptual Schema	Physical Schema
(3)		 (a)
		 (b)
		 (c)

The different representations presented consider the databases as being of any type. The actual physical variations are therefore much wider if we consider each type individually for a database. So if each database can be either relational, key value or document oriented, the nine physical schemas give rise to hundreds of different possible polystores [1].

In this context, one can easily imagine the interest of manipulating data at a conceptual level, which makes it possible to abstract oneself from these variations of configurations. Indeed, a developer who would have to retrieve, for example, all the products linked to an order, would have to (1) know where the products and orders are stored, (2) write code specific to the databases encountered for these data and (3) write join code at the application level. This is a complex task and can introduce multiple bugs at each of these code writing steps. Moreover, what would happen to this code if, during a change in requirements, the products had to be migrated to a different type of database? Each piece of code linked to the modification must be rewritten, introducing new possibilities of errors. This is why we propose, via an automatic code generation tool based on the HyDRa model, a set of conceptual methods allowing to manipulate the data at a higher level of abstraction allowing the developers to be transparent with respect to the underlying databases.

[1]The implemented API generation currently supports 3 data models, namely key-value, document and relational ones.

5.3 Specification of Generated Conceptual Classes and Methods

In this section we will detail the conceptual elements, classes and methods, generated as well as the correspondence with the conceptual schema given as input. Below, we further describe the object classes, service classes and access methods that are generated from the HyDRa polystore schema. Let us define the following sets, composed of the available conceptual objects :

$$\mathit{Entities} = \{E_1, ..., E_n\} :: E_i(a_{i1}, ..., a_{ix}), x > 0 \tag{5.1}$$

$$\mathit{Relations} = \{R_1, ..., R_m\} :: R_i(r_{i1} : E_{i1}, ..., r_{iy} : E_{iy}, a_{i1}, ..., a_{iz}), y > 1, z \geq 0 \tag{5.2}$$

Entities is the set of all entity types, each entity E_i has a set of conceptual attributes a_i. **Relations** is the set of all relationship types, each composed of a list of at least two or more *entity types* E. Each component entity type plays a role named r, with a minimum and a maximum cardinality. A relationship type may contain attributes.

5.3.1 Object classes

Object classes are classes that describe a conceptual element of the schema, an entity type or a relationship type. They wrap data conceptually and the application therefore uses them to perform manipulation operations transparently to the actual stores. All manipulation functions generated by the API take as input or return objects of this type. Listings 5.1 and 5.2 respectively formalizes the generation of object classes for entity and relationship types. First, $\forall E \in \mathit{Entities}$ and $\forall R \in \mathit{Relations}$, we generate an object class E and an object class R. An object class contains attributes declarations. There are two types of attributes, simple or class attributes. Simple attributes are declared for each conceptual attribute of the entity or relationship type, with its data type t_i (e.g., int, float, date, ...). If E is involved in a binary R relationship type, *i.e.,* $R(r : E, r' : E')$, then class attributes of E are declared as E' objects. Depending on the cardinality of role r (*i.e.,* [1-1] or [0-N]), we declare either a simple or a list attribute of type E' in E. If E is involved in an n-ary R relationship type, *i.e.,* $R(E, E_1, E_2, ...)$, entity E declares an attribute of type R instead.

```
$\forall \mathbf{E}(\mathbf{a_1} : \mathbf{t_1}, ..., \mathbf{a_n} : \mathbf{t_n}) \in \mathbf{Entities}, \mathbf{n} \geq \mathbf{1}$
Class $E$ {
    $\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} \leq \mathbf{n}$
    $t_i\, a_i$ ;

    $\forall \mathbf{R}(\mathbf{r} : \mathbf{E}, \mathbf{r'} : \mathbf{E'}) \in \mathbf{Relations}$
    ($E' \vee E'$[]) $r'$; // array type depending on cardinality of $r$

    $\forall \mathbf{R}(\mathbf{r} : \mathbf{E}, \mathbf{r_1} : \mathbf{E_1}, ..., \mathbf{r_x} : \mathbf{E_x}, \mathbf{a_1} : \mathbf{t_1}, ..., \mathbf{a_y} : \mathbf{t_y}) \in \mathbf{Relations}, (\mathbf{x} \geq \mathbf{2})\mathbf{or}(\mathbf{y} \geq \mathbf{0})$
    ($R \vee R[]$) $R$;
}
```

Listing 5.1: Entity object classes generation

```
    $\forall \mathbf{R}(\mathbf{r} : \mathbf{E}, \mathbf{r_1} : \mathbf{E_1}, ..., \mathbf{r_n} : \mathbf{E_n}, \mathbf{a_1} : \mathbf{t_1}, ..., \mathbf{a_m} : \mathbf{t_m}) \in \mathbf{Relations}, \mathbf{n} \geq \mathbf{2}, \mathbf{m} \geq \mathbf{0}$
    Class $R$ {
        $\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} \leq \mathbf{m}$
        $t_i\, a_i$ ;

        $\forall \mathbf{i}, \mathbf{0} \le \mathbf{i} \le \mathbf{n}$
        $E_i r_i$ ;
        }

```

Listing 5.2: Relationship object classes generation

5.3.2 Data manipulation classes

Data manipulation classes or service classes are in charge of providing data manipulation methods, including selection, update, insertion and deletion for a particular E or R. $\forall E \in Entities$ and $\forall R \in Relations$, we generate a service class, respectively *EService* and *RService*. Listing 5.3 specifies those methods, by providing the generated signatures. Actual example instances are provided in the next section. Multiple variants of selection methods are specified. An entity can be retrieved based on a particular attribute value of the entity either by (1) calling specific methods on this attribute and giving the attribute value (line 4) or (2) by giving a condition expression object that can represent simple or multiple *and* or *or* conditions (line 5).

Moreover, it is possible to (3) retrieve entities by exploiting the declared conceptual relations in which this entity type is concerned. In the lines 8 and 9 are expressed the methods selecting the entities E bearing the role name r_0 involved in a relation R. We can call these functions by giving as argument either a set of conditions, expressed on each of the roles of the relation, or by giving directly an object class instance of the other types of entities of the relation.

The lines 11 up to 13 add a possible condition in case the concerned relation contains relation attributes. Similarly, insert, update and delete methods are offered.

```
1   $\forall \mathbf{E}(\mathbf{a_1} : \mathbf{t_1}, ..., \mathbf{a_n} : \mathbf{t_n}) \in$ Entities, n ≥ 1
2   Class EService{
3       $\forall \mathbf{i}, \mathbf{0} \le \mathbf{i} \le \mathbf{n}$
4       E[] get$E$ListBy$a_i$($t_i a_i$);
5       E[] get$E$List(Condition <$E$> condition);
6
7       $\forall \mathbf{R}(\mathbf{r_0} : \mathbf{E}, \mathbf{r_1} : \mathbf{E_1}, ..., \mathbf{r_x} : \mathbf{E_x}) \in$ Relations
8       E[] get$r$0ListIn$R$([Condition <$E$> $r_0$], [Condition <$E_1$> $r_1$],..., [Condition
         <$E_x$> $r_x$]);
9       E[] get$r$0ListIn$R$([Condition <$E$> $r_0$], $E_1$ $e_1$,..., $E_x$ $r_x$);
10
11      $\forall \mathbf{R}(\mathbf{r_0} : \mathbf{E}, \mathbf{r_1} : \mathbf{E_1}, ..., \mathbf{r_x} : \mathbf{E_x}, \mathbf{a_1} : \mathbf{t_1}, ..., \mathbf{a_y} : \mathbf{t_y}) \in$ Relations
12      E[] get$r$0ListIn$R$([Condition <$E$> $r_0$], [Condition <$E_1$> $r_1$],..., [Condition
         <$E_x$> $r_x$], [Condition <$R$> c]);
13      E[] get$r$0ListIn$R$([Condition <$E$> $r_0$], $E_1$ $e_1$,..., $E_x$ $r_x$, [Condition <$R$> c]);

14
15      void insert$E$($Ee$, [$E_1 e_1$,..., $E_x e_x$]); // $E_x$ are the other mandatory entity
         types of $r$ where $E$ is involved
16      void update$E$List(Condition <$E$> c, Set <$E$> set);
17      void delete$E$List(Condition <E> c);
18      }
```

Listing 5.3: Entity data manipulation classes generation

5.3.3 Generic illustration

Here we will give an example of the methods and classes generated for the conceptual schema CS (3) of Table 5.2, being the most complete and allowing to illustrate all previously specified rules. Figure 5.2 shows the object classes of entity types $E1$ and $E2$ with their respective a_x attributes. The attributes types are specified as T, which means that it will be replaced by the corresponding primitive conceptual type (such as int, float, string, etc.). Moreover, these two entity types being linked together via a relation R with attribute and cardinalities [0-N] they also have an attribute of type *List* containing objects of type R. On the right side of the figure is the object class of the conceptual relationship R, declaring two object attributes of the involved entity types. Those attributes are named with their respective role names. The R class also contain the relationship attribute r.

```
Class E1 {
     T id ;
     T a1 ;
     T a2 ;

     List<R> RListAsRoleE1;
     // Getters and setters, constructors,...
}
```

```
Class E2 {
     T id ;
     T a3 ;
     T a4 ;

     List<R> RListAsRoleE2;
     //...
}

```

```
Class R {
     E1 roleE1 ;
     E2 roleE2 ;
     T r;
     //...
}
```

Figure 5.2: Object classes generated for CS 3.

Next are illustrated the service classes of these two respective classes in Figure 5.3, they follow the specification established in Listing 5.3. First, an entity type can be retrieved by (1) giving an attribute value, *e.g., List<E1> getE1ListByA1(T a1)* or (2) using the generic method which takes as input an object representing a condition (which are described further in Section 5.4.2) *e.g.,* (*List<E1> getE1List(Condition<E1> condition)*).

Then follow the access methods exploiting the R relation with the different choices of arguments, *i.e.,* Condition objects (*getRoleE1ListInR(Condition<E1> roleE1Condition,...)*) or instances of objects on the opposite role (*getRoleE1ListInRByRoleE2(E2 roleE2)*), respectively $E1$ or $E2$.

Listing 5.4 details the service of relationship type R. Selection methods return R objects, depicted on the right side of Figure 5.2 containing instances of linked entity types $E1$ and $E2$. Insertion and deletion methods on R instances also exist. Inserting a new R instance consists in connecting two existing $E1$ and $E2$ instances via R. Deleting a R instance consists in disconnecting (*i.e.,* removing their reference attribute) those instances, without deleting them.

```
class E1Service {
    List<E1> getE1List();
    List<E1> getE1List(Condition<E1
     > condition);
    E1 getE1ById(T id);
    List<E1> getE1ListByA1(T a1);
    List<E1> getE1ListByA2(T a2);
    List<E1> getRoleE1ListInR(
     Condition<E1> roleE1Condition,
      Condition<E2> roleE2Condition
     , Condition<R> R_condition);
    List<E1>
     getRoleE1ListInRByRoleE2(E2
     roleE2);

    void insertE1(E1 e1);
    void updateE1List(Condition<E1>
      roleE1Condition, SetClause<E1
     > set);
    void deleteE1List(Condition<E1>
      condition);
}
```

```
class E2Service {
    List<E2> getE2List();
    List<E2> getE2List(Condition<E2
     > condition);
    E2 getE2ById(T id);
    List<E2> getE2ListByA3(T a3);
    List<E2> getE2ListByA4(T a4);
    List<E2> getRoleE2ListInR(
     Condition<E2> roleE2Condition,
      Condition<E1> roleE1Condition
     , Condition<R> R_condition);
    List<E2>
     getRoleE2ListInRByRoleE2(E2
     roleE2);

    void insertE2(E2 e2);
    void updateE2List(Condition<E2>
      roleE2Condition, SetClause<E2
     > set);
    void deleteE2List(Condition<E2>
      condition);
}
```

Figure 5.3: Entity service classes of CS 3.

```
class RService {
    List<R> getRList();
    List<R> getRList(Condition<E1> roleE1condition, Condition<E2>
     roleE2condition, Condition<R> Rcondition);

    void insertR(R r);

    void updateRList(Condition<E1> roleE1condition, Condition<E2>
     roleE2condition, Condition<R> rcondition, SetClause<R> set);
    void delete(Condition<E1> roleE1condition, Condition<E2> roleE2condition
     , Condition<R> rcondition, SetClause<R> set);
    //...
}
```

Listing 5.4: Relationship service class of CS 3.

5.4 Specification of Generated Algorithms

We have first described which classes and objects would be generated according to a HyDRa polystore schema as input. This section now details the implementation of these generated methods, *i.e.,* the important steps to be respected, whatever the final database that will be reached, in order to retrieve or insert data using conceptual objects. This challenge is increased by the additional specificities brought by the HyDRa modelling language (*e.g.,* **data duplication**, **heterogeneity** or **structure splitting**).

The challenges to be managed in the implementation code include:

- The identification of the databases involved in the conceptual objects being manipulated.

- The construction of multiple native queries, whether for selection, insertion, update or deletion.
- The join of datasets retrieved from different databases.
- The reconstruction of conceptual objects.
- The conceptual extraction of complex fields (**ComplexFields**) and fields built on patterns (**LongFields**).
- The identification of possible data conflicts.

Through the following sections we will detail some generated methods and specify their algorithms, which, depending on the final goal of the method, integrate one or the other of the above-mentioned challenges.

5.4.1 Database specific methods

First, we specify that in parallel to the methods located at the conceptual level of abstraction, such as *getEList* or *insertE* (Section 5.3.3), there are also methods generated at a lower level of abstraction, in charge of interacting with actual databases.

Indeed, the goal of conceptual methods is that the developer only needs to manipulate these high level abstraction methods in order to manipulate the data. Regardless of the mapping rules established in the HyDRa schema, if entity type *E* is stored in a document, relational or key-value database, or even if it is duplicated or separated on several databases, the conceptual methods offered to the user remain the same (*i.e.,* the signatures are identical) and are not impacted by the physical representation.

However, these generic methods must ultimately be able to reach the real databases containing the data of the entity type concerned. This is why the data manipulation classes, specified in Section 5.3.2, also contain methods that manipulate the conceptual data at the level of the physical structures indicated in the physical schema. These functions are responsible for performing the operation on a specific database where the conceptual element has been stored.

As an example, taking the CS (1) schema with the physical schema (b) of the Table 5.1, the entity type *E1* is distributed on two different databases, *DB1* and *DB2* in two physical structures having the same name as the entity type. This distribution means that in the generated *EService* class, in addition to the methods described above such as *insertE1* or *getE1List(), etc.* There will also be the presence of these database specific methods :

- *E1[] getE1InDB1*
- *E1[] getE1InDB2*
- *insertE1InDB1*
- *insertE1InDB2*

In the following specifications of the generated implementation algorithms, we refer to these methods using the notation *getEInDB*.

5.4.2 Condition classes

In order to perform filtered selections on entity types, the *Condition* class is generated and can be instantiated to represent a filter condition on one or several attributes, according to given values and operators. These *Condition* objects can then be given as arguments to conceptual data selection methods of type *getEList*.

A condition consists of three elements:

- **A conceptual attribute** existing for an entity type or a relation type. This attribute will be the one on which the condition will be checked.
- **An operator**. Designates the comparison operator to be performed, among which are *equal*, *smaller*, *larger*, *contains*, etc.
- **A value**. Contains the value to compare with the actual value of the object.

Listing 5.5 describes the *Condition* class. A condition can be simple, *i.e.,* containing the three elements mentioned or also be complex, *i.e.,* it is a condition composed of other conditions. Complex conditions are used to represent AND/OR operators. An OR or AND condition will be composed of two simple conditions, one left and one right, which themselves can be complex, this allows the construction of multiple chains of conditions. Finally, this *Condition* object can be evaluated on an instance of a conceptual object (instance of interface *IPojo*).

```
interface Condition<E> {

  <E> SimpleCondition<E> simple(E attribute, Operator op, Object value);

  <E> AndCondition<E> and(Condition<E> left, Condition<E> right);

  <E> OrCondition<E> or(Condition<E> left, Condition<E> right);

  boolean evaluate(IPojo obj);
}
```

Listing 5.5: Condition object interface

5.4.3 Selection methods

Here we describe the algorithms involved in the implementation of conceptual objects selection methods. First we start with conceptual level method implementation and then we go deeper and specify database specific methods.

Algorithm 1 illustrates the generic implementation code for a generated conceptual selection method (such as *getEList(Condition<E> condition)* or *getEListInR(...)*). Line 3 declares the final object list of *E* objects that will be returned. Line 2 is a list of lists of *E*, it is a temporary variable containing the results of each of the databases where data of *E* is stored. The first step of the algorithm (line 5) thus consists in filling this list by calling for each of the databases concerned by *E* (*db*) its specific generated method *i.e., getEInDB(condition)* (as in Section 5.4.1), returning a list of objects of type *E* satisfying the given condition. Then if indeed several databases had to be queried, then this implies that a join must be performed on these datasets in order to consolidate the results. This operation is represented by the call to *fullOuterJoin* at line 8, the implementation of this method can be of different degrees of complexity depending on the physical schema encountered as shown in Section 5.2.

Algorithm 1 Generic implementation of selection method of entity type *E*

```
1: $E \in Entities$
2: $List < List < E >> entitiesList = \{\}$
3: $List < E > finalResult = \{\}$
4: for all Database $db$ where $E$ attributes are stored do
5:     $entitiesList \leftarrow getEInDB([conditions]) \cup entitiesList$
6: end for
7: if $entitiesList.size() \geq 1$ then
8:     $finalResult \leftarrow fullOuterJoin(entitiesList)$
9: else
10:    $finalResult \leftarrow entitiesList[0]$
11: end if
12: return $finalResult$
```

Database specific implementation algorithm (*i.e., getEInDB(condition)*) is described by Algorithm 2. The necessary variables consist of (1) a list of objects of type *Row* and (2) the final list of objects of type *E*. The *Row* type is a class type that acts as a wrapper for the various specific object types that each of the database access drivers can return. In order to select a set of data from a database, it is necessary to query it in the language it understands, so the first step of our algorithm is to build a native query satisfying the given *condition.* This is represented by the call to a sub-method *generateNativeQuery* (line 4). It will return, according to the type of database, a query such as **SELECT ... FROM ... WHERE ...** (for relational databases), a **$match:...** (document databases), or a key pattern (key-value databases). Once this query is built, it is executed on the right database connection, and the results are converted into *Row* list, a custom wrapper object. However, these *Row* objects do not correspond to the conceptual object of type *E* expected as a return by the user. Therefore, the last step of the algorithm is to convert each *Row* object into an object of type *E*. Note that at this stage the *E* object will only contain values for the attributes concerned by the queried database. The complete construction of all attributes of the *E* object, if it has been separated, will take place in the *fullOuterJoin* step (line 8 of Algorithm 1).

5.4.4 Insertion of entity types

The implementation of the insertion methods is, like the selection methods, dependent on the different mappings performed at the physical level. This is to identify and build the right insertion query on the corresponding underlying database(s). The particular additional aspect brought by the insertion in a model integrating complex NoSQL structures (such as the modelling of nested structures *i.e.,* physical schema (d) of CS (2) in Table 5.1) is that the insertion can concern an entity that is at the lowest level of the nested structures. This means that the insertion results in an update query on existing data instead of a new complete object insertion. In

Algorithm 2 Specific database selection *getEInDB* implementation

```
1: $E \in Entities$
2: $List < Row > rowResults = \{\}$
3: $List < E > entitiesResult = \{\}$
4: $query \leftarrow generateNativeQuery(condition)$
5: $rowResults \leftarrow executeQuery(db, query)$
6: for all $row \in rowResults$ do
7:     $entitiesResult \leftarrow buildObject(E.class, row) \cup entitiesResult$
8: end for
9: return $entitiesResult$
```

order to be able to build these update requests, the data on the structure containing the nested data must also be available when calling the generic method *insertE(E e)*. This is why in the signatures of the insertion methods, as arguments, are all the entity types opposite to the linked mandatory roles (1-1 and 1-N cardinalities) of E. These signatures are generated using only the cardinalities of the conceptual relationships. It is therefore critical that the mapping rules between roles and physical structures are correctly written in the HyDRa schema.

Types of physical structures

In addition to the conceptual signature considerations of the insertion methods, it is also necessary to construct the correct query based on the mapped physical structures. By considering the different physical schemas proposed in Section 5.2 we can identify different types of physical structures having an impact on insertion implementation.

- Insert in **standalone structures**. Concerns the addition of data for an entity type in a physical structure not linked to another (Physical schema (a) and (b) of CS 1 in the table 5.1). The data of E in this structure are neither referenced nor contain mandatory references. This insertion will result in the translation of the E object and its attributes into a simple native insert query.
- Insert in **complex embedded structures**. Here we consider the addition of data in a nesting structure where (1) there is at least one element nested in an already nested structure (meaning at least a three-level structure), where (2) the first level contains an attribute of the E entity to be inserted, (3) the lower levels are mapped to mandatory roles. These structures require special implementation since they may involve multiple relationships and mandatory entity types outside the direct scope of E.
- Insert in **descending structures**. In this configuration we study the insertion in a nesting structure consisting of two levels. The first one contains attributes of E and the nested array is mapped to a mandatory role of type E_1. This means that the insertion must also insert at the same time one of the entity types passed as an argument to the insertion method.
- Insert in **ascending structures**. The ascending structure represents the fact that an entity E is stored in a nested attribute mapped to a role whose opposite

is mandatory. These structures mean an update of existing data by inserting E into the complex attribute.

- Insert in **reference structures**. Here we identify the structures concerned by reference attributes mapped to mandatory roles of E or to its opposites E_1 which have not been involved in the previous structures. These structures represent foreign key type constructs. In this category we also detect and process join structures involved in many-to-many relationships.

In order to illustrate the different insert structures as well as the influence of the conceptual roles on the signatures of these insert methods, we present an extended conceptual schema of Chapter's 4 schema representing an application domain on movies. In this schema we have added the entity types *Review* and *User*. A *User* writes reviews (0-N) via the *writes* relation type. These reviews are written by a user (1-1) and concern a single movie (1-1).

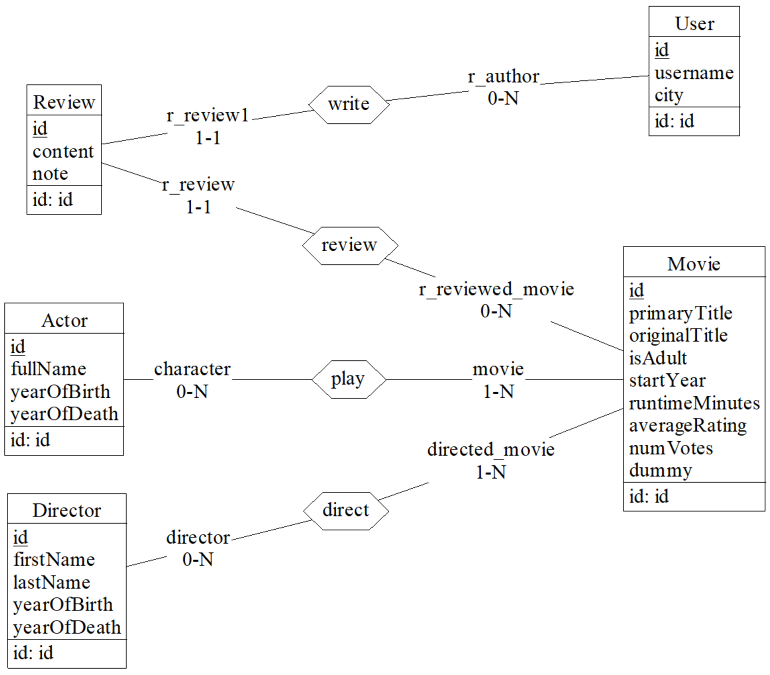


Figure 5.4: Extended IMDB example conceptual schema

The presence of these mandatory roles implies that a review cannot exist conceptually without its movie or user. This is why the code generator, according to the specification of Listing 5.3 will generate the following insertion method in *ReviewService* :

```
insertReview(Review review, Movie r_reviewed_movie, User r_author)
```

For movies, the mandatory roles being of maximum cardinality N, the arguments will be of type lists

```
insertMovie(Movie movie, List<Director> directorDirect, List<Actor>
    characterPlay).
```

Figure 5.5 represents a **complex embedded structure** *reviewCol* for the entity type *Review*, this structure is a collection in a document database which fulfils the conditions stated in Section 5.4.4 :

- There are top-level attributes mapped to *Review* (*idreview*, *content* and *note*)
- There is a complex attribute, *movie*, mapped to a mandatory role of *Review*, in this case it is mapped to *r_review* since it contains information about the movie review.
- Moreover, this complex attribute contains itself another complex attribute mapped to a role which is also mandatory, *actors* is an array mapped to the role *movie* (1-N) of the relation *play*. This array attribute contains the actors of the movie containing it.

The example of this *reviewCol* structure shows that when we want to insert a new review, in order to respect the conceptual constraints, it is also necessary to fill in information about the film and its actors. However, if we look at the signature generated for the insertion of a review, we notice that it does not contain any information about the actors. This is normal given that the entity type *Review* is not directly linked to the entity type *Actor* via any relationship type. The link between *Review* and *Actor* is indeed indirect, and goes through the *Movie* entity type first. This kind of link requires a special implementation to check that the passed *Movie* object contains the actor information in its attributes.

In Figure 5.6 is illustrated a **descending structure** for the entity type *Movie*, when inserting a movie, the complex array object *actors* being mapped to the mandatory conceptual role *movie* (1-N) will be constructed, populated and inserted along with the new movie into the collection *movieCol*.

At the same time, the insertion of a movie will impact the *actorCollection* structure, represented in Figure 5.7. This structure is indeed an **ascending structure** affected by *Movie* because the complex object *movies* is mapped to a role (*character*) whose opposite is mandatory (role *movie*). This means that although movies are optional in the *actorCollection*, the fact that a movie cannot exist without its actors from the conceptual point of view, it is possible that an already existing actor in the database has to be updated in order to mention the new movie in its *movies* array.

Insert in standalone structure algorithm

Algorithm 3 describes the generic algorithm for implementing an insertion of an instance of *E* into a standalone structure. *struct* designates the standalone structure in which it has been identified that there are attributes of *E* entity type. *db* designates the database in which this structure is stored, and *db.type* designates the type of this database. The main steps of this algorithm are :

(i) Identify the database type of the structure (Lines 3, 10, 17).

(ii) Identify the conceptual attributes of *E* that are mapped into this structure (Lines 4, 12), for structures *kvpairs* (Section 4.5.3), there is a distinction be-

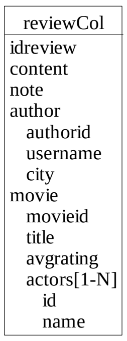


Figure 5.5: Double embedded structure

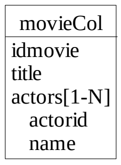


Figure 5.6: Desce structure

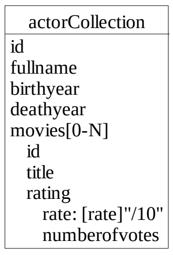


Figure 5.7: Ascending structure

tween attributes mapped into the key part (line 24), or into the value part (line 31).

(iii) For each of them the corresponding native structure with the name of the attribute and its value must be build.

(iv) Construct the final native query based on the constructed structure (Lines 8, 15, 35).

5.4.5 Join of datasets and data conflict identification

As described in Section 5.4.3, in order to retrieve the data concerning an entity type *E*, it is necessary to first perform a selection on all the databases, and particularly on each physical structure where at least one attribute of *E* is mapped. However, each of these structures does not necessarily contain all the attributes of the entity type, this is called *structure split* and is one of the advantages brought by the HyDRa modelling language. This is why, in Algorithm 1 at line 8, the result lists of each call to the specific databases are passed in a join function(*fullOuterJoin*). This function, starting from a list of several lists of partially filled *E* objects, will return a single list of complete *E* objects, *i.e.,* with all attributes having a filled value. Furthermore, this join function fulfils a second objective of identifying data value conflicts.

The join is a common operation that consists of comparing two subsets of objects with each other on the basis of a common attribute. If the two object instances of each of these sets share an identical value for this attribute, then their other attributes are combined to form a single object. There can be two types of joins, an *inner* or an *outer*, the difference lies in the set operator applied. Either it consists in the application of the **INTERSECTION** operator on the join attribute, or it is the application of a **UNION** between two subsets. Its implementation (described by Algorithm 4) relies on several input parameters. *datasets* is the set of subsets of *E* on which to perform the join. *joinMode* contains the type of join to perform, *inner* or *outer*.

Once we have identified two instances of *E* on two subsets of *datasets* which

Algorithm 3 *insertE(E entity)* in standalone structures algorithm

```
$struct$ is the structure containing mapped element of $E$
$db$ is the database of $struct$
if $db.type$ is RELATIONAL then
    for all $attr \in entity.attributes$ mapped to $struct$ do
        $columns \leftarrow attr.name$
        $values \leftarrow attr.value$
    end for
    $query \leftarrow$'INSERT INTO ($columns$) VALUES ($values$)'
end if
if $db.type$ is DOCUMENT then
    $doc$ is an empty Document
    for all $attr \in entity.attributes$ mapped to $struct$ do
        $doc.append(attr.name, attr.value)$
    end for
    $query \leftarrow setOnInsert(doc)$
end if
if $db.type$ is KEYVALUE then
    $key$ is the string containing the key
    $value$ is the value
    for all $component \in struct.keypattern$ do
        if $component$ is a string constant then
            $key \leftarrow component$
        else if $component$ is a mapped variable then
            $key \leftarrow entity.value$ of mapped attribute to $component$
        end if
    end for
    if $struct.value$ is a string value type then
        $value \leftarrow entity.value$ of mapped attribute to $struct.value$
    else if $struct.value$ is a hash structure then
        $value \leftarrow List < Tuple < String, String >>$
        for all $attr \in entity.attributes$ mapped to $struct.value$ do
            $value.add(attr.name, attr.value)$
        end for
    end if
    $query \leftarrow SET\ key\ value$
end if
```

share the same identifier value[2], a comparison of the value for each of the attributes they have in common is carried out (this is possible in the case of a duplication of data across physical structures permitted by the HyDRa framework, but not in the cases of a duplication of data in the same context). If we notice two different values for the same attribute, we have detected a data inconsistency for the same identifier.

[2]Each structure must indeed have the identifier, this is a constraint of the HyDRa framework (described in Section 5.7)

This error is recorded and reported to the user via a log. If no conflict is detected, *finalEntity* which contains all attributes of *E*, is added to the final list of results to be returned.

Algorithm 4 $join(List < List < E >> datasets, joinMode)$ algorithm

```
 1: $finalDataset \leftarrow datasets[1]$
 2: for all $d \in datasets$ do
 3:   for all $E\ entity1 \in d$ do
 4:     for all $E\ entity2 \in finalDataset$ do
 5:       if $entity1.id = entity2.id$ then
 6:         for all $attr \in \{entity1.attributes \cap entity2.attributes\}$ do
 7:           if $entity1.attr = entity2.attr$ then
 8:             $finalEntity \leftarrow entity1.attr$
 9:           else
10:             DATA CONFLICT FOUND for $attr$
11:           end if
12:         end for
13:         $finalDataset \leftarrow finalEntity$
14:       end if
15:     end for
16:   end for
17: end for
```

5.5 Illustrative Example

The conceptual schema of Figure 5.8 represents a more complete model including all the conceptual constructions supported by the HyDRa modelling language. This schema is based on the Northwind database [6] which is a representative example of data of a company, containing products, orders, suppliers, customers, *etc.* In this schema we find structures such as one-to-many relations, mandatory (1-1) or optional (0-1) cardinalities, many-to-many relations as well as a relationship type with attributes.

The physical schema, presented in Figure 5.9 illustrates the tables, collections, and sets of key values storing the data of the conceptual schema. In this physical schema, the following structures are notably present:

- Hybrid foreign key. *Products*, *Orders* and *Order_details* contain a reference field to a target database different from the source structure.
- Join table. *Order_details* is a join table representing a many-to-many relationship (*composedOf*) with attributes.
- Entity split. The entity type *Shippers* and its attributes are split into two different key-value structures, *SHIPPERS:[ID]:COMPANYNAME* and *SHIPPERS:[ID]:PHONE*, two queries are needed to retrieve all the attributes, as well as a join to reconstruct the conceptual object.

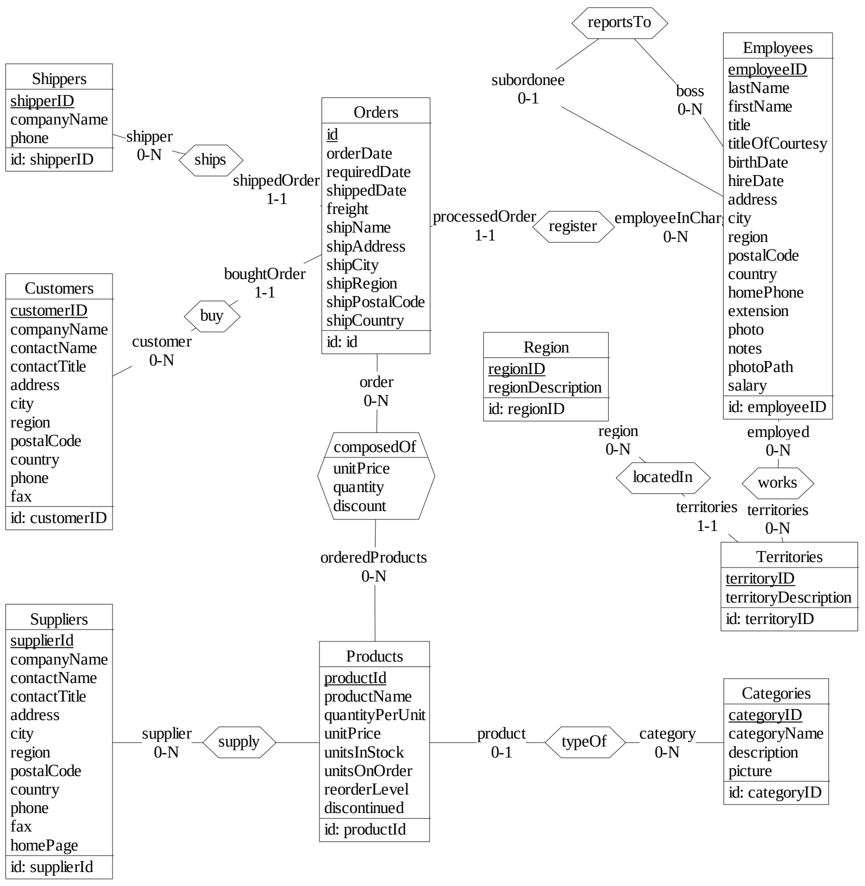


Figure 5.8: Conceptual model of running example

- Nested structure. The collection *Customers* contains an array of objects *orders*, containing the orders bought by a customer (expressing the *buy* relation of the conceptual schema).
- Data duplication. The data concerning the orders are present both in *Orders*, of the key-value database, and in the nested structure of the *Customers* collection in the document database.

In this section we will present some concrete examples of the generated Java code for the different specifications mentioned above. These examples present the important parts [3] of the specifications presented in Sections 5.3.3 and 5.4.

5.5.1 Select entity in split structures

In the illustrative use case presented, *Shippers* is an entity type whose complete set of conceptual attributes is distributed over several physical structures. According to the

[3]Lines of code not part of the core of the example case explained have been removed.

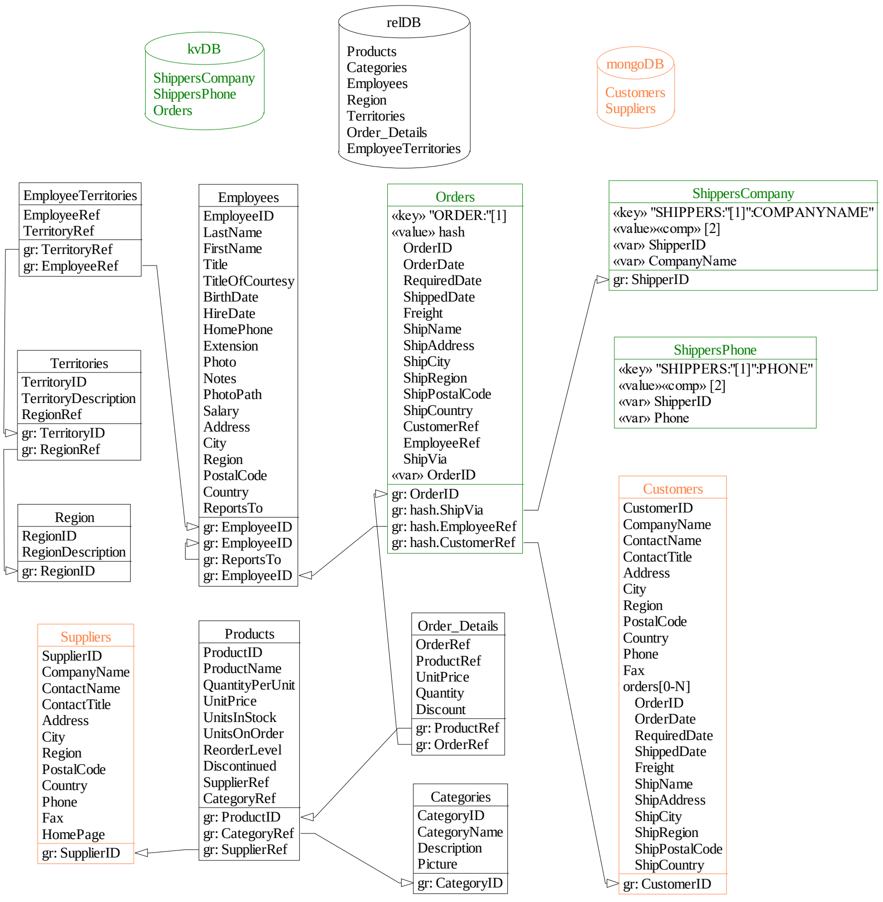


Figure 5.9: Physical model of running example

physical schema in Figure 5.9, the data concerning *Shippers* is found in the kvpairs *ShippersCompany* and *ShippersPhone*. This representation is an implementation of the design **key value per field**. Here these physical structures are on the same database.

According to the generator specification, each of the entity types as well as the relationship types will have a dedicated service class, allowing access to the data of the conceptual object it represents via different conceptual access methods. We will illustrate the generated code of the method *getShippersList(Condition condition)* in *ShippersService*, this method is the implementation of Algorithm 1. Listing 5.6 presents this code. This method takes as argument a *Condition* object which can also be null, *i.e.*, in the case of the selection of all the data of *Shippers*. It will return a *Dataset* object, a class inheriting from the *List* class, containing *Shippers* conceptual objects. The generator, having read the physical schema and mapping rules, identified that *Shippers* was distributed in two distinct physical structures. The implementation of the method to select all the entities will therefore call the two specific methods on the identified structures. At lines 6 and 8 we will retrieve the datasets coming from the two *kvpairs* involved. One of these specific methods is detailed in the next subsection. Structure specific results are stored in a provisional list which will then be combined via the join algorithm (Section 5.4.5) returning a single dataset of *Shippers* objects.

```
1  class ShippersService {
2      // ...
3      public Dataset<Shippers> getShippersList(Condition<ShippersAttribute>
        condition){
4              List<Dataset<Shippers>> datasets = new ArrayList<Dataset<
        Shippers>>();
5              Dataset<Shippers> finalResult = null, d = null;
6              d = getShippersListInShippersPhoneFromMyRedisDB(condition);
7              datasets.add(d);
8              d = getShippersListInShippersCompanyFromMyRedisDB(condition);
9              datasets.add(d);
10
11             if(datasets.size() > 1) {
12                 finalResult=fullOuterJoinsShippers(datasets);
13             }
14             return finalResult;
15         }
16     //...
17     }
```

Listing 5.6: Implementation of *getShippersList()* for Shippers entity type (structure split)

5.5.2 Select in specific database

In the previous example we showed that the call to *getShippersList* involved two calls to database specific methods implied by *Shippers*. Here we will detail one of these calls, *i.e., getShippersListIn**ShippersPhone**FromMyRedisDB* which will return a Dataset of *Shippers* objects, containing only the data of *Shippers* present in the kvpair structure *ShippersPhone*. This function is an implementation example of Algorithm 2. The main steps consist in the creation of a native query compatible with the target database type, then the execution and the retrieval of the results in a generic data format. And finally the last step converts these generic objects into conceptual objects.

As the target database is a key-value database, the retrieval of data can only be done via the key. First, we have to build the key pattern based on the pattern indicated in the physical structure, *i.e., "SHIPPERS:"**[ShipperID]**":PHONE*. It consists of three elements, two of which are string constants. The variable part *ShipperID* is mapped to the conceptual identifier of the entity type *Shippers*. The Lines 3 to 15 build the complete pattern by concatenating the elements. This part also reads the passed selection condition, in order to identify if it does not concern an element which is part of the key [4]. Indeed, in this case, it may be possible to directly indicate the value of the attribute to recover only the desired key. If it is not the case, the dynamic element of the key will be replaced by the wildcard character *. The

[4]This condition must also be an equality operator

treatment of a condition on an element of the value will be carried out in the method calling this method and is not the focus of this section.

Once the key pattern has been constructed, it is sent to the database and executed (line 19) in order to retrieve objects of type *Row*, which contain for the *ShippersPhone* structure two String objects with the retrieved key and value.

Then these objects will be converted into conceptual objects *Shippers*, for each of the attributes of this object we identify if its mapped physical fields are in the key or in the value, then according to a pattern and regular expression construction, the actual value is extracted and put into the object *shipper_res* via the corresponding setter. Given that the concerned structure contains only the identifier *ShipperID* and the attribute *phone*. The final shipper object will only be populated with those attributes. In the end a list (or Dataset) of partial *Shippers* is returned. As indicated in the previous section, in Listing 5.6, at line 12, it is the generic selection function that will take care of the join of the different partial Dataset retrieved from the two physical structures mapped to *Shippers*.

```
 1  public Dataset<Shippers> getShippersListInShippersPhoneFromMyRedisDB(
         Condition<ShippersAttribute> condition){
 2      // Build the key pattern
 3      keypattern=keypattern.concat("SHIPPERS:");
 4      if(...)// condition on attribute in key pattern
 5      {
 6          // Get value in condition argument
 7      }
 8      else{
 9          valueCond=null;
10      }
11      if(valueCond==null)
12          keypattern=keypattern.concat("*");
13      else
14          keypattern=keypattern.concat(valueCond);
15      keypattern=keypattern.concat(":PHONE");
16      // ...
17
18      Dataset<Row> rows;
19      rows = SparkConnectionMgr.getRowsFromKeyValue("myRedisDB",keypattern);
20      // Transform to POJO. Based on Row containing (String key, String value)
21      Dataset<Shippers> res = rows.map((MapFunction<Row, Shippers>) r -> {
22                  Shippers shippers_res = new Shippers();
23                  // attribute [Shippers.ShipperID]
24                  // Attribute mapped in a key.
25                  key = r.getAs("key");
26                  // Build pattern and regex code to retrieve attribute.
27                  // ...
28                  shipperID = // matched element;
29                  shippers_res.setShipperID(shipperID);
30                  // attribute [Shippers.Phone]
31                  // Attribute mapped in value part.
32                  value = r.getAs("value");
33                  // Build pattern and regex code to retrieve attribute.
34                  // ...
35                  phone = // matched element
36                  shippers_res.setPhone(phone == null ? null : phone);
37                  return shippers_res;
38              }, Encoders.bean(Shippers.class));
39      return res;
40  }
```

Listing 5.7: Specific implementation code for *getShippersListInShippersPhoneFromMyRedisDB*

5.5.3 Select by entity attribute

The selection methods also include simplified access methods that allow data to be selected according to the value directly given as an argument for a specific attribute. The implementation of these methods is very simple and consists solely of calling the generic *getEList* with a constructed *condition* transparent to the user. This code is presented in Listing 5.8.

```
public Dataset<Customers> getCustomersListByPhone(String phone) {
    return getCustomersList(conditions.Condition.simple(conditions.
     CustomersAttribute.phone, conditions.Operator.EQUALS, phone));
    }
```

Listing 5.8: Implementation of *getEListByAttribute()*

5.5.4 Role selection

As shown in Figure 5.3 in lines 7 and 8, the generated selection methods also exploit the relationship types and their roles. Thus, there are methods that can select data of one entity type based on another entity type linked by a relationship. In our conceptual schema of Figure 5.8 we can quote for example the presence of :

- *getBoughtOrderListInBuy(Condition<Customers> condCustomers, Condition<Orders> condOrders)*. This function retrieves the objects *Orders*, playing the role *boughtOrder* in the relationship *buy*. As arguments, *Conditions* objects (See 5.4.2) can be passed. They are specific to each of the two entity types involved in *buy*, namely *Orders* or *Customers*. There are also variations allowing to give an object as argument instead of a condition. *e.g., getBoughtOrderListInBuyByCustomer(Customer customer)*.
- *getCustomerListInBuy(Condition<Customers> condCustomers, Condition<Orders> condOrders)* This one is the opposite equivalent of the previous one. Here we want to retrieve *Customers* from the purchase relation.
- *getOrderListInComposedOf(Condition<Orders> orderCondition,Condition<Products> orderedProductsCondition, Condition<ComposedOf> composedOfcondition)* This method implies the many-to-many relationship type *composedOf* which also contains attributes, this implies the possible presence of a condition on the relationship type itself.

The signature of these methods follows a precise pattern based on the names declared in the conceptual schema. This takes the form of

get**ROLE1NAME**$ListIn$**RELATIONSHIPNAME**($Condition$...)
or
get**ROLE1NAME**$ListIn$**RELATIONSHIPNAME**By**ROLE2NAME**($ENTITY2$...)

for the instance-based version.

These methods are also conceptual and are therefore not impacted by possible physical mappings. The same results will be returned no matter if the database of *role 1* entity type data is different from the one where *role 2* entity type data is stored.

The implementation of these methods relies mainly on the entity type selection methods described above, and on the join algorithm that will return the right subset based on the fields containing the source and target references.

5.5.5 Insert an entity type

The insert method illustrated in Listing 5.9 allows to insert a conceptual object *Orders* into its mapped physical structures. The first important thing is that the signature of *insertOrders* does not only contain the *Orders* object to be inserted. Indeed, according to the Section 5.4.4 the signature also takes into account the cardinalities of the roles of the relations in which the entity type to be inserted is involved. Here according to the conceptual schema presented in Figure 5.8, the entity type *Orders* is involved in three relations in which *Orders* plays a role of cardinality 1-1. This means that an order cannot exist without a delivery person (*Shippers*), nor without an employee who has encoded it (*Employees*), nor without the customer who has placed this order (*Customers*). This is why in the signature of the insert function, instances of these objects are also requested.

Then the implementation of the insert will, similarly to the selection methods, call the database specific sub-methods. In the example case and given the physical schema of Figure 5.9, *Orders* is involved in (1) an ascending structure (*Customers*) (due to *boughtOrder* role), (2) a reference structure *Orders*, which is a kvpair stored in the *kvDB* key value database.

```
public void insertOrders(Orders orders, Customers customerBuy, Employees
      employeeInChargeRegister,
     Shippers shipperShips){
         // Insert in standalone structures
         // Insert in structures containing double embedded role
         // Insert in descending structures
         // Insert in ascending structures
         insertOrdersInCustomersFromMyMongoDB(orders,customerBuy,
      employeeInChargeRegister,shipperShips);
         // Insert in ref structures
         insertOrdersInOrdersFromMyRedisDB(orders,customerBuy,
      employeeInChargeRegister,shipperShips);
         // Insert in ref structures mapped to opposite role of mandatory
      role
     }
```

Listing 5.9: Insert code for Orders entity type

5.5.6 Insert in an ascending structure

The method in Listing 5.10 presents the implementation to insert the entity type *Orders* into the ascending structure *Customers* identified by the insert function. The *myMongoDB* database hosting the *Customers* collection is of type document. We must therefore first create the object of type *Document* which will be inserted using the native MongoDB driver, as described in Section 1.2.1. Lines 6 to 8 actually create this document by going through the attributes of the *Orders* object passed as parameter of the insert method. Only the attributes mapped to the physical structure considered will be taken into account.

Then the created order document, for the considered ascending structure must be added to the *orders* array of the client's root document. The client identifier is also passed in parameter. Line 13 declares the filter operator specifying the client identifier, allowing to select the right client on which to add the order. In the native mongodb library, there is a specific operator allowing to add an element to an array, Line 14 declares this parameter by assigning it the order document created above. This operator and the filter condition are finally transferred and executed on the *myMongoDB* database.

```
1  public boolean insertOrdersInCustomersFromMyMongoDB(Orders orders,
2      Customers customerBuy,
3      Employees employeeInChargeRegister,
4      Shippers  shipperShips) {
5              // Implement Insert in ascending complex struct
6          Document docorders_1 = new Document();
7          docorders_1.append("OrderID",orders.getId());
8          docorders_1.append("OrderDate",orders.getOrderDate());
9          // ... remaining attributes mapped to current physical structure.
10
11         // level 1 ascending
12         Customers customers = customerBuy;
13             Bson filter = eq("CustomerID",customers.getCustomerID());
14             Bson updateOp = addToSet("orders", docorders_1);
15             DBConnectionMgr.upsertMany(filter, updateOp, "Customers", "
       myMongoDB");
16
17         return true;
18     }
```

Listing 5.10: Insert code for Orders entity type in ascending structure

5.6 Algorithms

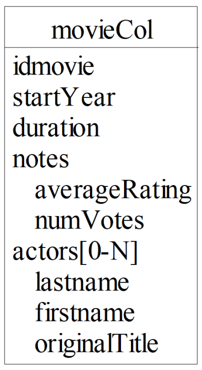
As described in the generated API specification, the methods used by the user are at a conceptual level of abstraction. For the selection methods, this means that no matter what the actual representation of the underlying physical structures is, with duplicate or entity split structure, the returned value will be a set (*i.e.*, no duplicate with respect to the identifier) of complete conceptual objects (which contain all the attributes declared in the conceptual entity type). The satisfaction of this requirement by the generated code, and by consequence also by the algorithm generating the code, depending on the complexity of the physical structures encountered, give rise to several challenges. The following section does not detail the those algorithms in a generic way but illustrates, via particular examples, some complex and/or original configurations that are managed by the code generator.

5.6.1 Creation of objects from multi level embedded structure

Returning complete conceptual objects is particularly complex in certain situations made possible by the nested structures of the document data model (Section 1.1.4). Indeed, these structures, in a single document, can hold several entity types, contain several instances thanks to the object arrays, and even have attributes of the same entity type spread over several embedded levels. This section illustrates such a situation with an example.

In the following structure (Figure 5.10) are represented data concerning movies in a collection (document database), at the first level are the identifier (*idmovie*), the year (*startYear*) and the duration (*duration*) of a movie. Then, in an object attribute, *notes*, the average of the notes (*averageRating*) as well as the number of votes (*numVotes*) regarding that particular movie are found. These attributes are *embedded* at a lower level. Next, in *actors*, an attribute of type array containing objects as well, we find the actors of this movie. However, here, for a reason specific to the user, the title of the movie *originalTitle* is included for each actor [5].

On the right side of the Figure we illustrate such structure in with example data in JSON format.



```
{
idmovie : 54836684,
startYear: 2020,
duration : 150,
notes : {
    averageRating : 7.3,
    numVotes : 487523
  },
actors : [{
  lastname : "Washington",
  firstname : "John David",
  originalTitle:"Tenet"
  },
  {
  lastname : "Pattinson",
  firstname : "Robert",
  originalTitle : "Tenet"
  }]
}
```

Figure 5.10: Example of structure where attributes are mapped to multiple nested level

When calling the *getMovieList* method, in order to build the complete *movie* object, it will be necessary to :

- Construct a *root* object with the movie attributes of the first level.
- Retrieve the attributes of the second level in the complex field *notes* and add them to the root object.
- Iterate on all the elements of the array *actors* in order to recover each *originalTitle* and check that they have the same value since they are linked to the same root object.
- Set the definitive title attribute to the root object and add it to the final returned list.

This process can be confronted with undefined levels of nesting and where the attributes of the concerned entity type are found at each of them. This is why the code implementing this reconstruction is a recursive algorithm.

[5]This is an interesting structure indeed, but it is possible as NoSQL models are designed to respond to queries that are as close as possible to the user's needs.

5.6.2 Filtering on attributes in a split entity

In the case where the attributes of an entity type are split over several databases (a construction called *entity structure split*), the selection based on a condition on one of these attributes must also be able to be satisfied. Indeed, *Conditions* objects passed as arguments to the selection methods are also conceptual and therefore independent of the database where the attribute of the condition is actually stored. In the example we show that a condition can concern an attribute present only in a single physical structure and that the data stored on the other physical structures will be retrieved and will also respect the selection constraint.

In Figure 5.11 is represented a many-to-many relationship type between movies and directors. *directorTable* contains information on the directors, *directed* is the join table containing the foreign keys to movie and director structures. The movie structure (*movieCol*) is a document collection containing information about movies as well as an array (*actors*) including the actors of that movie. For user-defined reason, the ratings information for a movie is stored in another structure (*movieRating*). Therefore, the conceptual attributes forming the complete conceptual entity type *Movie* are thus split into two structures. The access methods to a movie such as *getMovieList(Condition<Movie> condMovie)*, but also the access method to a director according to a condition on the movie *getDirectorListInDirect(Condition<Movie> condMovie)* must be able to satisfy any condition on any attribute of *Movie*.

For example in the case where the user wants to retrieve all movies whose average (*avgRating*) exceeds 9 the following steps will be performed:

- A query on *movieRating* respecting the condition on *avgRating* will be executed. It will thus return all the satisfying *Movie* objects. Which will only contain the attributes mapped in this structure (*numVotes* and *avgRating*).
- A query retrieving *movieCol* data will also be built and executed. However, here, there is no field allowing to satisfy the condition on the average, so the constructed query will retrieve all the data. A flag indicating that the condition could not be directly verified will be created.
- Two datasets containing *Movie* objects are retrieved. Each of them consists of *Movie* objects partially filled (given the attributes encountered in their respective structures). Moreover, one will contain more elements than the other one since the condition could not be directly satisfied using the query.
- These two results lists must be joined in order to reconstruct a complete *Movie* object. This join will read the selection condition and thus be in charge of re-filtering the possible *Movie* objects not satisfying the condition.

With this example we illustrate that it does not matter how the conceptual attributes are physically distributed in the different databases. The conceptual selection conditions can be satisfied.

5.6.3 Get role in relationship

We have specified that it is possible to select entity type data based on its role in a relationship, and also based on conditions on the other entity types involved in

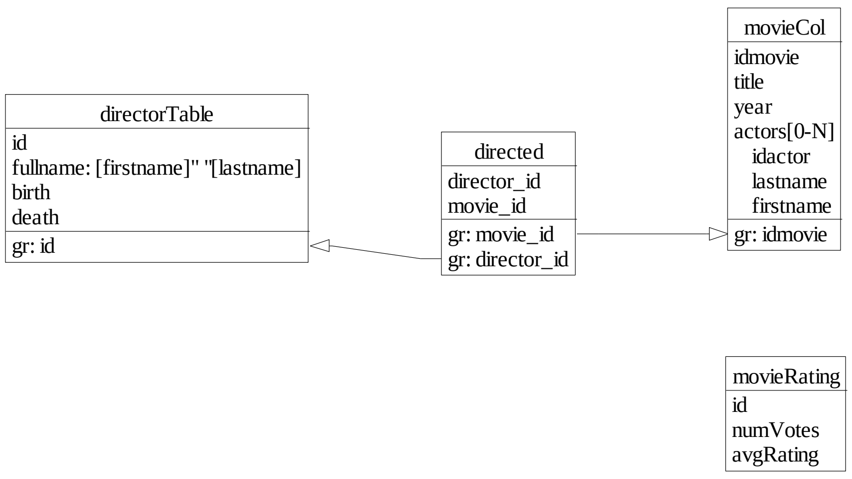


Figure 5.11: Movie split structure example

that relationship (Section 5.5.4). Their implementations are particular and several configurations must be taken into account.

If we consider $R(role_1 : E_1, role_2 : E_2)$, a relation R between two entity types, E_1 and E_2, playing respectively the roles named $role_1$ and $role_2$. In the data access class *E1Service* there are therefore selection methods that exploit the R relation. As a reminder, these take the following form:

- *getRole1ListInR(Condition<E_1> role1Cond, Condition<E_2> role2Cond)*
- *getRole1ListInRByRole2(E_2 role2)*

The implementation of these functions depends, like all the others, on the physical schemas and the mapping rules of the HyDRa schema. It varies greatly depending on the possibilities of physical configurations. We can quote several elements which influence the code to be generated:

- The physical structures hosting the data of $role_1$ and $role_2$. Are they in the same database or a database of the same type?
- The type of link between the two roles. Is it a foreign key reference, or a nested structure?
- The direction of the reference. Is it $role_1$ that references $role_2$ or the other way around?
- The type of the reference field. Is the reference mono or multivalued, *i.e.,* is it a single field or an array of references?
- The cardinalities of the roles. Is the relationship many-to-many? Is there a join table?

Each of these dimension leads to different implementations of the selection methods. These must be taken into account by the code generation algorithm. In order to illustrate the configurations managed by the algorithm we will define :

- $E1$ as being the physical structure being mapped to the conceptual attributes of E_1 (and thus to the role $role_1$ as well).

- $E2$, the physical structure mapped to the attributes of E_2.
- $E1 \rightarrow E2$. This arrow indicates that there is a **reference** (or foreign key) in the physical structure $E1$ to a field of $E2$.
- *refE1refE2.* Indicates a join structure that contains two fields referencing a value of $E1$ and a value of $E2$.

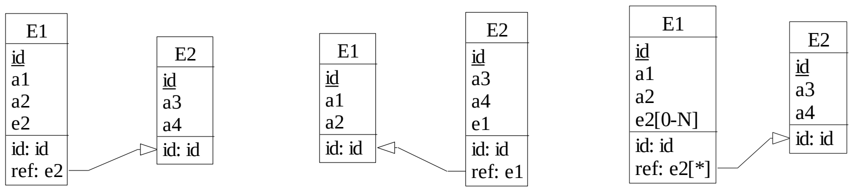
In the following we will describe different situations that have an impact on the implementation of role-based selection methods.

Reference structures

This category identifies configurations that contain two separate physical structures, declaring a reference from one to the other. Figure 5.12 represents some of these variations, 5.12a represents a foreign key field from the $E1$ structure to $E2$. This field will be used to host a reference value to the identifier of $E2$. When executing a role selection method we also need to capture this field and include it in the constructed native query. In Figure 5.12b this logic is reversed, and it is $E2$ which references $E1$.

Finally, the last impacting dimension is the type of reference, the schema 5.12c indicates that the field $e2$ which contains the reference to the structure $E2$ is an array of several values.

It is interesting to note that for all represented situations of Figure 5.12, if they are in the same database, implementation is again impacted. Indeed, in this case the generated code is different and more efficient in the sense that we can perform the two operations of selections and join in a single native query [6].



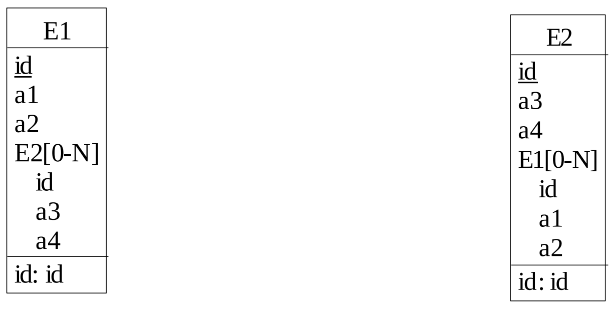
(a) $E1 \rightarrow E2$ (b) $E1 \leftarrow E2$ (c) $E1 \rightarrow E2$ (multi valued)

Figure 5.12: Reference structures

Embedded structures

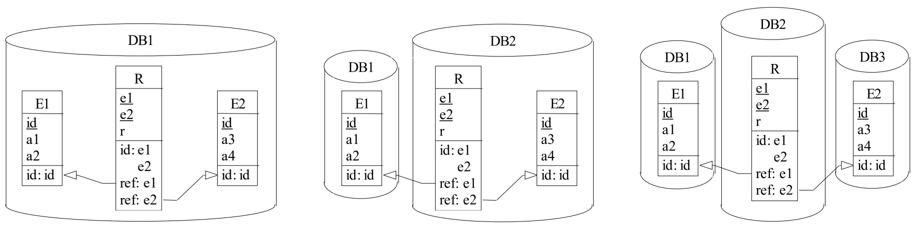
Figure 5.13 depicts the possibilities of modelling a conceptual relationship type between $E1$ and $E2$ in nested structures. There are mainly two ways, either the first entity type contains the second (Schema 5.13a) or vice versa (5.13b). Here we only represent a structure with two levels of nesting, the algorithm generating the access code takes into account nesting at any level.

[6] If the native database language allows it



(a) $E1[E2]$ (b) $E2[E1]$

Figure 5.13: Embedded structures



(a) $(E1 \leftarrow refE1refE2 \rightarrow E2)$ (b) $E1 \leftarrow (refE1refE2 \rightarrow E2)$ (c) $(E1 \leftarrow refE1refE2) \rightarrow E2$

Figure 5.14: Join structures possible physical representations

Join structures

Making a selection on the role of a relationship type is ultimately influenced by the presence of a join structure (and thus a many-to-many relation). The main challenge of the implementation algorithm will be influenced by the groupings of physical structures within the databases. The number of queries as well as their type will be different depending on whether the join structure is in the same database as one or both entity types (5.14a and 5.14b in Figure 5.14), or whether this structure is on an isolated database (5.14c).

5.7 Tool Implementation

In this section we will describe the technical aspects of implementing and using the HyDRa framework.

5.7.1 Architecture

HyDRa is implemented as an Eclipse plugin, publicly available on GitHub [53]. The plugin includes the textual editor for the polystore modelling language of Chapter

4, and the conceptual Java API generator. The modelling language grammar was specified with Xtext [7], a framework for developing Domain Specific Languages. The editor provides auto-completion, syntax checking and highlighting. It currently supports the design of relational, document, graph, column and key-value databases.

The user can create a polystore schema file to specify the conceptual and physical schemas as well as mapping rules to possibly pre-existing databases. This file can then be given as input of the API generator, that then produces a set of ready-to-use Java classes with their configuration files. The API generator uses the Acceleo technology [8] and as of June 2022 is made of 10302 lines of code (excluding comments, blank lines and generated code). The generated API currently supports selection and insertions operations on relational databases (MySQL and MariaDB), document databases (MongoDB) and key-value databases (Redis) via generated code using native access libraries (JDBC, Mongo Java driver and Jedis).

5.7.2 Modelling constraints

In order for the generated code to work properly there are some technical constraints that need to be respected. Some of them could be translated into validation rules directly in the schema editor of the Eclipse plugin. If any of these validation rules are not met, then the schema will be indicated as being in error and the line in question will be underlined (cf Figure 5.15). The rules implemented are the followings:

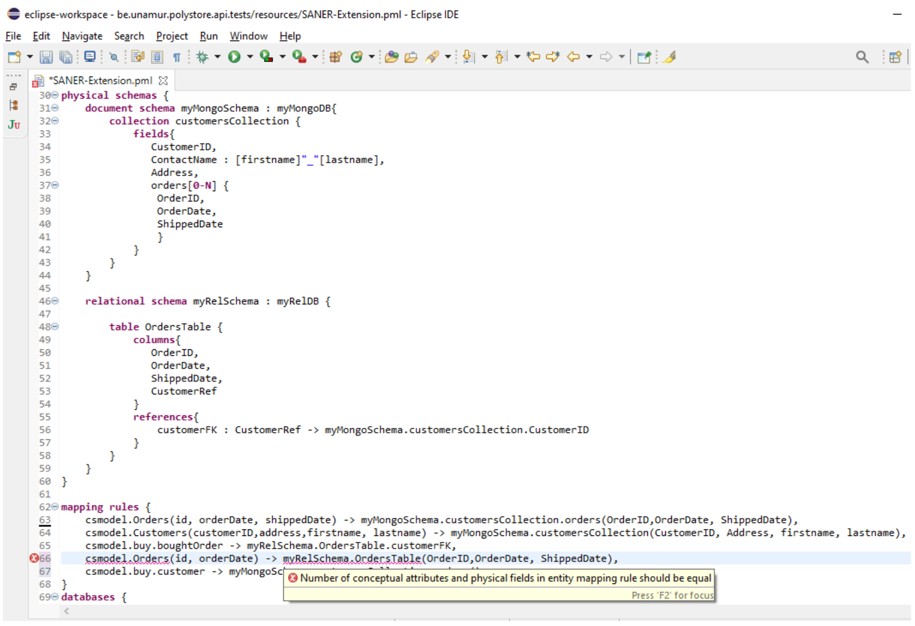
- Conceptual attributes must not start with an uppercase letter. As method names are generated according to the camelCase notation, a getter method generated on a "name" or "Name" attribute will generate two identical methods *getName()*. This must be avoided.
- Relationship role names must be unique among the model. To ensure uniqueness of attribute names, getters and setters in object classes without having too long names and signatures.
- Mandatory identifier. In order for the conceptual join (Section 5.4.5) and reconstruction of a single split entity type to work correctly, the join must be done on an identifier field. Therefore, conceptual entity types must declare one and furthermore, each physical structures hosting attributes of a particular entity type should also contain the identifier.
- The number of conceptual attributes specified on the left side of a mapping rule should be equal to the number of physical fields specified on the right side. This is so that each item listed has a match.

5.7.3 Plugins example usage

Figure 5.15 shows a view of the Eclipse plugin editor of a HyDRa model. We can see the syntax highlighting associated with the language and also notice the implementation of one of the validation rules. Here it indicates that the model being written is in error. The illustrated rule specifies that the number of conceptual elements on

[7] https://www.eclipse.org/Xtext/
[8] https://www.eclipse.org/acceleo/

the left of a mapping rule must be equal to the number of physical fields mentioned in the physical part on the right of the rule.


eclipse-workspace - be.unamur.polystore.api.tests/resources/SANER-Extension.pml - Eclipse IDE
File Edit Navigate Search Project Run Window Help
*SANER-Extension.pml

```
physical schemas {
    document schema myMongoSchema : myMongoDB{
        collection customersCollection {
            fields{
               CustomerID,
               ContactName : [firstname]"_"[lastname],
               Address,
               orders[0-N] {
                OrderID,
                OrderDate,
                ShippedDate
                }
            }
        }
    }

    relational schema myRelSchema : myRelDB {

        table OrdersTable {
            columns{
               OrderID,
               OrderDate,
               ShippedDate,
               CustomerRef
            }
            references{
                customerFK : CustomerRef -> myMongoSchema.customersCollection.CustomerID
            }
        }
    }
}

mapping rules {
    csmodel.Orders(id, orderDate, shippedDate) -> myMongoSchema.customersCollection.orders(OrderID,OrderDate, ShippedDate),
    csmodel.Customers(customerID,address,firstname, lastname) -> myMongoSchema.customersCollection(CustomerID, Address, firstname, lastname),
    csmodel.buy.boughtOrder -> myRelSchema.OrdersTable.customerFK,
    csmodel.Orders(id, orderDate) -> myRelSchema.OrdersTable(OrderID,OrderDate, ShippedDate),
    csmodel.buy.customer -> myMongoSc
}
databases {
```

```
Number of conceptual attributes and physical fields in entity mapping rule should be equal
                                                                        Press 'F2' for focus
```

Figure 5.15: HyDRa modelling language Eclipse editor

Once the HyDRa schema file (.pml) is completed and valid right-clicking on this file brings up the pop-up menu to generate the conceptual API code. Figure 5.16 illustrates this menu.

Finally, the project containing the model file will contain several packages, depicted in Figure 5.17 including the two packages containing the services classes and object classes.

5.8 Benefits

In this section we will review the benefits of using this conceptual API for manipulating a hybrid polystore.

5.8.1 Conceptual manipulation

Manipulating data via its conceptual representation allows the developer to abstract from the databases and thus not be required to master the data query language, or the access libraries as illustrated in Section 1.2.1. In this way the developer can write code that is completely transparent to the actual data storage. This is a potentially significant time saving.

Moreover, in a context of physical data evolution, *i.e.,* migration or duplication, preserving the conceptual model of the polystore. The application code written

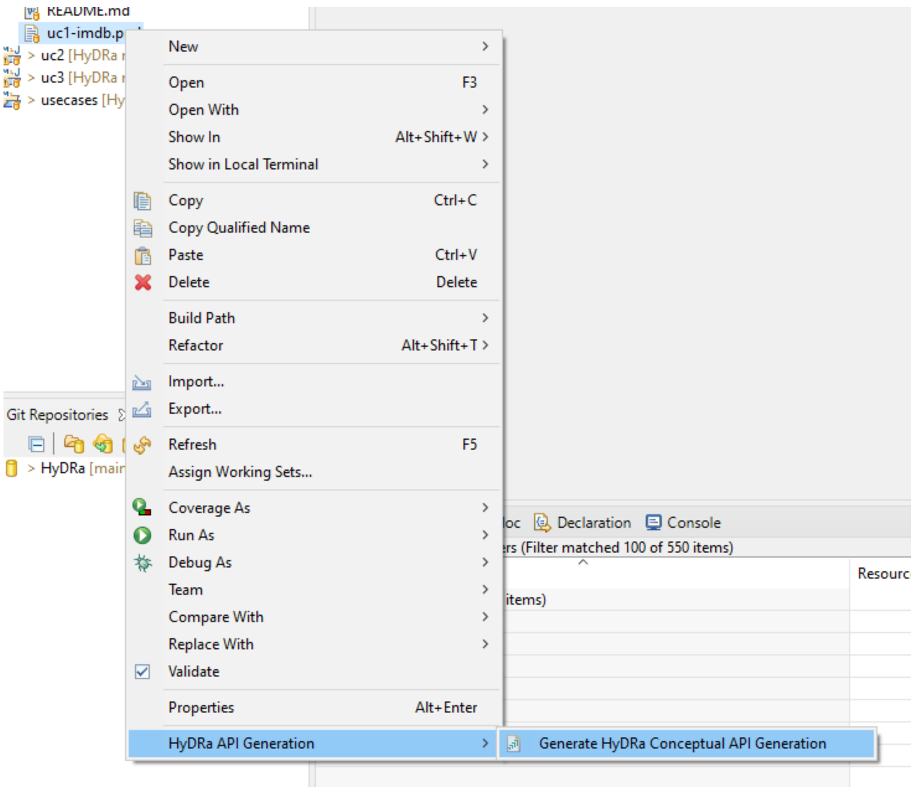


Figure 5.16: HyDRa API generator menu in Eclipse

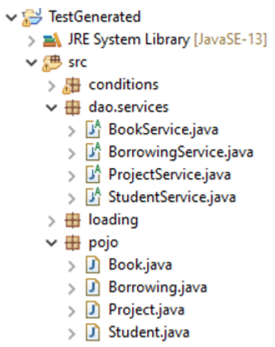


Figure 5.17: Example of generated packages and classes

using the conceptual API does not require modification. This is illustrated in the next sections.

5.8.2 Data validation

By exploiting the declared mapping rules and the generated conceptual API. HyDRa users are able to perform two types of data validation when using the API in their applications.

First, at the conceptual level, our entity reconstruction algorithm is independent of any entity structure split or duplication while retrieving data instances. It returns complete conceptual objects, by joining data from multiple backends when necessary. Therefore, HyDRa-based applications could detect possible inconsistencies between the values of a conceptual attribute stored into multiple databases. Going back to our running example in Figure 5.9, *Orders* attributes are duplicated into two databases, first in *Orders* key value structure and second in *Customers* document collection. When querying the *Order* entity type through the conceptual API, the latter would then detect when data instances sharing the same identifier have different orders attribute values. This verification happens when joining the datasets, which was detailed in Section 5.4.5.

A second data validation is carried out at the level of referential integrity. Thanks to the mappings from relationship type roles to reference blocks, a generated HyDRa API provides data manipulation methods exploiting the related entity types. An example of role-based selection method is *getBoughtOrderListInBuy*, role based selections were detailed in Section 5.5.4. A similar referential constraint implementation is provided for insert methods where, due to the presence of a mandatory role, inserting a new *Order* requires a *Customer* object, validating that the given entity to save contains mandatory linked entities, as a referential value or as an embedded structure.

Thanks to these two validation methods, a developer can detect possible inconsistencies in the data when using the selection methods. This is particularly useful when HyDRa is used in a database reverse engineering context. The use of the insertion methods provided by the API allows keeping **conceptual consistency** of the data across several heterogeneous databases.

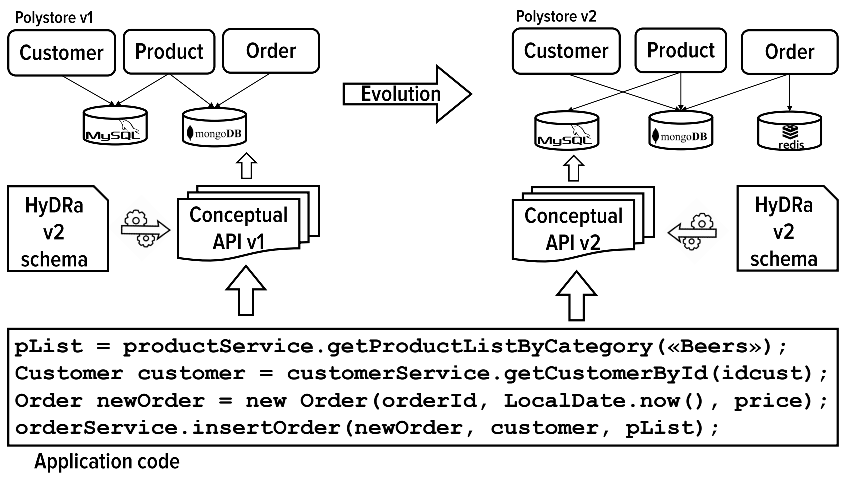
5.8.3 Evolution

Evolving data intensive systems is a complex and error-prone task, since it often requires co-evolving the application code to keep the system consistent. HyDRa, by generating conceptual data manipulation APIs, aims to facilitate this co-evolution problem.

Figure 5.18 illustrates how HyDRa can help with this co-evolution task. The upper left of the figure depicts a first version v_1 of a polystore system, manipulating *Product*, *Order* and *Customer* entities. The mapping rules are represented with arrows, linking the conceptual elements to their physical databases. *Customer* instances are stored in a relational MySQL database, *Orders* are stored in MongoDB database and *Product* data is stored in both databases. This v_1 version of the polystore schema was given as input to the HyDRa API generator. Based on the generated API, developers developed application code accessing the polystore databases declared in v_1.

Let us now assume that evolving requirements require the polystore to evolve from version v_1 to v_2. *Customer* is migrated to the MongoDB database and *Order* is duplicated in a newly deployed key-value Redis database. Without HyDRa, application programs would use database native queries to access the polystore data. Those programs would need to be manually rewritten to (1) change the queries manipulating *Customer* from MySQL to MongoDB, (2) add Redis queries for *Order* data, (3) add glue code to handle the duplication of *Order* data in multiple databases. On top of that, the developers would need to know, or to learn, the Redis query language.

In contrast, by using the HyDRa API conceptual methods to access the polystore in version v_1, the developers only need to adapt the HyDRa polystore model, regenerate the API, and recompile their application programs. The latter remain unchanged and can immediately manipulate the v_2 data structures of the target polystore.



```
pList = productService.getProductListByCategory(«Beers»);
Customer customer = customerService.getCustomerById(idcust);
Order newOrder = new Order(orderId, LocalDate.now(), price);
orderService.insertOrder(newOrder, customer, pList);
```

Figure 5.18: HyDRa API In Evolution Context

5.8.4 Improving test coverage

The low rate of test coverage of the database access methods is due to several factors identified in Chapter 3. The generation of this code as presented in the current chapter brings solutions to some of these problems in order to allow a better coverage of tests and thus a better quality of the system.

Thanks to the *databases* section of the HyDRa modelling language (see Section 4.4) it is easy to generate an API specific to the environment dedicated to the tests. This makes it possible to easily insert test data via the use of conceptual insertion methods. Indeed, one of the important problems highlighted by the taxonomy of difficulties (see Chapter 3, Section 3.3) is the *database management, i.e.,* the management of connections and data specific to test databases.

Conceptual access methods integrates the handling of the databases connections and allows easy access to the data. Developers can thus write tests in a more intuitive way which requires less effort. Furthermore, those conceptual level tests can benefit from the support of evolution as described in the previous section (Section 5.8.3), in this way the tests remain consistent with the evolution of the schema, this being also a recurrent problem raised by developers. Finally, it is also possible to generate test classes in parallel to the service classes and object classes. These can therefore integrate a certain number of best practices recommended by the developer community. This is a future development of the API.

5.9 Conclusion

In the last two chapters we have detailed the two main components of the HyDRa framework, *i.e.,* the modelling language and the conceptual API generator. The HyDRa language has made it possible to separate the conceptual schema (the application domain of the system) from the physical schemas (the actual representation of the data in the databases). The link between these two parts is established via the mapping rules offered by the language.

Based on this HyDRa schema, this chapter has described how **a conceptual data access library** is generated. First we have described the different elements generated by our algorithm. Two main classes are thus at the heart of the API, (1) the object classes, which are the representations of the conceptual entity and relationship types, and (2) the service classes, which contain data manipulation operations using the conceptual object classes.

Second we presented the main steps each method generated must respect. The implementation of these manipulation methods depends on the correspondences with the actual physical schemas, but the main steps are common.

In the third part of this chapter we have illustrated some available generated methods and their implementation with the help of the actually generated Java code. Then we presented the different challenges that the code generator algorithm had to meet, and we also listed some technological constraints of the chosen tools.

In the light of these descriptions and illustrations, we then reviewed the advantages brought by such a conceptual access code generator, *i.e.,* keeping **conceptual consistency** across multiple heterogeneous databases, allowing **conceptual querying** and **facilitating co-evolution of application code** towards structural evolutions

CHAPTER 6

EVOLUTION FRAMEWORK & QUERY ADAPTATION

Contents

In this chapter we propose a theoretical framework to manage the evolution of database schemas with a top-down approach, starting from the evolution operation to all the involved artefacts. Then we detail an approach developed to automatically adapt queries according to schema evolutions.

The work presented in this chapter has been published [48,52] in the 46th International Conference on Very Large Databases (VLDB 2020 PhD Workshop) and in the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2020 NIER Track) where it received the Best New Idea and Emerging Results Paper Award.

6.1 Introduction

Database schema change has long been recognized as a complex, time-consuming and risky process. These evolutions can be of different types, depending on whether they impact the semantics of the conceptual schema (by adding or removing constructions), or whether they only affect the structure of the data (constant semantic evolutions). Moreover, these evolutions have an impact on several artefacts of the data intensive system considered, those involve (1) the structure of the databases, (2) the format or the data themselves, (3) the conceptual, logical or physical schemas as well as (4) the programs or queries manipulating these databases. The developer in charge of implementing a database schema change will have to be able to identify and adapt these artefacts correctly in order to keep the system running. This co-evolution task is therefore at high risk of introducing errors, and has already been the subject of several approaches, techniques and tools (detailed in Chapter 2). We argue that the difficulty of this task is even higher in a hybrid polystore context.

In the previous chapters we have presented a modelling language allowing to unify, for a system comprising several databases, the conceptual and physical schemas. On the basis of this model, a conceptual data access library has been proposed, and we have described the advantages that this approach brings to the evolution of databases (Section 5.8.3) and particularly the support to the evolution of data manipulation programs.

In this chapter, we first propose a complete theoretical framework for the evolution of polystores implementing a top-down approach. It starts from the desired schema evolution operation and goes towards the different impacted artefacts. Secondly, we will detail an additional approach to the evolution of data manipulation code, this time focused on queries.

6.2 Background

The work presented in this chapter has been developed within the framework of the European project Typhon [8]. It is a project involving several academic and industrial partners with the aim of developing a global architecture allowing the design, manipulation, evolution, analysis and deployment of hybrid polystores. The contributions concerning the evolution of hybrid polystores are therefore partly based on two elements, (1) TyphonML and (2) TyphonQL developed by other partners of the project.

TyphonML Firstly, the TyphonML modelling language, a polystore modelling language, at the basis of the reflection and development of the HyDRa language (Chapter 4). Both languages share the presence of a conceptual part that describes the semantics of the data of the polystore (in terms of entity types and relations between them), as well as a physical part describing the physical components (tables, collections, etc.) of the databases involved.

Compared to HyDRa, TyphonML imposes implicit restrictions on the way conceptual entities, attributes and relationships are physically translated in each different native backend. In other words it does not leave developers the freedom to explicitly define the mappings between the conceptual schema elements and the physical schemas of the polystore at the level of the attributes. In addition, it does not support data structure split, data instance partitioning, and data redundancy across the different polystore databases. However, TyphonML integrates an additional section in its language allowing to specify schema evolutions to be performed.

This part represents the starting point of the evolution framework and will be described later in this chapter. Listing 6.1 exposes an example of TyphonML schema, where we notice the presence of the conceptual and physical schemas sections, and finally in the last lines the presence of change operators.

```
// Conceptual section
entity Description{
    id : int
    description : string[500]
    product :-> Product[1]
}
entity Product{
    id : string[256]
    name : string[50]
    price: float
}
entity Order{
    id: int
    total_price : float
    products :-> Product[0..*]
    owner :-> User[1]
}
entity User{
    id : int
    name: string[50]
    cardNumber : string[16]
    orders :-> Order."Order.owner"[0..*]
}

// Physical mapping section
relationaldb RelationalDatabase{
    tables{
        table{
            ProductDB: Product
            index productIndex{
                attributes('Product.name')
            }
            idSpec('Product.name')
        }
    }
}
documentdb DocumentDatabase{
    collections{
        UsersDB: User
    }
}

//Schema Modification Operators
ChangeOperators[
        merge entities Product Description as Product,
        split entity User to CreditCard attributes:[cardNumber]
    ]
```

Listing 6.1: TyphonML example

TyphonQL A Typhon polystore can be queried using the TyphonQL language [82]. The language proposes a unified concrete syntax for CRUD operations performed on the polystore. The TyphonQL query engine compiles those queries into native queries manipulating the actual databases of the polystore. This means that migrating a polystore entity from one database platform to another within the polystore, does not impact the related TyphonQL queries, which may remain unchanged. The

TyphonQL engine maps dynamically the query towards the right DBMS based on the physical part of the TyphonML schema. Some examples of TyphonQL queries are given in Listing 6.2.

```
// select the review entity of a specific product
from Product p select p.price where p.name == 'laptop'

// insert a product
insert Product {id: '298', name:
'kettle', price: 5.23}

// update
update Product p where p.id == 563 set{name: 'laptop'}
```

Listing 6.2: Query examples

Our query adaptation approach presented next in this chapter is able to automatically transform TyphonQL queries based on given evolution operations present in the TyphonML schema.

6.3 Evolution Framework

Based on the arguments developed in the introduction, we believe that the database schema evolution process can greatly benefit from an integrated approach. This approach allows to perform the co-evolution of other artefacts of the system in an exhaustive and semi-automatic way by having as a single starting point the evolution operations to be performed. This section describes the theoretical framework developed for this purpose.

Below we specify the elements that constitute the expected inputs and outputs of the framework.

Inputs

- M_{Source}, represents the hybrid database schema at both conceptual and physical levels.
- $\{SMO\}$ is a set of schema modification operators to apply to M_{Source}.
- $\{DB_{Source}\}$ is the set of source native data structures and data instances.
- $\{Q_{Source}\}$ is the set of existing queries expressed on M_{Source}.

Outputs

- M_{Target} is the source model adapted according to the evolution operators in $\{SMO\}$.
- $\{DB_{Target}\}$ represents the set of database structures and instances adapted according to those operators.
- $\{Q_{Target}\}$ gives, when possible, equivalent queries as $\{Q_{Source}\}$, but expressed on M_{Target}.
- $\{DAO\}$ is a set of database access classes on specific drivers.

Figure 6.1 summarizes the different components of our framework. The rest of this section will further detail each of those components.

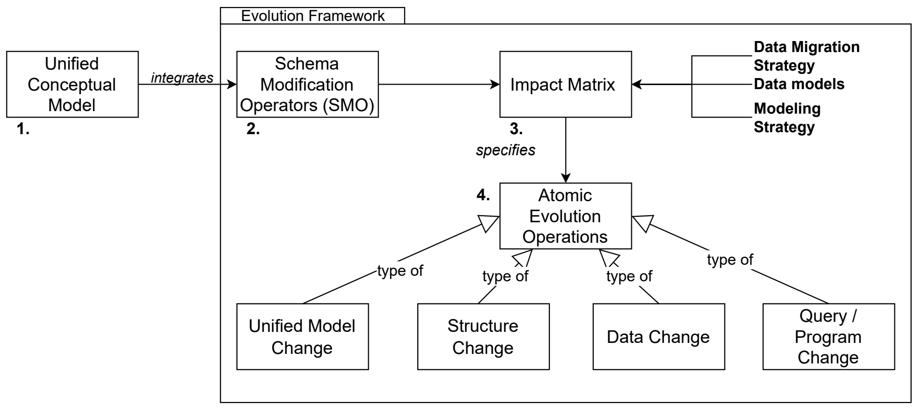


Figure 6.1: Evolution Framework

Unified Conceptual Model (1)

Our approach starts from a data model that allows one to design hybrid polystore systems. The HyDRa modelling language presented in this thesis is an example of such a unified language integrating multiple data models (*i.e.,* relational, document, key value, graph and column based). The evolution framework implements a top-down evolution approach, meaning it propagates from model to the related software artefacts (data, queries, programs). For this reason the modelling language must therefore be extended and integrate *schema modification operators* to allow users to trigger the execution of evolution operations. HyDRa does not integrate these evolution operators in the language for the moment, however they have been tested via the TyphonML language [23].

Schema Modification Operators (2)

Schema Modification Operators (SMO) are the entry point of the whole evolution process and help keeping the software artefacts of the hybrid polystore consistent with each other. It is a combination of a domain model object (*i.e.,* a conceptual element) and an evolution operation. The domain model object can be either *entity type, attribute, relationship,* or *identifier.* The available evolution operations span from simple changes, manipulating a single domain object *add, remove, rename, modify* to more complex ones such as *horizontal split, vertical split, merge,* or *migrate.* Figure 6.2 gives an example list of operators applied in an extended unified data model syntax. First is the example of adding a *rating* attribute of type *int* to the entity type *Review.* Then we want to add a one-to-many relationship *responses* from *Review* to *Comment.* And finally the last example of operator expresses the will to merge the two types of entities *User* and *CreditCard.*

```
changeOperators [
  add attribute rating : int to Review,
  add relation responses to Review -> Comment[0..*],
  merge entities User CreditCard as User
]
```

Figure 6.2: Example of evolution operators expressed within the TyphonML unified model

Impact Matrix SMO (3)

The impact matrix defines the atomic operations that need be applied to related software artefacts to actually propagate the polystore schema evolution operators. The rows of the matrix correspond to the available *Schema Modification Operators*. The columns of the matrix then characterize each of the operators according to several dimensions; each influencing the propagating operations that should be applied. The first dimension specifies the underlying data models of the object(s) subject to evolution. The possible values of this dimension include *relational, document, key-value, graph* or *column* databases. The second dimension concerns the chosen design strategy, as detailed in Section 2.2. The third dimension relates to the chosen data migration strategy (as stated in Section 2.4 migrating data in NoSQL systems can follow different strategies). Each cell of the impact matrix specifies the set of atomic, paradigm-specific operations to apply to the polystore artefacts (native structures, data, queries, etc.), in order to propagate the requested SMOs according to the dimensions' values.

Atomic Evolution Operations (4)

The set of atomic operations will be an exhaustive list of function specifications. Their goal is to enable the actual transformation required to propagate the desired schema evolution. They are classified into four categories, depending on which software artefacts are involved.

- *Unified conceptual schema change.* Each evolution operator will result in a change in the unified schema reflecting that evolution.
- *Native data structures.* According to the underlying logical mappings some SMOs require the adaptation of the native data structures. This adaptation, in the case of a relational database, translates as SQL *DDL* commands, *i.e., create table, alter table, create index, add constraint,...*. Equivalent structure manipulation for NoSQL data models include *create collection, create index* for document data model, *create table, create column family* for column databases.
- *Data.* The propagation of operators to data instances aims at transforming/migrating the data in order to make them comply with the target polystore data model/paradigm.
- *Queries and programs.* Evolution operations may also require the adaptation of existing database queries, that would become invalid due to the applied schema changes. Depending on the semantics-preserving nature of the op-

erator, it will be possible or not to translate the source database queries into equivalent queries expressed on the target polystore schema, using rule based transformations.

6.4 Query Adaptation

Chapter 5 presented the conceptual data access library generated based on the HyDRa schema. The use of this library in application code made this code impervious to evolutions of physical structures. Evolutions that did not modify the conceptual schema. The approach presented in this section focuses on **query adaptation** managing evolutions made on the conceptual schema. The proposed approach takes advantage of a conceptual modelling language for representing the polystore schema (TyphonML), and considers a generic query language for expressing queries on top of this schema (TyphonQL). Given a source polystore schema, a set of input queries and a list of schema change operators, our approach (1) identifies those input queries that cannot be transformed into equivalent queries expressed on the target schema, (2) automatically transforms those input queries that can be adapted to the target schema, and (3) generates warnings for those output queries requiring further manual inspection.

6.4.1 Approach

Hence, our problem becomes: *how to adapt TyphonQL polystore queries to an evolving TyphonML polystore schema?* Without the TyphonML and TyphonQL abstractions, this process would require query transformation rules for each specific native query language of the polystore. This would result in a complex and hard to maintain implementation. The adoption of the TyphonQL unified query language allows us to mainly focus on the semantic changes applied to the polystore schema. The evolution process of an hybrid polystore consists into producing a target version of the polystore starting from a source model and applying a set of Schema Modification Operators. Figure. 6.3 depicts the evolution process involved in the query adaptation tool. It takes as input a source TyphonML schema (as described in Listing 6.1), a set of TyphonQL queries (as in Listing 6.2) running on the source schema and a set of Schema Modification Operators (SMOs section in Listing 6.1) to apply on the source schema. The output of the query adaptation process is the transformed set of queries running on the target TyphonML schema with their categories and annotations.

The starting point of the query evolution process are the *Schema Modification Operators* (Section 6.3). They are evolution operations that manipulate objects of the TyphonML model, those objects include the **Entity**, the **Relation** or the **Attribute**. Our query adaptation approach is able to handle all changes that could happen on one of those three types of objects. The evolution operators can be classified in three categories, depending on their semantic impact, *i.e.,* the extent to which they preserve, augment or decrease the informational content of the polystore. A semantics-preserving schema modification ($S^{=}$), also called *schema refactoring,*

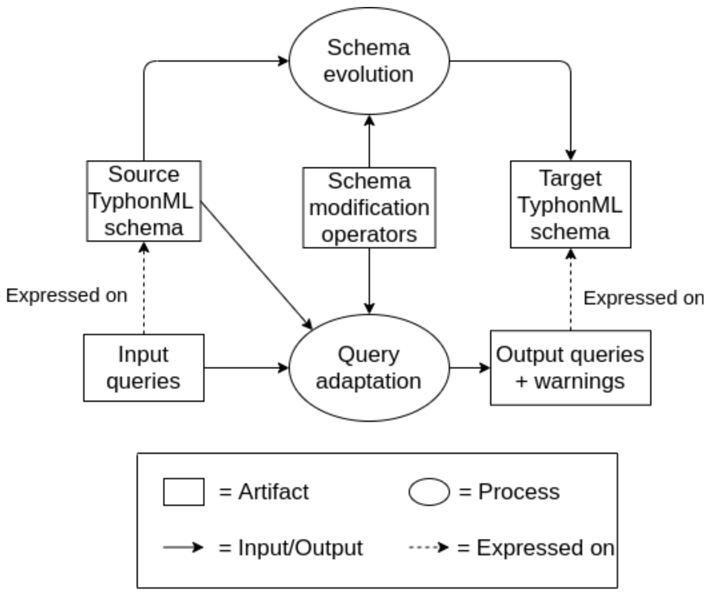


Figure 6.3: Overview of the hybrid polystore evolution process

does not impact the informational content of the polystore, but only the way the data is structured. This is the case, for instance, when renaming an attribute or when migrating an entity. *Semantics-augmenting* schema modifications (S^{+}) add informational contents to the polystore, for instance by adding an entity or an attribute. Conversely, *semantics-decreasing* schema modifications (S^{-}) remove some informational contents, *e.g.,* when removing an entity or when restricting the cardinalities of a relationship.

$S^{=}$ operations can generally be propagated automatically to related queries and in some cases the queries may even be left unchanged. In the case of S^{-} or S^{+} operations, automated adaptation of queries is not always possible or needed. To express those different situations, our query adaptation process distinguishes four possible categories of output queries:

- **(U)NCHANGED**: the input query has not been changed since it remains valid with respect to the target schema;
- **(M)ODIFIED**: the input query has been transformed into an equivalent output query, expressed on top of the target schema;
- **(W)ARNING** : the output query (be it unchanged or transformed) is valid with respect to the target schema, but it may return a different result set;
- **(B)ROKEN**: the input query has become invalid, and it cannot be transformed into an equivalent query expressed on top of the target schema.

The queries labelled as BROKEN or WARNING are also annotated with a message explaining to the user which operator caused trouble. This helps the user to identify the issue and to manually adapt the query (or its context) to the new schema semantics, when needed/possible. Table 6.1 lists all the schema modification operators that we currently support and shows the worst result expected by this change on each type of queries (create, read, update and delete).

Object	**Operation**	**Semantic class**	**Create**	**Read**	**Update**	**Delete**
Entity	Add	S^+	U	U	U	U
	Remove	S^-	B	B	B	B
	Rename	$S^=$	M	M	M	M
	Merge	$S^=$	B	W	M	W
	Split	$S^=$	B	W	B	B
	Migrate	$S^=$	U	U	U	U
Attributes	Add	S^+	B	W	U	W
	Remove	S^-	B	B	B	B
	Rename	$S^=$	M	M	M	M
	Change type	$S^{-/+}$	W	W	W	W
Relations	Add	S^+	U	U	U	U
	Remove	S^-	B	B	B	B
	Cardinality change	$S^{-/+}$	W	W	W	W
	Rename	$S^=$	M	M	M	M

Table 6.1: Schema modification operations supported by our approach and the worst result category for each input query type. **U**: Unchanged, **M**: Modified, **W**: Warning, **B**: Broken

6.4.2 Implementation

Figure 6.4 goes into details of the query adaptation process. Firstly the tool parses the TyphonML schema provided as input to extract, on one hand, the current model structures and, on the other hand, the set of SMOs to apply. Secondly each operator is applied sequentially to each input query. The operator, the query and the schema are passed through routing rules that send them to the correct handler function according to the change operator processed. This routing is done by using extensively the pattern matching scheme of the Rascal Meta Programming Language [80]. Finally, the handler function produces the required transformations and query adaptation category. This complete structure makes it easy to support further additional SMO. The developer just has to add a new routing rule and a new handler function for the new change operator.

The resulting tool consists of an Eclipse plugin. Its code is open sourced[1] and an installation guide is also available along with demo scripts.[2]

[1]https://github.com/typhon-project/typhon-evolution/tree/master/plugin-evolution
[2]https://figshare.com/articles/online_resource/Query-adaptation-examples/12821567

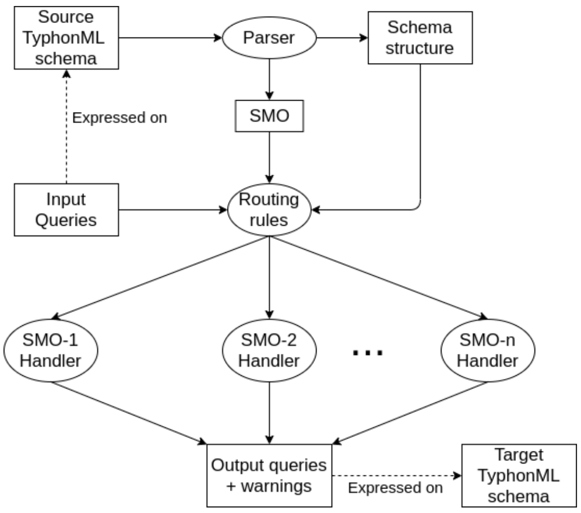


Figure 6.4: Detailed view of the *Query Adpatation Process*

6.4.3 Illustrative example

In this section we illustrate the query adaptation process using a concrete example. Previously described Listing 6.1 represents a TyphonML schema containing evolution operators, which describe the modification required by the user. The change operators are applied sequentially to the source schema. In this example, firstly the user wants to merge the *Description* entity type into the *Product* entity type (Line 45) and secondly she wants to extract the attribute *cardNumber* of the entity type *User* to a new entity type called *CreditCard* (Line 46).

Applying those evolution operations consists of several atomic operations, respectively removing the *Description* entity, adding an attribute *description* to entity *Product* for the merge operation, the creation of a new relationship *to_CreditCard* in *User* and the creation of entity *CreditCard* for the split operation. The application of those operators results in a target TyphonML schema as shown in Listing 6.3.

```
entity Product{
    id : string[256]
    name : string[50]
    price: Real
    description : string[500]
}
entity Order{
    id: int
    total_price : float
    products :-> Product[1..*]
    owner :-> User[1]
}
entity User{
    id : int,
    name: string[50],
    orders :-> Order."Order.owner"[0..*]
    to_CreditCard :-> CreditCard[1]
}
entity CreditCard{
    cardNumber : string[16]
}
//... Physical section
```

Listing 6.3: Target TyphonML Schema

Let us consider the set of TyphonQL queries expressed in Listing 6.4. They consist of CRUD queries involving entities or attributes impacted by the provided SMOs.

```
/*1*/ from Product p, Description d select d.description
where d.product == p, p.id == "AZKIU",
/*2*/ from User u select u.cardNumber where u.name == "Doe",

/*3*/ insert User {id: 1, name: "Doe", cardNumber:"536864726"},

/*4*/ delete Product p where p.id = "EYIR",
/*5*/ delete User u where u.id == 5,

/*6*/ update Product p where p.id == "EYIR"
set{name: "Blender"},
/*7*/ update User u where u.id == 5
set {cardNumber:"5362637"}
```

Listing 6.4: TyphonQL queries expressed on the source TyphonML schema

The result of the query adaptation process is shown in Listing 6.5.

```
/*1*/ MODIFIED
#@ Product and Description merged @#
from Product p select p.description
where p.id == "AZKIU",

/*2*/ WARNING
#@ Entity User split into User, CreditCard @#
from User u, CreditCard c select c.cardNumber
where u.name == "Doe", u.to_CreditCard == c,

/*3*/ BROKEN
#@ Entity User split into User, CreditCard @#
insert User {id: 1, name: "Doe", cardNumber:"536864726"},

/*4*/ WARNING
#@ Product and Descriptions merged.
Delete will erase more information than before @#
delete Product p where p.id == "EYIR",

/*5*/ BROKEN
#@ Entity User split into User, CreditCard @#
delete User u where u.id == 5,

/*6*/ MODIFIED
#@ Product and Description merged @#
update Product p where p.id == "EYIR"
set{name: "Blender"}

/*7*/ BROKEN
#@ Entity User split into User, CreditCard @#
update User u where u.id == 5
set {cardNumber:"5362637"}
```

Listing 6.5: Output TyphonQL queries expressed on the target TyphonML schema

- Query 1 in Listing 6.4 selecting the *Description* attribute does not require the join condition anymore as this attribute in the target schema is now in the *Product* entity.
- Query 2 now needs a new join condition.
- Query 3 & 7 are now broken as *cardNumber* is not in *User* anymore.
- Query 5 is also marked as broken as *cardNumber* will not be deleted with the user anymore, in order to keep the same semantics two queries are required, one deleting the correct *CreditCard* entity and one deleting the *User*. This multi query adaptation constitutes a current limitation of this tool.
- Query 6 is not changed as it does not involve impacted attributes.

Using the adaptation classification (broken, warning, unchanged, modified) and its motivation messages above each query the user can make an informed decision of whether to use or not the output queries in his programs.

6.5 Conclusion

In this chapter, we have first proposed a **theoretical framework for evolution** allowing to implement an evolution management of database schemas via a top-down approach. **Evolution operators** are integrated into the unified modelling language of the hybrid polystore. These operators characterize an evolution on one of the conceptual or physical elements declared in the schema. An **impact matrix** taking into account all the dimensions of the polystore (the type of databases, the design or data migration strategy *etc.*) as well as the evolution operator makes it possible to specify each of the **atomic evolution operations** to be carried out on each of the system's artefacts (structures, data, models and programs).

In a second part of this chapter we proposed a tool-supported approach, available through an Eclipse Plugin[3], which, given a conceptual polystore schema and a list of changes applied to this schema, is able to **transform polystore queries** into equivalent queries expressed on the evolved schema, when it is possible. If this is not possible, the user is provided with insight about what went wrong or what should be checked manually. The proposed approach is designed to work on a hybrid polystore, and relies on (1) a conceptual modelling language for representing the polystore schema, (2) a finite set of atomic schema modification operators to apply to this schema, (3) an intermediate polystore query language enabling the manipulation of the polystore data independently of the physical platforms (relational, NoSQL) where the data are actually stored.

[3]https://figshare.com/s/bf85f501ec3e6546df45

Part III

Evaluation

CHAPTER 7

USER EVALUATION

Contents

In this chapter we describe an evaluation of the HyDRa framework conducted on a set of about 40 students. First we present the context of the experiment as well as the profile of the participants, then we analyze the results and finally conclude with the analysis of the survey to which each of them responded.

7.1 Introduction

The HyDRa framework, *i.e.,* the modelling language (Chapter 4) and the generated conceptual API (Chapter 5) have been evaluated by users. These users, of different levels of expertise in terms of programming or knowledge of database modelling, were confronted with a scenario including (1) the reverse engineering of polystores, (2) the physical and conceptual modelling of this system, (3) the programming of representative functionalities, with or without the HyDRa API and finally (4) the evolution of these functionalities. This allowed us to obtain quantitative metrics on the written code, as well as qualitative comments via a questionnaire submitted to users.

In this chapter we present the context of the evaluation as well as the profile of the participants. Then we will detail the statement of the given exercise. We will then analyze the results and draw some illustrative examples of the problems and benefits brought by the framework. Finally we will analyze the comments given by the students on the two main parts of the framework, the design and the manipulation via the conceptual API.

7.2 Practical Context & Participants Profile

The participants who had the opportunity to use and give feedback on the HyDRa framework are mainly composed of two groups. The first group consisted of 36 master students who took the course entitled "Big Data: Engineering and Processing" at the University of Namur [1]. They carried out the exercise as part of the course assignments. The different deliverables were part of their final evaluation of the course. As a prelude to the exercise they received a 4-hour session introducing conceptual modelling, the NoSQL models supported by HyDRa, as well as a guide to the usage of mapping rules specific to HyDRa. The exercise was spread over four working sessions in class where two experts of the HyDRa framework were present to answer any questions. In addition, resources such as guides, documentation, videos or example cases were available on the HyDRa GitHub [53].

The second group of experimenters was composed of three doctoral students of the Università Svizzera italiana in Lugano who were following a two-day introductory course on database engineering. They had deliberately chosen to follow this course among several choices proposed. The work requested was evaluated in a "pass or fail" manner. They had similar resources to the first group of users. In the remainder of this chapter, unless otherwise stated, the discussions will be valid for both groups tested.

In Table 7.1 we find the age distribution of the participants, since the course is an integral part of the first master's degree program in data science in daytime, we find logically an overwhelming majority of respondents belonging to the 21-25 years old bracket. Then we asked them to evaluate their level of expertise concerning the different topics addressed by the HyDRa framework. They answered with a value between 1 (None) and 5 (Expert) for their knowledge of (1) the modelling and use of relational databases (SQL), (2) the modelling and use of NoSQL databases and (3) the use of the Java programming language. We identify that the participants are globally more experienced with relational databases, which is explained by the fact that many of them have benefited from previous courses and projects using them in academic lessons. Regarding NoSQL expertise, this is relatively low and none of the participants indicated a level higher than 3. Finally, experience in Java programming is more spread out and balanced with about ten people for levels 1 to 3, and six responses for the maximum levels 4 and 5.

In addition to this numerical evaluation of their own expertise, the participants also had the opportunity to provide a free text response in order to specify this expe-

[1] https://directory.unamur.be/teaching/courses/IDASM101/2021

< 20 years old	1
21-25 years old	31
26-30 years old	2
31-35 years old	2
36-40 years old	3

Table 7.1: Participants age repartition

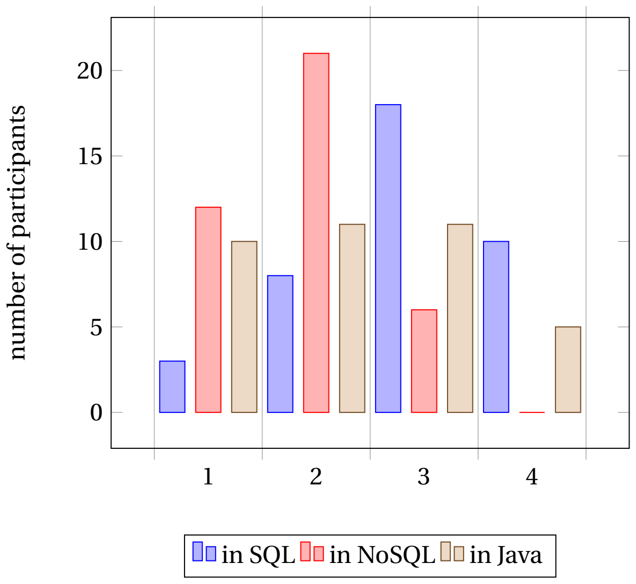


Figure 7.1: Expertise level of participants (1-None to 5-Expert)

rience. Among these answers, we can mention that seven people have professional experiences with relational databases and that almost all the respondents have only a very limited knowledge (notions) of NoSQL databases.

7.3 Exercise Design

In this section we will describe precisely the requested exercise as well as the main steps and deliverables to be provided in order to evaluate the work done.

The participants were divided into groups of four people for the UNamur students, the composition of these groups was made in order to balance the profiles of the students, indeed some had followed their bachelors in computer science while others came from the faculty of economic sciences, and could thus have a more limited experience of programming. The doctoral students from Switzerland, constituted two groups of two participants [2].

The main tasks to be performed were:

[2]One participant did not complete the exercise and therefore did not answer the final questionnaire.

- Model physical and conceptual schemas of an existing hybrid polystore (reverse engineering).
- Manipulate the data of the polystore by writing representative usage functionalities. These functionalities had to be written in two different ways, using (1) the native Java libraries of the databases (Section 1.2.1), (2) the generated HyDRa conceptual API.
- Evolve the HyDRa schema and the written code in order to make the written functionalities work on a new polystore implementing physical evolutions [3].

7.3.1 Modelling of existing hybrid polystore

Each group of students had a polystore deployed with the different databases containing data according to the conceptual schema in Figure 7.2. This conceptual schema repeats the case already presented in previous part of this thesis. It is based on the Northwind database [6], which describes an e-commerce system, with customers, orders, employees, products, etc.

These polystores were different from each other in terms of their physical schemas and the way data was distributed and structured in the databases. In order to create these ten different physical configurations, we used the data migration possibilities offered by the HyDRa framework (this complete process is described in Section 9.2). The different physical schemas used design construction permitted by the HyDRa modelling language. Constructs such as hybrid, mono or multivalued references, multi level embedding, structure split, or duplication. Tables 7.2, 7.3, 7.4 give the physical mappings of the conceptual elements represented for each of the polystores given to the students (named *S1* to *S9*).

On the basis of these polystores, the groups of students had to model the HyDRa schemas representing their assigned polystore. To do this they connected to the databases and browsed the data in order to transcribe the different physical structures encountered. Once the data was modelled, they were able to infer the corresponding conceptual schema. Finally, they established the links between these physical and conceptual elements and expressed them concretely in the mapping rules part of the HyDRa schema.

The HyDRa file representing the polystore marked the end of the first phase of the exercise.

7.3.2 Manipulating polystore data

After modelling the conceptual schema and the physical schemas of the databases in the given polystore, the second step is to query the data contained in this polystore. Each group has been assigned data query functionalities to implement. These queries are meant to be realistic and exploit the different complex structures of the polystore, the split structure or the hybrid foreign keys, so that each functionality requires one, two or three databases. These were expressed in a conceptual way :

- Retrieve a list of all *suppliers*.

[3]This task was only asked to the PhD students of Lugano.

Conceptual element	S1	S2	S3
Orders	Embedded in Customers + KV	DocDB	DocDB
Suppliers	DocDB	KV	DocDB
Products	RelDB	Rel	KV + RelDB (Structure split)
Shippers	KV (Structure split)	Embedded in Orders (DocDB)	KV (Structure split)
Customers	DocDB	RelDB	DocDB + Embedded in Orders
Categories	RelDB	RelDB	RelDB
Employees	RelDB	RelDB	DocDB
Region	RelDB	RelDB	RelDB
Territories	RelDB	RelDB	RelDB
composedOf	RelDB	RelDB	RelDB
works	RelDB	RelDB	RelDB

Table 7.2: Description of conceptual elements and their physical storage in each polystores

Conceptual element	S4	S5	S6
Orders	DocDB	RelDB	DocDB
Suppliers	DocDB	DocDB	KV
Products	KV + RelDB (Structure split)	KV + RelDB (Structure split)	RelDB
Shippers	RelDB	RelDB	RelDB
Customers	DocDB + Embedded in Orders	DocDB (with an array of references to orders)	Embedded in Orders + RelDB
Categories	KV	KV	KV
Employees	DocDB	KV	KV
Region	Embedded in territories	DocDB	DocDB
Territories	Embedded in Employees + RelDB	DocDB	DocDB
composedOf	RelDB	RelDB	Embedded in Orders
works	via embedding	RelDB	RelDB

Table 7.3: Description of conceptual elements and their physical storage in each polystores (continued)

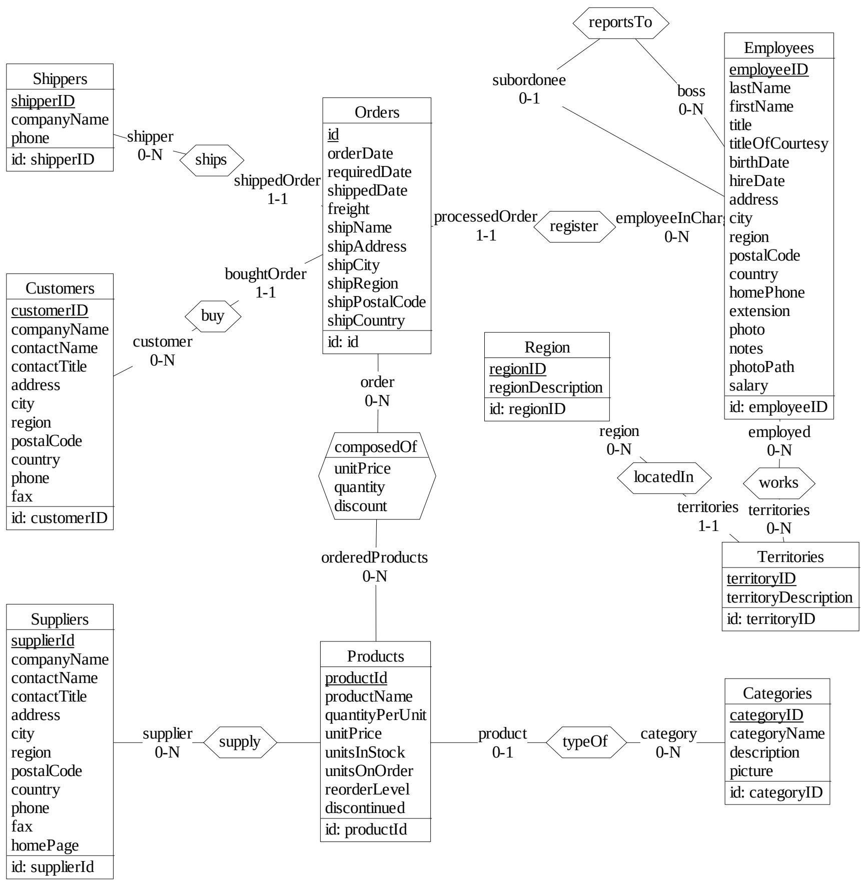


Figure 7.2: Conceptual model of student use case

- Retrieve the *supplier* of the product named "Escargots Nouveaux".
- Retrieve the list of all *shippers*.
- Retrieve *customers* having placed an order encoded by the employee named "Margaret".

As an example, applied to the polystore *S1* of the Table 7.2, the first functionality will only query the document database, whereas the second one will have to query the products in a relational database, and then query the document database before joining the results together. For the fourth query, if we take the polystore *S*5 of the Table 7.3, the three types of databases will be interrogated, since the employees, the orders and the customers are distributed on three different databases.

For each of these tasks, the final result had to be a list of conceptual objects (*i.e.,* a class containing the simple conceptual attributes of the requested entity type as specified in Section 5.3.1).

The implementation of these features had to be done in two different ways. The

Conceptual element	S7	S8	S9
Orders	DocDB	KV	DocDB
Suppliers	KV	KV	KV
Products	RelDB	RelDB	Rel
Shippers	RelDB	RelDB	Embedded in Orders (DocDB)
Customers	Embedded in Orders	DocDB (with an array of references to order)	KV
Categories	KV	KV	RelDB
Employees	RelDB	RelDB	RelDB
Region	DocDB	DocDB	RelDB
Territories	DocDB	DocDB	RelDB
composedOf	Embedded in Orders	RelDB	RelDB
works	DocDB	DocDB	RelDB

Table 7.4: Description of conceptual elements and their physical storage in each polystores (end)

first one by using only the native database access libraries, *e.g.,* Jedis [4], mongo-java-driver[5] or a JDBC driver. The second way is to use the conceptual API generated by HyDRa. The code of these respective implementations were the second and third deliverables of the student project.

Figure 7.3 shows the different components of the exercise at this stage, the elements in italics are the contributions that had to be written manually by the students.

7.3.3 Evolving polystore

Finally, a last part of the exercise was asked only to the Swiss doctoral students. It consisted in adapting the written artefacts to adapt to evolved physical structures of the polystores. To do this we gave each group access to another polystore containing data still conforming to the same conceptual schema but with different physical schemas. They also had a list of evolution scenarios which occurred on their first polystore in order to comply with the new target polystore. Below is an example of the given evolutions:

- Each supplier in the collection now contains a new array *products* containing the identifiers of their supplied products.
- The details of an order (its products) are now embedded in an array named *products* in the embedded *orders* array of the Customer collection.
- The products are removed from the relational database and are now stored in hash structure in a Redis database.

[4]https://github.com/redis/jedis
[5]https://mongodb.github.io/mongo-java-driver/

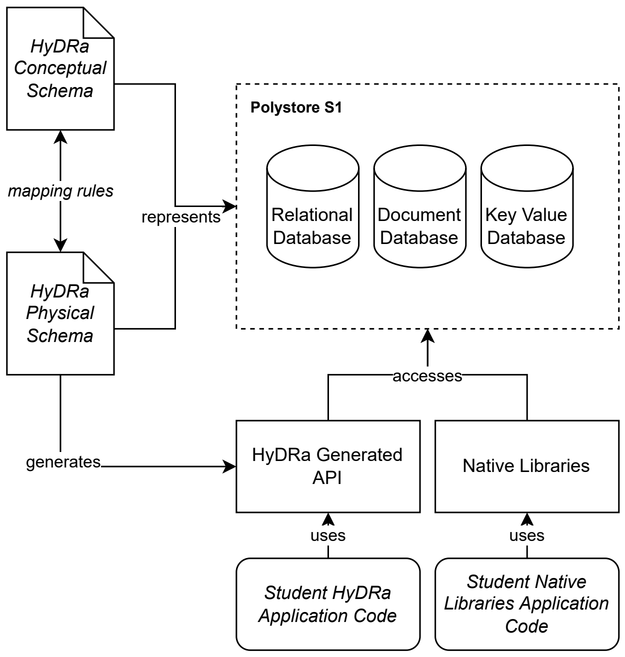


Figure 7.3: Component diagram of exercise

Figure 7.4 shows the components of the exercise by highlighting (in bold) the components impacted by the polystore replacement. It was requested that the data manipulation code written for the first polystore be re-run on the new polystore and return the same results. To do that they had to modify the :

- **HyDRa Physical Schema**. The schema is indeed different because it must reflect the data of the databases of the new polystore.
- **Mapping rules**. These new structures must be adapted in the existing mapping rules.
- **HyDRa Generated API**. The API must be regenerated so that the conceptual data access classes query the new structures.
- **Student Native Libraries Application Code**. The native code written by the students accessing directly to the databases must be adapted because it has become obsolete since these old structures no longer exist.

We can thus notice that the application code written by the students accessing the data via the use of the HyDRa generated API will not have to be adapted since it is at a conceptual level, the conceptual schema not having been modified, the available methods will be identical. Only the implementation of these methods is impacted, and will be automatically updated via the API regeneration.

7.3.4 Survey

Finally, each student had to fill in a questionnaire about the use of the HyDRa framework. The questions submitted and complete set of responses are available in

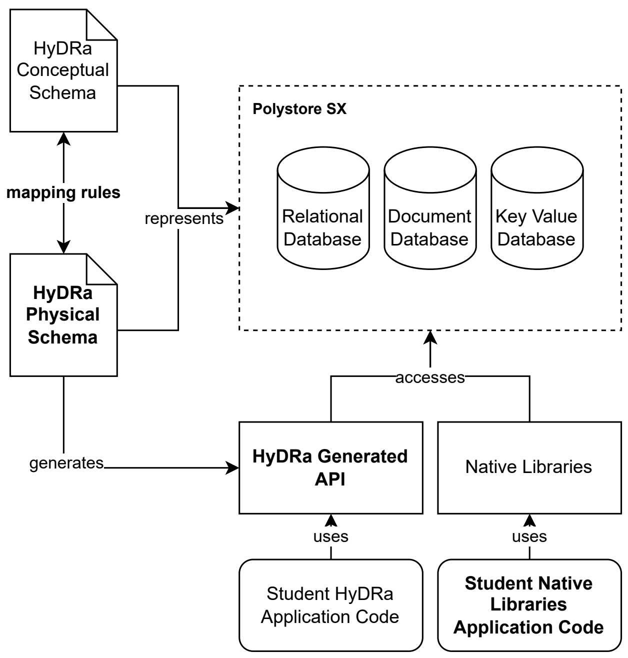


Figure 7.4: Impacted component of polystore evolution

Appendix D.1. The quantitative and qualitative results are described in the following sections.

7.4 Lessons Learned

In this section we describe the lessons learned from the analysis of the deliverables for each phase of the exercise.

7.4.1 Polystore schema produced

In addition to some problems of reverse engineering methods applied by the students, we were able to note through the analysis of the produced HyDRa schema files several difficulties specific to the HyDRa modelling language as well as to the Eclipse tool implementing it. These difficulties highlight areas for improvement of the tool, the language and the documentation of HyDRa.

The first recurrent error detected in several of the students' diagrams concerns the mapping of the relationship type roles to their physical representations. On one hand we found that the physical fields of complex types, representing nested structures, were not mapped to any role, while on the other hand the foreign keys/references were properly mapped to a role. This can be explained by the fact that the students were already familiar with relational modelling and that foreign key structures are a more familiar concept than nested structures. However, although the declared references were mapped to roles, the second recurring error was an inversion of the correct role to be actually mapped.

A third type of error occurred when the polystore integrated an entity structure split on the attributes (construction present in polystores S1, S3, S4, S5), in which case the students declared a reference between the identifiers of the physical structures concerned. This is neither necessary nor correct, because foreign keys are used to represent roles of relationship types, but in the case of a structure split, no relationship is involved.

These detected errors have allowed improvements of various types, which have been partly implemented in later versions of the framework.

- An additional validation rule can check that the cardinality and the type of the role are compatible with the mapped physical element (reference or nested structure).
- An improvement to the documentation concerning the mapping of roles.
- Suggestions for correct mappings can be provided.

7.4.2 Implementation of data access functionalities

Browsing through the code written by the students revealed several interesting cases. Many of them, in the code using only the native access libraries, were performing non-filtering queries and retrieving all the data from the databases in memory to finally filter the finer selection requested by the query in the Java code. Not only can this slow down the overall execution time, but it is also not a good practice since the system memory may not be large enough to hold all the data, which will eventually crash the system.

On the other hand, the code written using HyDRa made good use of the *conditions* objects which are automatically translated into native filtering queries. By these examples we can consider that HyDRa can help to build more precise selection queries.

Some groups guessed the identifiers for Redis data in order to do a direct get (hardcoded a value). Instead of extracting the identifier component by splitting the key using the pattern. This proves that manipulating Redis data natively without direct get on keys is not intuitive.

However offered by the generated API, few groups made use of the relationship specific selection methods. This led to a longer and more complex code instead of having the complete operation done in one line. This proves that the documentation of the code or the possibilities offered by the API were not properly exposed to better guide the user.

Table 7.5 exposes the numbers concerning the code manually written to implement the requested functionalities using the HyDRa conceptual API or using only the native libraries. For each way the table shows the number of files and the number of lines of code. The lines of code have been calculated by the tool *cloc* [6] and do not include blank lines and comment lines. These measures only take into account

[6]https://github.com/AlDanial/cloc

	HYDRA		**NATIVE**	
Group number	# loc	# files	# loc	# files
1	65	1	641	10
2	74	1	967	15
3	68	1	836	16
4	112	4	866	12
5	76	4	396[8]	4
6	74	1	758	6
7	55	1	370[8]	10
8	89	1	156[9]	3
9	75	1	565	9

Table 7.5: Manually written code, HyDRa vs Native libraries

handwritten code, so for the implementation using the API, all code generated by HyDRa is excluded from the line count [7].

The numbers in this table indicate that the size of the program to be written to fulfil the same functionalities is significantly smaller when using the generated conceptual API. This may indicate that the development time is also reduced.

7.5 Survey Feedback

After completing the various steps of the exercise, students were asked to complete a questionnaire that included questions with qualitative and quantitative answers. This section provides a table of the answers obtained for these two categories.

7.5.1 Quantitative results

The quantitative questions were formulated according to a Likert Scale, evaluating the satisfaction on a scale of values from 1 (Strongly disagree) to 5 (Strongly agree). The questions were separated into two main parts, those on the HyDRa modelling language, and those on the generated API.

HyDRa modelling language

Within the modelling part, the different sections of the language, *i.e.,* the conceptual part, the physical schema part and the mapping rules were the subject of specific questions. The combination of all these questions was generally related to the usability of the HyDRa modelling language, and they could be used to calculate the internal consistency of the questionnaire. According to Gliem [51] the interpretation results of a questionnaire including Likert scales, are only valid for a questionnaire including several questions and whose Cronbach's Alpha value lies in a precise interval. The general rule of thumb is :

[7]The number of lines for the API generated for an empty HyDRa schema is 2729

[8]These groups have not written the getters and setters on the attributes of the object classes.

[9]This group did not write the object classes.

Question	Mean
1 - I find the language in general was easy to use	3,54
2 - I think that most people would learn to use the language very quickly	3,62
3 - I would use HyDRa to design polystores	3,41
4 - I find the conceptual schema section of the language easy to use	4,23
5 - I think that most people would learn to use the conceptual schema language very quickly	4,05
6 - I find the physical schema section of the language easy to use	3,82
7 - I think that most people would learn to use the physical schema language very quickly	3,69
8 - I find the mapping rules section of the language easy to use	3,28
9 - I think that most people would learn to use the mapping rules section very quickly	3,10

Table 7.6: Quantitative question about HyDRa modelling language

- $\alpha > 0.9$ - Excellent
- $\alpha > 0.8$ - Good
- $\alpha > 0.7$ - Acceptable
- $\alpha > 0.6$ - Questionable
- $\alpha > 0.5$ - Poor
- $\alpha < 0.5$ - Unacceptable.

We calculated this value from the nine questions listed in Table 7.6. Using the IBM SPSS Statistic and Viewer software[10], the 39 results obtained for these nine questions were encoded, and the Cronbach's Alpha value obtained is 0.823, therefore the internal consistency is qualified as good.

The Table 7.6 gives information about the means obtained for each of the questions.

The remainder of this section presents results for individual questions.

Figure 7.5 shows the details of the answers on the questions concerning the usability (ease and speed of learning) of the HyDRa modeling language in general. The questions included are Q1 and Q2 of Table 7.6.

In Figure 7.6 is presented the detail of the results concerning the specific parts of the language. As can be seen in the table of questions asked, for each of these parts two questions were asked, one about the ease and the second about the learning speed of the language (Q4 & Q5, Q6 & Q7, Q8 & Q9). We were thus able to calculate the internal consistency of these questions on the usability of specific parts. We obtained respectively α values of 0.658 for the conceptual section, 0.832 for the physical schema part, and 0.911 for the section concerning the mapping rules. Considering the results obtained as well as the mean of answers of the questions, we

[10]https://www.ibm.com/spss

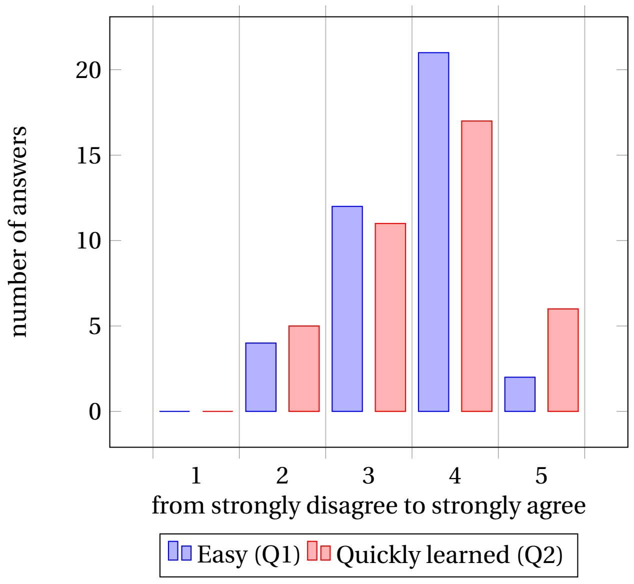


Figure 7.5: Usage of the HyDRa language in general

can conclude that the mapping rules section is considered the least usable of the three. Nevertheless, a score above three corresponds to "Neither agree nor disagree" in the Likert scale. These findings will be supported in the following section by the comments received via the qualitative questions.

HyDRa generated API

Concerning the use of the generated conceptual API, the questions asked also concerned the usability of this API, several questions of the same order were asked and could therefore be used together in order to calculate the Cronbach's Alpha variable verifying the internal integrity of the questionnaire. The questions used as well as the average of the answers are listed in Table 7.7. It should be noted that question 6 being the strict opposition of question 1, the values have been inverted when calculating the alpha variable. The result for the five questions related to the usability [11] of the API is 0.7, which falls into the "Acceptable" category according to the scale presented.

Figure 7.7[12] details the results concerning the use of the conceptual API com-

[11](Q5) was excluded from this calculation as it was not directly linked to the usability aspect of the API.

[12]Note that for those results there was an inconsistent response for one student where the free text response was the complete opposite of the rate given. For this particular case we adapted the number from 5 to 1 (Q6). The complete free text answer was the following : *Taking data from the database was faster and less tedious in Hydra. It took us several hours to connect and retrieve the data natively, which was not the case with Hydra. Hydra is thus easier to handle for data recovery than Java. The fact that three of us had never really coded in Java and that coding with Hydra took us less time to execute the different queries shows that Hydra is easier to handle to execute the different queries. Also, it takes fewer lines of code in Hydra to get the same results.*

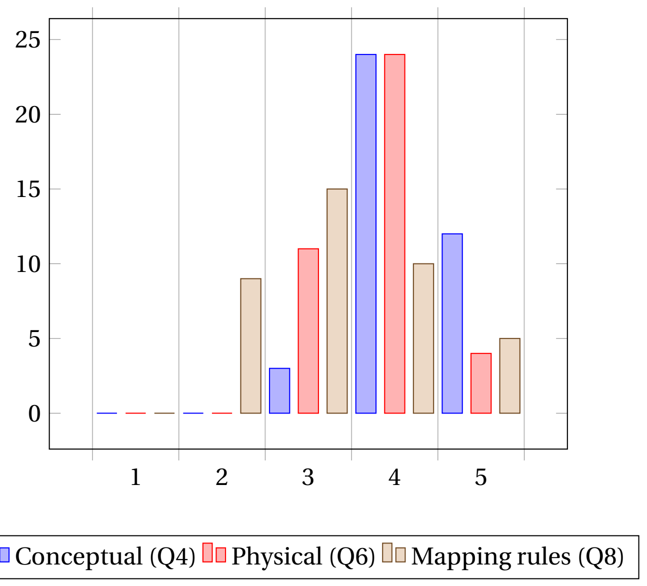


Figure 7.6: Usage of the specific HyDRa language part

Question	Mean
1 - I think that using HyDRa to access data is easier than using native Java code	4,34
2 - I find the database access code generated by HyDRa was easy to use	3,97
3 - I would use this generated code to access databases data in my future projects	3,24
4 - It was easy to find the method that I needed to access the data	3,71
5 - The data that I retrieved using the code generated by HyDRa was what I expected	4,37
6 - It is simpler to access data using native libraries than using HyDRa methods	2,1

Table 7.7: Quantitative question about HyDRa conceptual API

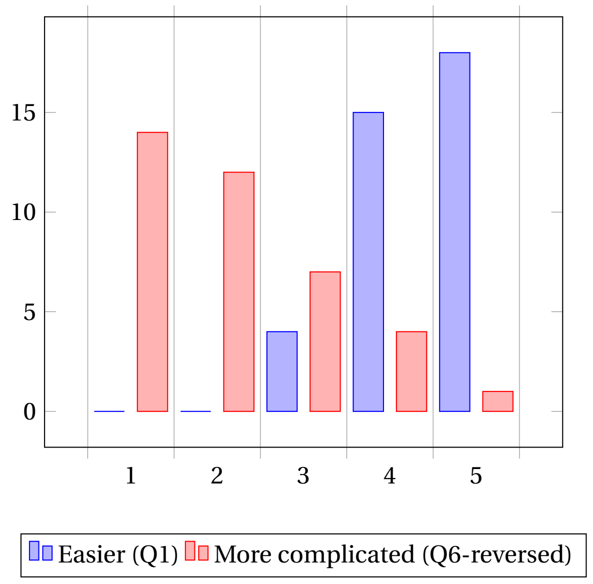


Figure 7.7: Usage of the HyDRa generated API compared to native libraries

pared to the implementation via the use of native access libraries. Thus, we can notice that the use of the conceptual API is considered simpler.

To finish the numerical analysis, Table 7.8 presents the answers given regarding their personal future use in other projects of the HyDRa framework. Either for polystore design tasks or for accessing data. 20 out of 39 answered "agree" or "strongly agree" for design, and 17 out of 38 for data access task.

7.5.2 Qualitative analysis

Here we will list the different characteristics that appear repeatedly in the students' responses to the qualitative questions. These characteristics will be illustrated with a quote. Some of these responses were originally in French and have been translated. In the online appendix are the original responses [4]. Again we divide this section according to the two main features of HyDRa, the modelling language and the conceptual API and then by presenting negative aspects followed by positive feedbacks.

HyDRa modelling language

The questions about the HyDRa modelling language were about the language in general as well as about the specific parts (conceptual, physical and mapping rules). For each of the questions we asked them to mention positive aspects as well as negative aspects. We finally ended with a question asking them to suggest improvements to the language. Below is a complete list of the qualitative questions asked:

- Mention positive aspects of the HyDRa modelling language

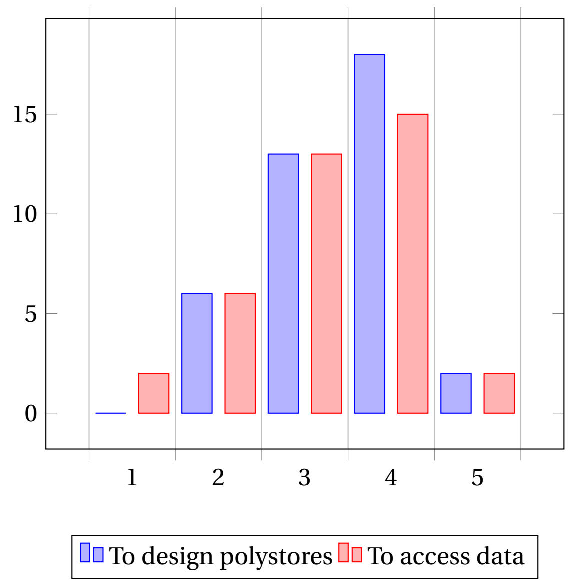


Figure 7.8: I would use HyDRa in my future projects

- Mention negative aspects of the HyDRa modelling language
- Which improvements regarding the HyDRa modelling language would you suggest?
- What are your positive or negative remarks about the conceptual part of the language?
- What are your positive or negative remarks about the physical schemas' section of the language?
- What are your positive or negative remarks about the mapping rules section?
- Do you have any other remarks?

Negative aspects and improvements The main criticisms made to HyDRa concern particular technical aspects constraints and bugs that hinder the good functioning and the handling of the tool. Some of them have been mentioned in the Section 5.7. The rule regarding the use of capitalization in conceptual attributes was a recurring complaint. Moreover, they disliked the use of the Eclipse IDE, which many students prefer to avoid.

Eclipse is not the best app and it has an old design aspect.

The most recurring item found among the responses is that students did not like to model using a textual language and would prefer a graphical interface. This was also a recurring item in recommendations of improvements. However this aspect is not completely related to the HyDRa language, but more to the lack of Computer-Aided Software Engineering (CASE) tools to help database development in which the HyDRa language can be integrated. Below are some example citations :

> *Graphical modeling will really be a plus for this language. In any case, doing it manually helped me a lot in the realization of phase I.*

> *Not easy to learn, only a textual language thus the representation in mind is hard and the mapping rules are also a bit hard to aboard.*

> *Text modelling is less intuitive than visual modelling (when modelling and also when you watch the model after). For modelling purposes only, I prefer to use conceptual and physical schemas. Also, write every column in conceptual, physical and mappings sections can be a bit boring.*

Some felt that the help of the teaching assistants during the exercise sessions was necessary to help with the installation or to clarify the documentation. These errors are nevertheless normal at this stage of the framework's development and are generally mitigated as the project matures.

> *As a developer, I have had the opportunity to use and experiment with a number of Java projects. However, the project documentation is not well provided in my opinion. All the cases seen during the course should be found in the GitHub project documentation. To bring precisions on how to configure its Eclipse. The management engineers of my group are totally lost by the absence of this information. Without the explanation seen in the course I would have been unable to produce correct mapping links (except die & retry).*

> *I think that the practical use of Hydra could be complicated because of the length of the implementation, and the particular tools to use. It would not convince any data analyst to use it instead of the classical SQL and NoSQL tools. It's normal that the use seems a bit rough since it's a development language, but I think that can be a big brake to its use.*

Others pointed the fact that conceptual modelling is required and may be a difficulty for bigger companies. However, we think that conceptual modelling is a needed part and common best practice in the database engineering process that should not be seen as an extra step required by HyDRa specially.

> *It needs conceptual schema so it can be harder to implement in bigger companies when they jump that step.*

Several remarks also concerned the mapping rules section, which confirms the feeling given by the metric values that this part is the least accessible to the user. In parallel, they recommended extending the auto complete in order to integrate all conceptual attributes at once.

> *The construction of mapping rules is very complicated when the cases considered are more complex, even if the logic behind it is similar to simple cases (tendency to make mistakes in the construction of certain references for example).*

> *More difficult to understand at first glance than the two schemas but easy to understand once the different examples have been studied in more detail.*

> *The mapping rules are complicated to understand and require a lot of focus and reflection to make right.*

> *Creating those was a bit more complicated for me. The rules can vary a lot depending on the type of relationships between tables, and especially in cases where a table is only used as a link, and therefore does not appear in the conceptual schema.*

And finally, several reported that the documentation was not sufficient or that the key value schema syntax was more complex.

> *Part on references are not well documented on the project.*

Positive aspects of HyDRa modelling language The most recurring positive items mentioned were the simplicity, comprehensibility of the language. Moreover, they spotted the usefulness of the examples provided in the documentation which allow a quick understanding of how the language works. Some also underlined the clarification brought by the separation of the conceptual and physical schemas as well as the interest of being able to model several types of databases. Finally, the positive contribution of the editor's auto-completion function and the implemented validation rules were regularly mentioned.

> *The conceptual part is easy to understand and apply to a real case. Translating from a conceptual schema to a HyDRa model is straightforward. Only the relational database paradigm knowledge is needed to understand the meaning of relations, in my opinion.*

> *The conceptual schema is very logical, if someone knows relational databases then he understands very quickly that first we create the tables via the entity, and then we create all the relations between the different tables, how they are connected together. With the help of a paper conceptual schema made in advance, the realization of the hydra conceptual schema is therefore simple.*

> *Easy to use when you have knowledge about NoSQL databases because the representation is almost the same as the representation in the native languages.*

> *No complex syntax, no unnecessary code, clear error reporting (most of the time), useful and simple autocomplete.*

> *Relatively easy to handle. Allows to "combine" different databases (SQL, Redis and MongoDB) with no additional constraints.*

> *Actually, I don't know any other tools that allow to integrate different kind of database. With a small example it is easy to understand how it works.*
>
> *Documenting separately the physical and conceptual schemas is its biggest strength IMHO.*

Others had the opportunity to understand more precise advantages of using the HyDRa modelling language in the process of data migration or inconsistency detection.

> *Types are tied to conceptual entities, thus introducing a single truth that should remain consistent through data migration processes of the physical schema of the polystore (inconsistencies should be matched fairly easily).*

Although the mapping rules section scored lower overall, some still considered their writing easy.

> *The mapping rules are the easiest HyDRa coding rules to understand, in my opinion. The mapping between tables can be done very easily for anyone with a bit of knowledge about databases.*
>
> *Once the conceptual schema and the physical schema have been created, the mapping step remains, which is the simplest of all since it is simply a matter of copying what we have written previously. This task is not very complex but the fact that you have to copy all the columns and not just the name of the table makes it time-consuming. I don't know if it's feasible but mapping the tables just by their name would be very useful.*

HyDRa Conceptual API

We will list the most recurrent elements and quote the most relevant and constructive feedbacks regarding the HyDRa conceptual API. Two questions were asked regarding the advantages and disadvantages.

- Mention positive aspects of the code generated by HyDRa
- Mention negative aspects of the code generated by HyDRa

Negative aspects Similar to the criticisms made against the modelling language, the conceptual API suffers from some shortcomings in the documentation provided. However, the remarks are more precise because it was for example raised that the provided methods were without comment blocks. This indeed gives an additional channel, *i.e.,* within the source code itself, to distribute the documentation.

> *Although the names are rather explicit, without comment it is sometimes not quite obvious to understand the principle of the method.*

> *Some methods, especially conditions methods, can be more complicated to use.*

> *Undocumented methods (naming is somewhat good but not enough IMHO).*

In addition, some people were surprised to discover the extent of the possibilities offered by the API by browsing the code, which had not been perceived through the documentation.

> *We don't necessarily realize that there are so many functions at our disposal, which makes us a bit lost when we start coding.*

Although there was a sense of being able to perform certain operations more easily, some had difficulty identifying whether a generated conceptual method existed.

> *For specific type of request it can be hard to find where the method is implemented. It's kinda hard to know if some requests can be done in one method or if we need more than one.*

Others pointed to difficulties with technical aspects of HyDRa, including the residual presence of bugs that prevented confidence in the logic of the written user application code.

> *It is difficult to know if the problems we are dealing with are bugs or a logic and writing problem on our part.*

Or the fact that the plugin is only operational in Eclipse.

> *The use of hydra over eclipse makes it less useful and nice to use because eclipse is getting old and ugly. (sic)*

Finally, we highlight two comments where the student takes into account the installation and modelling time involved before accessing the data, and puts this in perspective with the size of the system studied.

> *HyDRa generates a lot of code and methods that are not necessarily always interesting to use. By this I mean that I would use HyDRa for very large projects that require access to different data with different types of queries rather than for small projects.*

Positive aspects The positive point that comes up most often in student feedback is the ease of use and the time savings compared to using native libraries, even for users with very little experience in Java or NoSQL databases.

> *The use of the program is a real time saver and a simplicity (once mastered) that changes the game. Nevertheless, in my case, some were much harder in the realization of hydra, but I think that it is mainly due to my lack of knowledge in programming.*

Compared to native JAVA accesses, using HyDRa seems to be very simple for the tasks at hand. The code is relatively clear (even for me who never used JAVA), and allows intuitively to go and get the desired information. I think the real advantage is to think in conceptual terms, and thus to be able to detach oneself from the technique and the way the data are encoded, and where they are located. So you get away from that technical aspect with this preliminary phase of creating the polystore.

1. There is a way to retrieve the data you want easily, even if it is very specific. 2. Having generated code removes a lot of work, especially for connecting to different types of databases, automatic creation of models and services to access the data, ... 3. Offers a nice interface for data retrieval. Doesn't require us to know the specifics of the database connection libraries to be able to retrieve the data.

HyDRa is much more efficient than the native implementation because you don't have to make the database connections yourself. It is also easier to use HyDRa for the implementation because the code is more concise than the native implementation.

The undeniable advantage of HyDRa compared to using Java is that it is not necessary to take into account the types of databases from which the information comes. Also, for someone who doesn't have a lot of programming experience (mainly Python and SQL), the examples provided on the GitHub of the tool allowed me to easily get the hang of it.

Then even if gaps in the documentation were highlighted, many pointed out the consistency of the method names, making it easy to find them and understand their meaning.

The writing syntax is regular and logical. A certain pattern emerges

Names are clear / easy to search

First of all, I think it's not that difficult to learn. Secondly, the autocompletion helps a lot, especially when calling functions. The fact that you don't have to create the functions yourself is a great time saver and it also avoids errors. Moreover, even if the function name is long, I find that each name is explicit in relation to the purpose of the function and the use of capital letters at the beginning of each word makes it less difficult to read (e.g.: it is easier to read and understand "getProcessedOrderListInRegisterByEmployeeInCharge" than "getprocessedorderlistinregisterbyemployeeincharge")

And finally they have highlighted the potential of writing complex queries in a simpler way.

Complex queries can be done in one or two lines of code

There is also the condition package which makes queries much easier.

7.6 Conclusion

In this chapter we have detailed a scenario of use of the HyDRa framework by a set of more or less forty students. They had the opportunity to use the framework to model an existing hybrid polystore. They then manipulated the data of this polystore by writing two applications using (1) the native data access libraries, and (2) the HyDRa conceptual generated API. Some of them were also able to understand HyDRa in a context of evolution of the physical structures of a polystore. Finally, they completed a survey about their reactions to the use of the tool. This data was analysed and **quantitative and qualitative conclusions** were drawn.

Following the analysis of the polystore schemas created by the students, we could see that they were realized without major difficulties, and allowed us to identify more precisely the points that need a better documentation or adjustments of the editor. They also considered that writing the mapping rules was the most complex part of the HyDRa modelling language, but the ability to model several databases in a single file while being close to the native terms was appreciated.

As for the conceptual API, by analysing the code produced to implement the representative features, we were able to measure that the amount of code to be written manually is significantly smaller for the application code using HyDRa. This indicates a gain in time in the development of this code. The results of the questionnaire given by the students confirm this observation, both qualitatively and quantitatively that using HyDRa is considerably easier than using native libraries.

CHAPTER 8

SYSTEMATIC EVALUATION

Contents

In this chapter we describe the systematic evaluation performed checking the results obtained for a set of queries on 32 polystores. This allowed us to establish the correctness and measure the performance of the HyDRa conceptual API.

8.1 Introduction

In the previous chapters we have described and specified the HyDRa framework. Once the code generator producing the conceptual access library is implemented, it is necessary to perform checks to ensure that the correct results are returned for as many variations of physical schemas as possible. For this task we performed a systematic evaluation of the execution results of a set of queries (11) on a set of conceptual/physical schema and database type combinations (32). In addition, we performed these tests on a one GB dataset in order to gather indicative information about the framework's performance time.

The rest of this chapter describes the data, schemas and databases considered for this systematic evaluation. Then we describe how this helped us to establish the correctness of the developed code generator. We conclude with a description of the observed execution times and the lessons learned from them.

8.2 Experimental Schemas Data and Queries

In order to perform a large and comprehensive systematic experiment we have created 32 polystores combining the three types of databases (document, relational, key-value), as well as variations of conceptual and physical schemas. They include one-to-many, many-to-many, entity structure split, single or multivalued references.

8.2.1 Data

The data used for the tests come from Unibench[1] [134]. They generated data simulating an e-commerce system using multiple data formats. For example social network information was stored as graphs, purchase history as documents and customer information in relational tables.

For our tests we kept a subset of their conceptual schema, depicted in Figure 8.1. This schema consists of *Customer* who buy *Order*s, those orders are *composed of Products*. The size of the dataset is about 1 GB, Table 8.1 represent this data in terms of entity numbers. We then load it into the databases currently managed by HyDRa, *i.e.,* the relational, document and key value databases (similarly to the creation of the student's polystores, we used the data migration capabilities of HyDRa, *cf.* Section 9.2, to create our tests polystores).

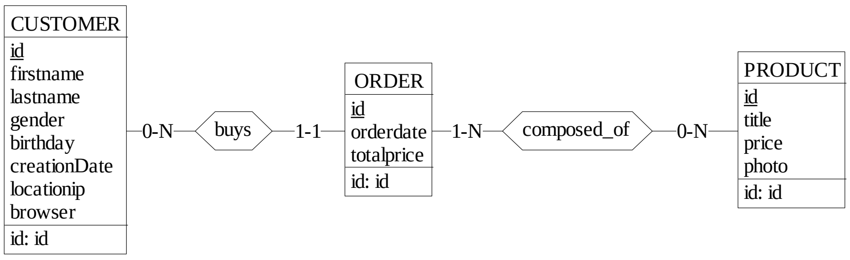


Figure 8.1: Conceptual schema of systematic tests data

[1]https://github.com/HY-UDBMS/UniBench

Conceptual element	#
Product	10 116
Order	142 257
Customer	9 949
Buys (relationship)	142 257
Composed_of(relationship)	693 910

Table 8.1: Number of elements in the dataset

8.2.2 Schemas considered

In order to test the selection queries generated by the API in a large set of possible scenarios, we have used the table of correspondences between conceptual and physical schemas described previously (in Section 5.2). The different conceptual configurations are again depicted in Tables 8.2 & 8.3. They represent respectively a single entity type, a one-to-many relation and a many-to-many relation. The three conceptual schemas give rise to nine different physical variations. Taking into account that each database ($DB1$, $DB2$ or $DB3$) can be of one of the three types of databases (document, relational or value key), this leads us to a set of 32 different configurations. Given the conceptual schema considered, CS (2) one-to-many relationship type is equivalent to the *buys* relationship between *Orders* and *Customer*, and CS (3) many-to-many relationship type refers to *composed_of* between *Product* and *Order*.

Each of the physical structures involved in one of the 32 possible polystore configurations has therefore been created and populated with the corresponding data. The 32 polystores schema files are available in the online appendix [2].

8.2.3 Queries

To test access times and correctness of data retrieval we have established several types of read queries. These are representative of selection queries that a user could perform, but they also represent core functionalities of HyDRa, such as the conceptual selection of a relationship type as well as on the basis of an attribute of a linked entity type.

- *Complete selection queries.* On the basis of a conceptual element (entity or relationship type) we wish to recover all the data (*e.g.,* read all the Orders).
- *Selection on the basis of an identifier.* This query gives an identifying value as input and the expected result therefore should contain only one object.
- *Selection on the basis of an attribute.* A value of an attribute is given and several objects are to be returned.
- *Selection based on an entity linked by a relationship.* These queries reflect the possibility of requesting entities linked to another according to the value of an attribute of the linked entity (*e.g.,* give the orders of a customer living at such address).

[2] https://github.com/gobertm/HyDRa/tree/main/be.unamur.polystore.unittests

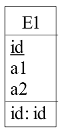
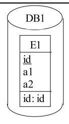
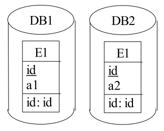
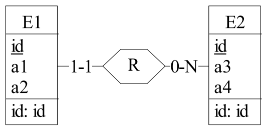
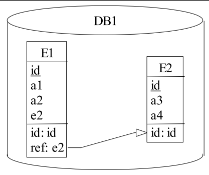
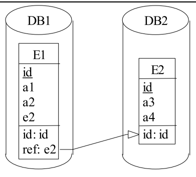
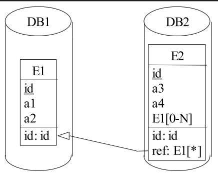
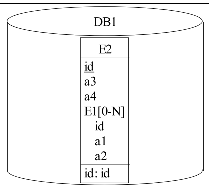
CS#	Conceptual Schema	Physical Schema
(1)		(a) 
		(b)  DB1 E1 id a1 id: id DB2 E1 id a2 id: id
(2)	 E1 id a1 a2 id: id 1-1 R 0-N E2 id a3 a4 id: id	(a)  DB1 E1 id a1 a2 e2 id: id ref: e2 E2 id a3 a4 id: id
		(b)  DB1 E1 id a1 a2 e2 id: id ref: e2 DB2 E2 id a3 a4 id: id
		(c)  DB1 E1 id a1 a2 id: id DB2 E2 id a3 a4 E1[0-N] id: id ref: E1[*]
		(d)  DB1 E2 id a3 a4 E1[0-N] id a1 a2 id: id

Table 8.2: Conceptual construction and physical correspondences

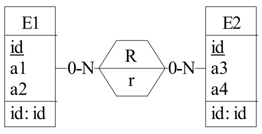
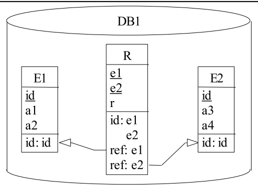
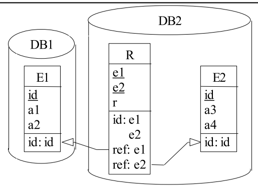
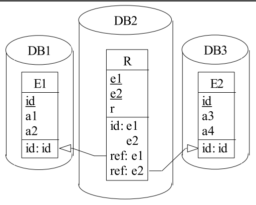
CS#	Conceptual Schema	Physical Schema
(3)		 DB1 E1 id a1 a2 id: id R e1 e2 r id: e1 e2 ref: e1 ref: e2 E2 id a3 a4 id: id (a)
		 DB1 E1 id a1 a2 id: id DB2 R e1 e2 r id: e1 e2 ref: e1 ref: e2 E2 id a3 a4 id: id (b)
		 DB1 E1 id a1 a2 id: id DB2 R e1 e2 r id: e1 e2 ref: e1 ref: e2 DB3 E2 id a3 a4 id: id (c)

Table 8.3: Conceptual construction and physical correspondences (continued)

Given the different conceptual constructs considered, these queries may concern different conceptual objects. The exhaustive list of the queries performed is the following : *getOrders()*, *getOrderByID(int id)*, *getOrderByOrderDate(Condition orderDateCondition)*, *getCustomerByOrderInBuys(Order order)*, *getOrderByClientInBuys(Customer client)*, *getBuys()*, *getComposedOf()*, *getOrderByProductInComposeOf(Product product)*, *getProductByOrderInComposedOf(Order order)*. The last two functions were tested using an argument criteria returning one or many elements.

8.3 Correctness

Performing these different data selection operations on these conceptual schema/-physical schema/database combinations allowed us to verify that the implementation of the code generator was correct for each of these variations.

The polystores considered are based on three different conceptual schemas, CS (1-3) Tables 8.2 & 8.3, we therefore have created three classes containing the execution code of the queries in the form of test methods. As these tests are conceptual, they are also valid, with the regeneration of the API, for all other physical schemas variations.

Listing 8.1 presents the application code of the test concerning the selection method of orders based on a condition on an order date *i.e., getOrderByOrder-*

Date(Condition cond). After having recovered the conceptual objects *Orders* (Line 4), we check that the number of elements contained is indeed the one expected (Line 5). We then check that each of the conceptual attributes is well populated (Line 6).

```
1      @Test
2      public void testGetByAttr(){
3          SimpleCondition<OrderAttribute> orderDateCond = SimpleCondition.
        simple(OrderAttribute.orderdate, Operator.GT, LocalDate.of(2021, 01,
        01));
4          orderDataset = orderService.getOrderList(orderDateCond);
5          assertEquals(85632, orderDataset.count());
6          for(Order o : orderDataset) {
7            assertNotNull(o.getOrderdate());
8            assertNotNull(o.getTotalprice());
9              //...
10         }
```

Listing 8.1: getOrderByOrderDate verification code

The overall process of checking the consistency of the generator implementation therefore includes the following steps:

- Generation of the conceptual API based on one of the 32 polystores.
- Execution of the test methods including the selection queries.
  - Call to the corresponding conceptual selection method.
  - Verification of the total number of objects returned.
  - Verification of the completeness of the attributes of the conceptual objects.

The rigorous verification of the results obtained for each of the executions of these methods for each of the polystores allowed us to identify implementation bugs and situations not covered in the algorithm under development. This crucial step has considerably improved the consistency of the returned results and the stability of the HyDRa framework.

8.4 Performance Evaluation

During the execution of the selection methods in order to establish the accuracy of the returned data we also recorded the execution time of these methods. The machine hosting the polystores and executing the methods of the HyDRa API was a common laptop[3], these measurements were taken only once, it is thus advisable to take these results with the necessary distance, they are given as an indication. However, these results reveal significant orders of magnitude that allow us to highlight some inefficient designs, as well as possible algorithmic and technological improvements in the implementation of the code generator.

As indicated in the previous sections we have tested the functionalities on 32 different polystores, Table 8.4 shows for each combination of conceptual schemas/-physical schemas the reference to the Table containing the details of the results for this configuration. Some of these results will be described in this section while the others are available in Appendix E.

[3]16GB RAM, i5 1.6 Ghz

	Physical Schema			
Conceptual Schema	a	b	c	d
(1)	Table 8.5	Table E.1	-	-
(2)	Table 8.5	Table E.2	Table E.3	Table E.4
(3)	Table 8.5	Table E.5	Table E.6	-

Table 8.4: Configuration and reference table results

Table 8.5 shows the results for all physical variations that involve only one database *i.e.,* all (a) physical schemas. Each of these schemas has been tested for each of the three supported database types. The queries included are the specific queries for each conceptual schema.

We can first notice that the execution time of a selection on a simple entity type (CS (1)) is not at all efficient (nearly 10 min of execution time) in a key-value database for the complete selection function (*getOrders*) and the selection based on a condition (*getOrderByOrderDate*). This is because these databases are not designed to perform selection queries based on values. There is no built-in mechanism that allow to perform this type of query. Unlike relational or document databases where the selection value is directly indicated in the executed query, key-value databases only allow direct access to a key-value pair using the key (Section 1.1.3). So, for HyDRa to retrieve all the stored orders, we have no choice but to (1) retrieve all the keys stored in the database, and (2) query each of them to retrieve their corresponding values. Then, in the case of condition criteria selection, (3) filter the data in the Java generated code. It is for this reason that execution times for large selection in key value databases are considerably longer. However, it is not the specific cause of the HyDRa framework, to support this we notice the response time of single selection method (*getOrderByID*) which is similar to other database types response time.

The second remarkable point is the longer time for operations involving joins in non-relational databases. This is particularly the case for CS (3), which is a many-to-many relationship with a join table. In fact, for document databases, these did not originally support native join queries [4], the implementation code generated by HyDRa does not exploit this type of query and therefore performs the joins in the application code (according to the implementation presented in Section 5.4.5). On the other hand, the execution time for these same queries on the physical schema (a) containing only relational databases is much smaller. This is because the implementation algorithm of the role selection methods (Section 5.5.4) has detected that all the databases concerned are relational, so only one SQL join query could be generated and executed. The information on the execution time during an application join allowed us to identify a performance problem blocking the production of results in certain cases. Indeed, in the case of the join on *composeOf* (which contains 693 910 rows), the previous implemented algorithm was not able to return results before several hours. We have therefore changed the join algorithm from a nested loop implementation to an algorithm involving hash structures.

[4]However, since version 3.2 of MongoDB the operator *$lookup* has been introduced.

CS#	Query	Relational	Document	Key Value
(1)	getOrders	1,36	8	600
	getOrdersByID	6,9	2,93	4,8
	getOrderByOrderDate	1,8	3,4	601
(2)	getOrderByClientInBuys	0,1	6,7	446
	getCustomerByOrderInBuys	0,82	0,8	23,9
	getBuys	3,77	7,25	448
(3)	getOrderByProductInComposeOf (one)	0,02	19,2	674
	getOrderByProductInComposeOf (many)	2,66	15,7	630
	getProductByOrderInComposedOf (one)	0,03	7,6	46
	getProductByOrderInComposedOf (many)	0,1	23,5	1647
	getComposedOf	7,9	18,7	-[5]

Table 8.5: Response time (in seconds) for single database physical schemas (a)

8.5 Conclusion

In this chapter we have presented the **systematic tests** performed with the HyDRa framework. We established an experimental environment with a 1 GB dataset, on different conceptual and physical configurations of polystores. Three conceptual schemas for nine different physical configurations to be run on three types of databases gave us **32 tested polystores configurations**. Each of these polystores was modelled using the HyDRa modelling language, then we generated the conceptual API for each of them.

Then a representative set of selection queries using the generated methods was tested. The tests included a check of the total number of instances returned as well as a completeness check of the returned objects. The error situations led to corrections of the implementation allowing a better coverage of the possible polystores managed by the HyDRa framework.

These different executions on several configurations allowed the achievement of two objectives, (1) the verification of the correct implementation of the API code generation, as well as (2) the identification of the performance difficulties of the API.

[5]This particular situation raises a bug that was not immediately fixed during the tests.

CHAPTER 9

USE CASES

Contents

In this chapter we present use cases of the HyDRa framework that go beyond its primary purpose. For example, it has been used to perform an administrative task and is now part of the business process. It can also perform data migrations between polystores.

9.1 Data Comparison

Within the context of a data analysis requested by the university IT service, the HyDRa tool have been used in a real business context. Here it is the use of the conceptual access methods generated that took precedence over the management of hybrid polystores.

Context

In an administrative department of the University of Namur, following the departure of a person performing a data analysis task manually, there was an urgent need to replace this manual task.

This request was received and after evaluation we concluded that the time saving linked to the use of the HyDRa framework allowed us to meet the request within the deadline. Moreover, we saw the opportunity to test the framework on a real use case database.

The requested data analysis consisted of comparing the data related to the courses given at the university between two distinct academic years and thus identifying and listing all the differences for each of the characteristics involved in the course programs.

Process

The entire process to arrive at the desired outcome result file involved the following tasks:

- Reverse engineering of the database.
- The modelling of the conceptual schema and physical schema of the database in HyDRa.
- The generation of the conceptual API code.
- Writing the code for the data comparison application using the generated API.
- Production of a result file.

Modelling

For the first reverse engineering task, this was done by combining a data mining approach with interviews with expert users of the application domain. We were thus able to produce a conceptual schema of the application domain. Figure 9.1 represents this schema, which is restricted to the information necessary for the analysis of the requested data.

The two main concepts of the application domain are the *Program* and the *Course*, they represent a course program, which contains several courses followed by students. These courses or programs are organized by academic year and may have particular characteristics for a specific year (*ProgramOrganization* and *Course-Organization*). Each program per academic year also has two *Person* who are the program secretary or chair. The concept of *Allocation* links together the programs in an academic year with the course organized for that academic year. An allocation also has its own attributes and is linked to a general subject group *SubjectGroup* (*i.e.,* a category).

After conceptualizing the database, we were able to model it using HyDRa (the complete schema is listed in Appendix C.1) and generate the conceptual API to manipulate the data using simple conceptual methods.

Application code

This allows to easily create an application program that performed the requested data comparison without having to master a database query language and perform manual joins. This application can also be reused from year to year. In Algorithm 5, we detail in a simplified way the algorithm of the application code allowing the comparison of courses between academic years.

The courses of an academic year being organized by program, the first phase consists in recovering all programs (generated conceptual method *getProgramList()* was

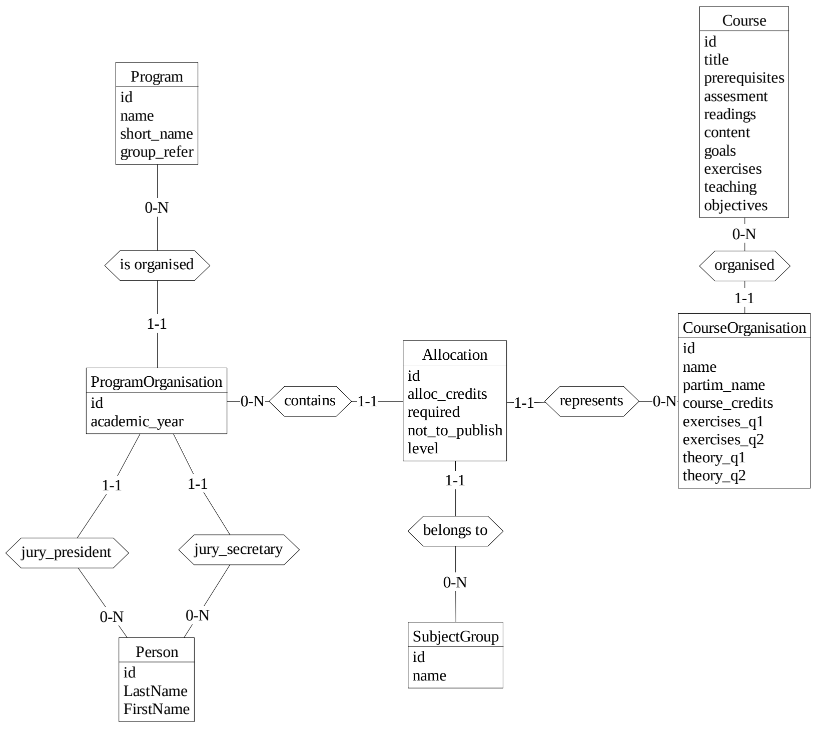


Figure 9.1: Conceptual schema of courses data

used for this) and iterating on each of them in order to recover the two *ProgramOrganization* of the considered academic years. This retrieval uses a role selection method exploiting the *organization* relationship between *Program* and *ProgramOrga* (*getOrganizationListInOrganizationByProgram(Program program)*). If nothing is retrieved for any of them, we have detected an addition or non-organization of a program.

The second phase then consists in comparing the courses of a program from one academic year to another, via the *Allocations*. These are also recovered using HyDRa methods (*getAllocationListInAllocationProg(Condition<ProgramOrga> progOrga, Condition<Allocation> condAlloc), getCourseOrga(...), getCourse(...)*). If the *Courses* is the same as for two allocations, it means that we have found two identical courses for the same program, organized on two different years. We can then go further, and compare these two allocations with each other. As HyDRa methods return conceptual objects that are easily manipulated, this comparison consists in comparing the attributes between them using simple *getters* functions. Finally, once again, if one of the allocations from one year to the other does not match, it means that we are facing an addition or a deletion of courses between two years.

The last step of the application program is to produce a readable file of the identified differences.

Algorithm 5 Simplified comparison algorithm

```
$programs \leftarrow$ getProgramList()
for $p \in programs$ do
  $programOrgaY1 \leftarrow$ getProgramOrga($yearOne$,p)
  $programOrgaY2 \leftarrow$ getProgramOrga($yearTwo$,p)
  if $programOrgaY1 = \emptyset \lor programOrgaY2 = \emptyset$ then
    Program $p$ has been deleted or created
  end if
  $allocationsY1 \leftarrow$ getAllocation(programOrgaY1)
  $allocationY2 \leftarrow$ getAllocation(programOrgaY2)
  for $allocY1 \in allocationsY1$ do
    $found = false$
    for $allocY2 \in allocationsY2$ do
      if $allocY1.getCourseOrga().getCourse() = allocY2.getCourseOrga().getCourse()$ then
        $found = true$
        compare($allocY1$,$allocY2$) # This is where we compare two Allocation objects and produce results if modification
      end if
      if $\neg found$ then
        An allocation of course has been removed in the program $p$
      end if
    end for
  end for
end for
```

Benefits

The result of this work has allowed us to highlight the possibility and simplicity of using generated conceptual methods to access data and the time gained compared to writing a classical data analysis program. The application written for this use case has greatly facilitated and accelerated the task of the requesting service and is now part of the year-to-year process.

9.2 Data Migration

As we have already mentioned in Chapters 7 and 9, HyDRa was used to perform data migration from one polystore to another. This has greatly facilitated and accelerated the creation of multiple polystores based on the same conceptual schema.

Indeed, for a single conceptual schema there may be several physical representations of the data, this is particularly true in the context of a polystore. As these different representations can have a significant impact on query performance, a developer may need to test several of these representations. Thanks to the HyDRa framework this task of migrating or loading data on different physical representations is considerably easier. For this, HyDRa offers four data migration strategies

:

- Native reading and writing
- HyDRa reading and native writing
- Native reading and HyDRa writing
- HyDRa reading and writing

Native reading and writing The first migration strategy consists of manually writing a migration program or scripts for each of the desired representations. The source data is read from files or databases and then written to the target databases using native code corresponding to each desired target structure. This is very time-consuming and requires the developer to know the languages of each of the technologies involved. Furthermore, there is no flexibility as the written program is only usable for the written configuration. In this migration strategy, to create the ten polystores of the user evaluation, ten different programs would have been needed.

HyDRa reading The second strategy, depicted in Figure 9.2 combines the use of HyDRa for reading data and the use of native access libraries for writing. The reading of data is done from existing databases that have been represented in a HyDRa model. Once the API has been generated it is possible to read the data using the conceptual methods offered. This brings two advantages, (1) the user can abstract from the source database query language and (2) the reading code can be reused on another source database. For example, in the scenario of a data migration from a relational database to a document database, using this strategy allows the user to abstract from at least one query language *i.e.,* he will not need to write SQL queries. However, the writing of data must still be done via the use of native queries. In case several polystores are to be created, this strategy still requires the writing of several corresponding programs.

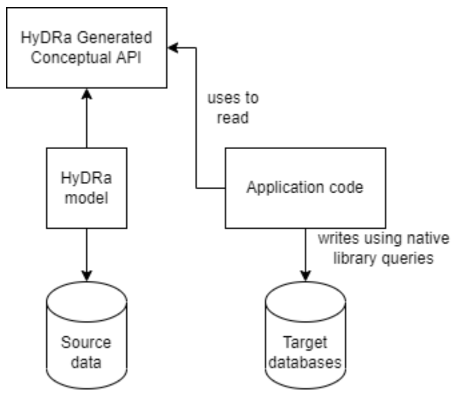


Figure 9.2: HyDRa reading and native writing strategy

HyDRa writing The third strategy shown in Figure 9.3 uses HyDRa conceptual methods for writing data to the target polystore. The reading is done manually from any sources, *e.g.,* csv files or databases. The target polystore is modelled in a HyDRa model. Once the API has been generated, it provides conceptual object classes and conceptual insertion methods capable of inserting the data into the corresponding mapped physical format. (See Chapter 5). The user-written migration program must therefore encapsulate the read data into the corresponding conceptual objects and use the associated service insertion methods. This way the migration program does not have to be adapted to create another polystore with a different configuration. The only required step is to model and regenerate the API of the new polystore's corresponding HyDRa schema. It is this method that was used to create the user evaluation polystores where a single application code file is needed.

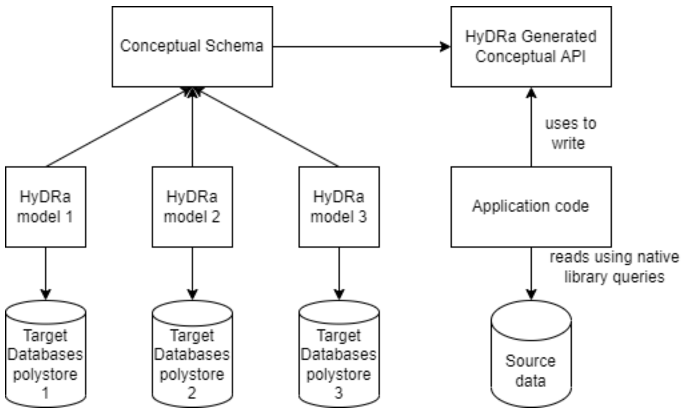

Conceptual Schema
HyDRa Generated Conceptual API
uses to write
HyDRa model 1
HyDRa model 2
HyDRa model 3
Application code
reads using native library queries
Target Databases polystore 1
Target Databases polystore 2
Target Databases polystore 3
Source data

Figure 9.3: Native reading and HyDRa writing strategy

HyDRa reading and writing The fourth and last method of creating or migrating polystores (Figure 9.4) is the HyDRa read/write data strategy. To use this approach it is necessary to model the source and target physical schemas in the same HyDRa file. The conceptual mappings must be mapped to the two structures concerned, *i.e.,* source and target databases. In this way the generated API implementation will integrate for each conceptual object both the source and the target database. The migration program written by the user consists therefore only in the use of the generated HyDRa methods. However, in order to avoid unnecessary reads and writes in the source or target databases, the generated methods to be used in this program are not located at the conceptual level but at the logical level (the code will use the database specific method generated, *cf.* Section 5.4.1). This implies that the written program is not fully reusable without modification for creating multiple configurations, but it is still fully independent of native database code.

Summary Table 9.1 summarises the different characteristics of the strategies presented. The column "Generic Migration Code" indicates if the program that the user writes to perform the migration is applicable to more than one physical configuration. The column "Needs native code or queries" specifies if the migration code will have to contain code reading or writing data with another library than HyDRa. "#

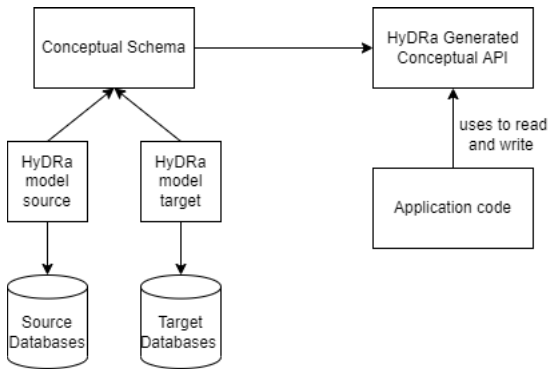


Figure 9.4: HyDRa reading and writing strategy

Hydra model" indicates how many HyDRa schemas files must be written in order to perform a migration. The columns "Source Data" and "Target Data" specify the format of the data needed to apply this strategy, "Any" indicates that the data can be stored in any format, *e.g.,* files or databases. "HyDRa dbs" specifies that the data must be stored in a database supported by the code generated in the HyDRa API.

Strategy	**Generic Migration Code**	**Native Code**
Native reading and writing	No	Yes
HyDRa reading / Native writing	Reading	Yes
Native reading / HyDRa writing	Writing	Yes
HyDRa reading and writing	Partially	No
# HyDRa model	**Source Data**	**Target Data**
No	Any	Any
One for source data	HyDRa dbs	Any
One for each target polystore	Any	HyDRa dbs
One	HyDRa dbs	HyDRa dbs

Table 9.1: Summary characteristics of data migration strategies

Data migration strategies using HyDRa have distinct advantages depending on the context of the migration. The number of physical schema variations is probably the most important factor, as a single program written using the HyDRa conceptual API represents a significant time savings and ease of use.

This migration functionality, its ease of use and the flexibility offered towards the target physical structures was a determining factor in the implementation of the student polystores and thus in the conduct of the evaluation presented in Chapter 7.

Part IV

Conclusion

CHAPTER 10

CONCLUSION & FUTURE DIRECTIONS

Contents

To conclude, we will return to the research questions posed at the beginning of the manuscript and provide the answers developed throughout this document. We will describe what we did, what it brought and we will end with what we will do, thanks to the opened perspectives.

10.1 Summary of the Contributions

RQ 1 : ***What are the problems developers face when testing database manipulation code and what solutions are recommended by the community?***

Our first research question, addressed in Chapter 3, sought to analyse the importance of testing database access code, this part of the code being particularly sensitive and error-prone in a data intensive system.

Through the analysis of the data access code as well as the test code of 72 open-source Java systems, we were able to **calculate** that 46% of them tested barely half of the access code and that 33% did not test it at all.

Based on this observation of a lack of test coverage, we sought to identify the problems that developers were encountering. To do this, we analysed more than 500 questions on database code testing from sites of the StackExchange network. We extracted a **taxonomy of 83 problems divided into 7 categories**, with a preponderance of general questions asking the best practices for testing this code as well as technical questions about database management.

Finally, we analysed and tagged 598 answers to the questions in the category *best practices*. We also built a **taxonomy of 363 recommendations** and 9 categories. The main recommendations concern the configuration of an environment and the management of test data.

From these results we could outline the main axes of research concerning the testing of data access code. Moreover, as these results are public in a **replication package** [57], they can also be used as a best practice guide for developers writing tests.

The conclusions of this research question were therefore an additional motivation for the development of the solutions proposed in the rest of this thesis. The generation of this data access code, manipulating the data at a conceptual level, constitutes the two approaches we have chosen to explore in order to minimize errors in this code and to help in the writing of these tests.

RQ 2 : ***How to build a conceptual and physical modelling language able to keep specific data model constructions?***

Chapter 4 presents a **modelling language** (**HyDRa**) of *hybrid polystore* composed of three main parts, a first one containing the conceptual schema, a second one with the physical schemas and a third one including rules of correspondence between the conceptual elements and the physical elements.

The conceptual modelling allows to represent the application domain of the polystore system. The language used is based on the Entity Relationship (ER) model, an existing model commonly used in the classical database engineering process. The use of this language has the advantage of not introducing any new concept to be understood by the designer.

The physical section of the HyDRa model is a language allowing to describe the structure of the data stored in the polystore databases. The supported databases are of several data models, *i.e.,* relational, document, column, key-value or graph. It uses a terminology close to the specific models in order to facilitate its learning. The developer can thus specify data both at the level of structures (tables, collections, nodes, etc.), called *Physical Structure*, and at the level of precise fields (*Physical Field*). Finally, the definition of *References* authorizes the representation of links between fields of different, possibly heterogeneous databases.

Finally, there is the section of mapping rules establishing the correspondences between the conceptual elements (*e.g.,* type of entities, attributes, type of associations, roles) and the specific physical elements (*i.e., Physical Structure* or *Physical Field*).

It is this physical breakdown and the low-level mapping rules that offer the possibility of integrating the different existing NoSQL design approaches as well as the flexibility to create a wide variety of physical configurations for the same conceptual schema. These variations can also include data duplication, cross-database references or structural separation of entity types.

RQ 3 : ***To what extent can we use this model to generate data manipulation code at the conceptual level independent of any physical storage data representation?***

Via Chapter 5 we describe a Java code generator allowing to **manipulate conceptually the polystore data**. This generator takes as input a HyDRa schema as specified in Chapter 4 in order to produce Java classes representing on the one hand the **conceptual objects** (*e.g.,* entity types and relationship types) and on the other hand **service classes** manipulating these objects. The manipulation of data via the service objects takes place at the conceptual level with functions such as *getEntityList()* or *insertEntity(Entity e)*. These methods are the entry point to the data used by a developer, the names of these methods being derived from the names of the conceptual schema, they are an intuitive way to query a polystore. Moreover, since the implementation code of these service classes is generated, the developer gains in ease and time of development of the final application. Indeed, in this way he is (1) exempted from knowing the exact location and physical structure of the queried data, (2) freed from the need to know the query languages of the specific databases hosting the data, (3) does not have to write the join code in case of queries involving several databases, etc.

To manage this, the generator code must be able to identify and produce the necessary code allowing (1) to identify the databases involved, (2) to build the corresponding queries in native language, (3) to make the joins of the results and to identify the potential data conflicts, and (4) to reconstruct the conceptual objects on the basis of the recovered physical structures.

The HyDRa modelling language as well as the generator have been developed as an Eclipse plugin that can be easily installed and used thanks to the guides provided [53].

RQ 4 : ***How can we support the databases' evolution of a polystore with regard to their impacted artefacts?***

Database evolutions in a polystore system can be of different types and involve several components of the system. An evolution can increase or decrease semantics, *i.e.,* it adds or removes elements of the system's conceptual schema, or it can be structural, *i.e.,* it involves only a change in the data structures in the logical/physical schemas.

In Chapter 5 we have proposed a conceptual API for data manipulation. The use of this API as *data manipulation code* in the user application allows the support of structural evolutions. Indeed, this type of evolution will only require a modification of the physical schemas and the mapping rules. Once the evolution is reflected in the HyDRa schema representing the polystore, **an API code regeneration will be enough to manage the co-evolution of the application code**. This is achieved thanks to the preservation of the method names by the API, which are based solely on the conceptual schema, which remains unchanged.

After having developed a solution for managing structural evolutions for the application code, we have proposed in Chapter 6 a solution for adapting queries to semantic evolutions. Based on a polystore model including operators specifying the semantic evolution to be performed, as well as a set of queries manipulating this polystore, we have developed **a tool allowing to adapt**, if possible, these queries to the new configuration, and if not possible, to notify the impact on the result.

Finally, as our first two contributions only concerned the co-evolution of programs or queries, we finally proposed a **theoretical evolution framework** to integrate all the artefacts to be modified in a system. The evolution of databases becomes a top-down process starting from evolution operators to impacted artefacts (*e.g.,* the schemas, the structures, the data, the programs). These operators are included in an impact matrix, which given the artefact of the system, the operator and other determining dimensions (*e.g.,* the data models, the modelling strategy, the migration strategy *etc.*) indicates the operations to be carried out to fulfil the evolution.

RQ 5 : ***What is the performance, usability and usefulness of the developed solution?***

We have evaluated the modelling language and the HyDRa code generator through two separate evaluations. Chapter 7 describes a scenario of the **framework usage by a set of students**, they were asked to perform a modelling of a polystore in a reverse engineering context using HyDRa modelling language. Then they had to use the generated methods of the API to query the data. Following this experience, we collected quantitative and qualitative data on the use of HyDRa, which shows that they positively evaluate the usefulness and the ease of use of the framework, despite some shortcomings in the documentation. Moreover, we found that the manually written application code is significantly less than the code written without HyDRa.

In a second experiment we **tested the performance and correctness** of the generated API by setting up dozens of different polystores to answer a series of representative requests to cover a maximum of situations. Thanks to these tests we were able to correct the implementation of the generator and identify the response time problems.

Finally, the Chapter 9 also showed that the developed solution could be used in real scenarios in order to perform a data analysis, or polystores migrations and loading.

10.2 Future Challenges

HyDRa framework security An important aspect of any information system and especially of systems using data is the security aspect. This was not given much attention in the initial development of the HyDRa framework, which is why it is an additional area of work for the future. In this section we will elaborate on the potential security problems, their solutions, and their application within HyDRa.

According to Denning [44], the data security in a system depends on four safeguards. The first is access control, which controls which users have access to which parts of the system and also to which data. The second is data encryption, which prevents the data from being read and manipulated by malicious interception. The third is the protection of the flow of information, which ensures that information of a defined security level cannot, through various operations (copies, transfers, etc.), end up in the hands of a user who does not have the required security level. And finally, the fourth safeguard, concerns inference control, which applies to statistical databases in order to avoid circumventing the confidentiality of the data with queries allowing the inference of information.

These four facets of data security have been established for several decades and are detailed in a reference work on cybersecurity [45]. In addition, other authors [133] have noted that the advent of Big Data systems brought additional challenges.

In the rest of this section we will however focus on access control and encryption applied to the HyDRa framework. Although we are aware that only a strategy integrating all dimensions can guarantee maximum security.

In HyDRa we have identified several moments where an access control mechanism (based on a previously established Role Based Access Control (RBAC) matrix) can be implemented:

- Before the generation of the API code. A user authentication can be requested before generating the access library in order to prevent a user from disposing of it.
- At use. Each call to the generated methods of HyDRa can be subject to an authorization check in the matrix before its execution.
- At the querying of the data. Once the execution is authorized, we can finally add an access control to the data, in the databases themselves. Indeed, each database managed by HyDRa also includes its own access control.

Adding an encryption to these access controls allows protecting against identity and data theft. This encryption can also occur at different points:

- The modelling file as well as the file containing the credentials of the databases.
- The communication between the API and the databases.
- The data. Using the native encryption mechanisms of the databases.

Finally, a last point of attention concerns the generated API code. Indeed, it is currently generated in the form of a library of classes in textual format. If this library were to be shared with users whose access to the data must be controlled. It will also be necessary to protect it against possible alteration. Indeed, a user by modifying the code has the possibility to perform request injections and alterations. In order to avoid this, the API can be generated only in binary format that cannot be modified.

The combination of access control and encryption at these different points allows an end-to-end control of the authentication and integrity of the code and data in the HyDRa framework.

Improve performance and big data scalability The performance tests presented were performed on a single machine with a reasonable size dataset. These have already highlighted some weaknesses of the framework and relatively long response times. Given the vocation of NoSQL databases to manage large volumes of data, a crucial aspect of the future development of the framework is its ability to manage these volumes.

There are several possibilities and areas of development that will improve performance and that may eventually allow HyDRa to manage large volumes of data and step up its game to be considered as a *big data tool.*

A first approach is to modify the specification and implementation of the selection methods, which currently require retrieving all the data from the underlying databases in order to return a list object containing the results. In addition to being inefficient, this technique makes the system vulnerable. Indeed, if the results returned by the databases are too large for the memory, the application will crash. To get around this, the methods may rely on streaming techniques, both for reading the database and for the returned objects. Streaming avoid creating a complete list in memory and return the results on-the-fly, however other constraints and difficulties will appear, notably in the management of joins between databases.

A second possibility involves using big data reading frameworks. Such frameworks implement data reading in a distributed manner and natively integrate optimization features such as caching, lazy loading, parallel processing, *etc.* A framework such as Apache Spark[1] is one of the most popular and promising beginnings of integration into HyDRa have been tested.

Finally, a shift to a client-server architecture can bring improvements in performance, scalability, availability and security. HyDRa is an Eclipse plugin generating Java code, which can be used in a client application as an import package. These packages can be encapsulated in a REST type service and hosted on a server. This way a client application will call the conceptual methods via this service. This allows to take advantage of the server's computing power in order to calculate the results. This also allows for resource monitoring, the addition of centralized access control, increased caching capabilities, *etc.* Moreover, this encapsulation opens the door to hosting and executing the generated API in a cloud computing environment such as Microsoft Azure or Amazon Web Services. These infrastructures offer many possibilities and can certainly help to make HyDRa a more powerful tool.

Transaction support HyDRa in its current version does not consider transactions, meaning that during an insert operation involving several databases, if the insert fails for one of them it is possible that the other databases involved have already performed the operation, leaving the polystore in an inconsistent state across all structures. This is not acceptable in a professional solution.

This is why it is necessary to group operations that concern several structures into one transaction. Transactions are operations that respect the Atomicity, Consistency, Isolation, Durability (ACID) properties. Atomicity, for example, guarantees that for a transaction, all the operations composing it are either all performed on all the databases, or none of them are performed at all.

Transactions occur at two different levels, they are either local, *i.e.,* specific to particular databases, which implement the necessary mechanisms to guarantee the ACID properties. Or they are distributed, *i.e.,* the ACID properties must be guaranteed across several resources of a system. For this purpose one can use the Two Phase Commit (2PC) protocol which coordinates the different resources involved by sending *prepare, commit* or *abort* messages.

[1]https://spark.apache.org/

The implementation of 2PC requires two conditions, (1) the databases or resources involved must support local transactions and (2) they must understand the XA protocol, which specifies the 2PC messages. The use of the 2PC process requires a Distributed Transaction Coordinator (DTC), which synchronizes the actors involved in a distributed transaction, there are several implementation libraries such as Atomikos, Bitronix or Narayana. In order for HyDRa to guarantee the transactions, Atomikos code encapsulating all the manipulations on the different physical structures (*i.e.,* in the database specific methods of Section 5.4.1) can be added in the generic code generated from the high level conceptual methods (Section 5.3).

The databases supported by the HyDRa API support local transactions, MongoDB[2], Redis[3] et MySQL[4]. However, Redis and MongoDB do not implement the XA protocol, it is necessary to implement it in a manual way, by relying on the operations of the local transactions and by using the library of the chosen DTC (*e.g.,* Atomikos library and implementation of the interface *Synchronization*).

Note that in case a future version of HyDRa handles databases that do not support local transactions. Integrating it in a 2PC process will still be possible, however the management of the local ACID properties will have to be implemented manually using for example a compensating transactions mechanism [121].

Add support of object-oriented design in conceptual model The conceptual model supported by HyDRa only integrates the simple elements of the Entity Relationship (ER) language (Section 4.3), which means that it is not possible for a designer to express inheritance relationships. However, in the context of generating application code in an object-oriented language, the absence of these constructs can be a real obstacle to the adoption of a tool such as HyDRa. Indeed, as it stands in HyDRa, an application domain including the concepts of *Person*, from which inherit other entity types such as *Employee* and *Worker* will have to be modelled using three distinct entity types, without any explicit inheritance relationship. This will result in separate object classes and services. The developer who wants to manipulate *Person* objects containing both workers and employees will therefore be forced to call specific selection methods on both *Employee* and *Worker*.

In order to support inheritance in HyDRa (single and disjoint inheritance as supported by Java), some parts have to be adapted:

- Add inheritance construction in the conceptual part of the language via the keyword *extends*.
- The physical part of the language is not impacted.
- The mapping rules are also identical, and as they are, they allow the mapping to be done according to the chosen physical representation of the inheritance (*i.e.,* ascending, descending or materialization [68]).
- Object classes. The generator code must be adapted in order to read the inheritance relations declared in the conceptual schema and then add these to the inherited classes.

[2]https://www.mongodb.com/docs/manual/core/transactions/
[3]https://redis.io/docs/manual/transactions/
[4]https://dev.mysql.com/doc/refman/8.0/en/commit.html

- Service classes and methods. This part will be the most impacted. The methods of the super-type class must now also call the database specific method of each structure mapped to a sub-entity type (this in order to return the workers as well as the employees when calling *getPerson()*). Moreover, the implementation of subtype methods will have to take into account the specific physical representation of the inheritance (*e.g.,* add a selection condition in the case of a representation with ascending inheritance)

Through these different adaptations of varying complexity it is indeed possible for the HyDRa framework to support object-oriented modeling.

Extend data models supported by the modelling language The data models currently supported by the HyDRa modelling language are four, *i.e.,* relational, document, key-value and column-oriented. These are currently the most popular, but other types exist and have met with some success in hybrid architectures. We can notably mention the *search engines* (Elasticsearch or Solr), databases specialized in textual data. It is also possible to integrate the reading of data from files in CSV or XML formats.

Integrate a recommendation algorithm In the context of the design of a polystore, it can be complex to choose the database model that best corresponds, and once this choice has been made, it is still necessary to apply an adequate physical data representation strategy. We have indeed seen in Chapter 2 that the NoSQL design methods are multiple and that these choices greatly impact performances. In order to help this decision, following the example of several works presented, it is possible to integrate other criteria in the conceptual schema which will be used in order to propose physical schemas coherent with user requirements. These criteria can include descriptions of requests, maximum response times, sets forming transactions, modelling strategies to be respected, *etc.* Once the conceptual schema and these criteria are filled in, a generator, possibly trained by machine learning on the basis of real cases, will be able to propose compatible physical schemas.

Recommendations can also be integrated into the schema editor to provide sets of physical schemas and mapping rules that meet common requirements. This saves the user from having to fill in these requirements and allows him to choose between several default possibilities.

Add an automatic physical schemas' extractor When using the modelling language in the context of reverse engineering of databases, it is possible to use existing database schema inference techniques, both relational and NoSQL, in order to generate as output the physical HyDRa schemas of the considered databases.

Integrate HyDRa in evolution of microservices applications More and more systems are based on microservices architectures, and according to [84], the evolution of the database schemas of these microservices constitutes a main challenge that can damage multiple systems relying on these microservices. HyDRa can be a solution to manage these evolutions, via the regeneration of the access code during an evolution. Here an additional challenge is to be able to manage the regeneration without interruption of services.

Improve user experience when modelling In order to improve the user experience when modelling a HyDRa schema, several avenues are possible. Via the feedback forms completed by the students, many of them mentioned that they would have liked a graphical interface for modelling the polystore. In addition, the auto-completion of the editor can be further improved in order to minimize the manual writing of mapping rules.

Extend support of language and of operations in generated API The current version of the API generator does not support all the data models proposed (the graph model is not taken into account) by the modelling language, nor all the CRUD operations for each of them. It will thus be necessary to continue the development of the generator, and also to extend the polystores and reading tests to these new developments.

Specify fully and implement the evolution framework The evolution theoretical framework is a promising way to perform and control the evolution of polystore systems. However, it is still necessary to specify the impact matrix completely by integrating all the dimensions that matter. Then this framework will have to be fully implemented in HyDRa.

APPENDIX A

STACKEXCHANGE REFERENCES

Table A.1: StackExchange posts

Id	Title	URL
SE1	Integration Testing best practices	http://www.stackoverflow.com/questions/1328730
SE2	TDD: "Test Only" Methods	http://www.stackoverflow.com/questions/2295965
SE3	Good approach/Strategy to keep integration...	https://softwareengineering.stackexchange.com/questions/302458
SE4	Managing database connections for unit tests	https://codereview.stackexchange.com/questions/201711
SE5	Unit-Tests and databases: At which point do...	https://softwareengineering.stackexchange.com/questions/206539
SE6	In JUnit 5, how to run code before all tests	http://www.stackoverflow.com/questions/43282798
SE7	Spring integration tests with profile	http://www.stackoverflow.com/questions/20551681
SE8	Different db for testing in Django?	http://www.stackoverflow.com/questions/4650509
SE9	How to run Django tests on Heroku	http://www.stackoverflow.com/questions/13705328
SE10	Django test to use existing database	http://www.stackoverflow.com/questions/6250353
SE11	How to suppress...	http://www.stackoverflow.com/questions/44080733
SE12	In-memory MongoDB for test?	http://www.stackoverflow.com/questions/13607732
SE13	Should mock objects for tests be created at...	https://softwareengineering.stackexchange.com/questions/216072
SE14	What's the idea behind mocking data access...	https://softwareengineering.stackexchange.com/questions/262686
SE15	Ways of unit testing data access layer	http://www.stackoverflow.com/questions/15000908
SE16	How to Mock Test Data for complicated...	https://softwareengineering.stackexchange.com/questions/405456
SE17	How to throw a SqlException when needed for...	http://www.stackoverflow.com/questions/1386962
SE18	How to write unit tests without mocking data	https://softwareengineering.stackexchange.com/questions/193614
SE19	Unit testing Systems with Logic Tightly...	https://softwareengineering.stackexchange.com/questions/356087
SE20	Rethinking testing strategy	https://softwareengineering.stackexchange.com/questions/212887
SE21	How to turn off parallel execution of tests...	http://www.stackoverflow.com/questions/11899723
SE22	Unit testing Room and LiveData	http://www.stackoverflow.com/questions/44270688
SE23	How to run tests in parallel in Django?	http://www.stackoverflow.com/questions/5303819
SE24	What's the best strategy for unit-testing...	http://www.stackoverflow.com/questions/145131
SE25	How do you handle testing applications that...	http://www.stackoverflow.com/questions/2393428
SE26	Integration Testing best practices	http://www.stackoverflow.com/questions/1328730
SE27	Rails 3.0.7 -> How do you get your tests to...	http://www.stackoverflow.com/questions/6087329
SE28	How can I automatically test my site for SQL...	http://www.stackoverflow.com/questions/9685884
SE29	Django: is there a way to count SQL queries...	http://www.stackoverflow.com/questions/1254170
SE30	Do you test your SQL/HQL/Criteria?	https://softwareengineering.stackexchange.com/questions/33182
SE31	How do I test database migrations?	http://www.stackoverflow.com/questions/2332400
SE32	How to show SQL query log generated by a...	http://www.stackoverflow.com/questions/6884408
SE33	SQL queries in integration tests	https://softwareengineering.stackexchange.com/questions/326003
SE34	Integrating Automated Web Testing Into Build...	http://www.stackoverflow.com/questions/1240057
SE35	What is a good method of storing test data...	https://softwareengineering.stackexchange.com/questions/238971
SE36	Do the terms "unit test" and "integration...	https://softwareengineering.stackexchange.com/questions/302559
SE37	Databases and Unit/Integration Testing	https://softwareengineering.stackexchange.com/questions/101273

Continued on next page

Table A.1 – continued from previous page

Id	Title	URL
SE38	How do you unit test business applications?	http://www.stackoverflow.com/questions/38598
SE39	Unit Test vs Integration Test in Web...	http://www.stackoverflow.com/questions/15292751
SE40	What's the best strategy for unit-testing...	http://www.stackoverflow.com/questions/145131
SE41	Should on each test create and nuke a...	https://softwareengineering.stackexchange.com/questions/394145
SE42	Testing - In-Memory DB vs Mocking	https://softwareengineering.stackexchange.com/questions/358491
SE43	Unit testing a service to return items from...	https://codereview.stackexchange.com/questions/98301
SE44	MySQL - force not to use cache for testing...	http://www.stackoverflow.com/questions/181894
SE45	Shouldn't unit tests use my own methods?	https://softwareengineering.stackexchange.com/questions/330304
SE46	How to test data based on SQL queries?	https://softwareengineering.stackexchange.com/questions/315178
SE47	Testing my VB.NET code?	https://softwareengineering.stackexchange.com/questions/159943
SE48	Should each unit test be able to be run...	https://softwareengineering.stackexchange.com/questions/64306
SE49	Is it bad form to count on the order of your...	http://www.stackoverflow.com/questions/497699
SE50	My first model test in PHPUnit	https://codereview.stackexchange.com/questions/59662
SE51	How do I unit test a WCF service?	http://www.stackoverflow.com/questions/37375
SE52	unit/integration testing web service proxy...	https://softwareengineering.stackexchange.com/questions/167906
SE53	How to unit test an object with database...	http://www.stackoverflow.com/questions/30710
SE54	Unit test for a method that adds tweets to a...	https://codereview.stackexchange.com/questions/128287
SE55	Unit/Integration Testing my DAL	https://softwareengineering.stackexchange.com/questions/133448
SE56	Is Unit Testing your SQL taking TDD Too far?	http://www.stackoverflow.com/questions/730488
SE57	Am I Unit Testing or Integration Testing my...	https://softwareengineering.stackexchange.com/questions/81801
SE58	How to test the data access layer?	https://softwareengineering.stackexchange.com/questions/219362
SE59	Is a class that is hard to unit test badly...	http://www.stackoverflow.com/questions/2658859
SE60	Basic Unit Test of Application Service,...	https://codereview.stackexchange.com/questions/234960
SE61	How to create unit/integration tests for my...	https://softwareengineering.stackexchange.com/questions/214529
SE62	SQLite Database inserting + Unit tests in...	https://codereview.stackexchange.com/questions/132742
SE63	Unit Testing - What not to test	http://www.stackoverflow.com/questions/1316848
SE64	Unit Testing - What not to test	http://www.stackoverflow.com/questions/1316848
SE65	How are people unit testing with Entity...	http://www.stackoverflow.com/questions/22690877
SE66	How do I unit test a WCF service?	http://www.stackoverflow.com/questions/37375
SE67	Should I Unit Test Data Access Layer? Is...	http://www.stackoverflow.com/questions/3333120
SE68	How to add rigor to my testing?	https://softwareengineering.stackexchange.com/questions/270422
SE69	How to write unit tests for database calls	http://www.stackoverflow.com/questions/1217736
SE70	What kind of unit tests should be written...	https://softwareengineering.stackexchange.com/questions/336880
SE71	Unit testing with MongoDB	http://www.stackoverflow.com/questions/7413985
SE72	Beginning Automated Testing	http://www.stackoverflow.com/questions/12907080
SE73	Does TDD include integration tests?	http://www.stackoverflow.com/questions/18988040
SE74	Do we need test data or can we rely on unit...	https://softwareengineering.stackexchange.com/questions/113441
SE75	Never written much unit tests, how can I...	https://softwareengineering.stackexchange.com/questions/128859
SE76	Removing the "integration test scam" -...	https://softwareengineering.stackexchange.com/questions/135011
SE77	How to test the data access layer?	https://softwareengineering.stackexchange.com/questions/219362
SE78	How to do database unit testing?	http://www.stackoverflow.com/questions/3772093
SE79	Should you hard code your data across all...	https://softwareengineering.stackexchange.com/questions/212678
SE80	Phpunit testing with database	http://www.stackoverflow.com/questions/4585345
SE81	How to simulate a DB for testing (Java)?	http://www.stackoverflow.com/questions/928760
SE82	Creating many random test database entries	https://codereview.stackexchange.com/questions/14411
SE83	Why do we write mock objects when writing...	https://softwareengineering.stackexchange.com/questions/61366
SE84	How to test DAO methods using Mockito?	http://www.stackoverflow.com/questions/28388204
SE85	How to create unit/integration tests for my...	https://softwareengineering.stackexchange.com/questions/214529
SE86	Testing properties with private setters	https://softwareengineering.stackexchange.com/questions/317121
SE87	Unit Testing with massive lookup tables?	https://softwareengineering.stackexchange.com/questions/287735
SE88	Where is the line between unit testing...	https://softwareengineering.stackexchange.com/questions/322909
SE89	How do I unit test jdbc code in java?	http://www.stackoverflow.com/questions/266370
SE90	How to Test Web Code?	http://www.stackoverflow.com/questions/2913
SE91	Unit testing database application with...	http://www.stackoverflow.com/questions/2609204
SE92	Best practices for database testing with...	http://www.stackoverflow.com/questions/3697815
SE93	Unit-testing an adapter	https://codereview.stackexchange.com/questions/38906
SE94	How to test Spring Data repositories?	http://www.stackoverflow.com/questions/23435937
SE95	Is this good practice with unit-testing?	https://codereview.stackexchange.com/questions/37584
SE96	Laravel 5 : Use different database for...	http://www.stackoverflow.com/questions/35227226
SE97	JUnit tests always rollback the transactions	http://www.stackoverflow.com/questions/9817388
SE98	Initialising a database before Spring Boot...	http://www.stackoverflow.com/questions/38262430
SE99	How can I specify a database for Django...	http://www.stackoverflow.com/questions/4606756
SE100	How to create table during Django tests with...	http://www.stackoverflow.com/questions/7020966
SE101	Do you have any SQL Injection Testing "Ammo"?	http://www.stackoverflow.com/questions/274659
SE102	Testing for security vulnerabilities in web...	http://www.stackoverflow.com/questions/2351315
SE103	Best way to test SQL queries	http://www.stackoverflow.com/questions/754527
SE104	How do you unit test your T-SQL	http://www.stackoverflow.com/questions/2765212

APPENDIX B

CONCRETE GRAMMAR OF HYDRA LANGUAGE

B.1 Xtext Grammar

Also available online[1]

```
	grammar be.unamur.polystore.Pml with org.eclipse.xtext.common.Terminals

generate pml "http://www.unamur.be/polystore/Pml"

// Root
Domainmodel:
	conceptualSchema=ConceptualSchema & physicalSchema=PhysicalSchemas &
	 mappingRules= MappingRules & databases = Databases & sparkFlag?='spark
	 '?

;

// Conceptual schema elements
ConceptualSchema :
  'conceptual' 'schema' name=ID '{'
	(entities+=EntityType)* & (relationships+=RelationshipType)*
  '}'
;

EntityType returns EntityType:
  'entity' 'type' name=ID '{'
	(attributes+=Attribute (','attributes+=Attribute)*)?
	(identifier=Identifier)?
	(unique+=Unique)*
	'}'
;

Identifier:
  'identifier' '{'
	attributes+=[Attribute](',' attributes +=[Attribute])*
```

[1]https://github.com/gobertm/HyDRa/blob/main/be.unamur.polystore/src/be/unamur/polystore/Pml.xtext

```
  '}'
;

Unique :
  'unique' '{'
    attributes+=[Attribute](',' attributes +=[Attribute])*
  '}'
;

Index :
  'index' '{'
    fields+=[PhysicalField](',' fields+=[PhysicalField])*
  '}'
;

RelationshipType:
  'relationship' 'type' name=ID '{'
  (roles+=Role (','roles+=Role)*)+
  (','attributes+=Attribute)*
'}'
;

Role:
  name = ID ('['cardinality=Cardinality']') ':' entity=[EntityType]
;

Attribute:
  name=ID ('['cardinality=Cardinality']')? ':' (type=DataType)
;

enum Cardinality returns Cardinality:
        ZERO_ONE = '0-1' | ONE = '1' | ZERO_MANY = '0-N' | ONE_MANY = '1-N';


// Physical schemas elements
PhysicalSchemas:
   {PhysicalSchemas} 'physical' 'schemas' '{'
    physicalSchemas+=AbstractPhysicalSchema*
  '}'
;

AbstractPhysicalSchema:
//  (kvschemas+=KeyValueSchema* & documentschemas+= DocumentSchema* &
     relationalschemas+=RelationalSchema* & graphschemas+= GraphSchema* &
     columnschemas += ColumnSchema*)
  KeyValueSchema | DocumentSchema | RelationalSchema | GraphSchema |
     ColumnSchema
;

AbstractPhysicalStructure:
  Table | Collection | EmbeddedObject | TableColumnDB | Node | Edge |
     KeyValuePair
;

Reference:
  name=ID ':' sourceField+=[PhysicalField|QualifiedName](','sourceField+=[
     PhysicalField|QualifiedName])* '->' targetField+=[PhysicalField|
     QualifiedName](','targetField+=[PhysicalField|QualifiedName])*
;

PhysicalField returns PhysicalField:
  ShortField | EmbeddedObject | BracketsField | LongField
;

LimitedPhysicalField returns PhysicalField:
  ShortField | LongField
;
```

```

ShortField :
  name=ID
;

LongField returns LongField:
  physicalName=ID ':'pattern+=TerminalExpression*
;

BracketsField:
  '['name=ID']'
;

EmbeddedObject:
  name=ID '['cardinality=Cardinality']''('{'
    (fields+=PhysicalField (',' fields+=PhysicalField)*)?
  '}')?
;

TerminalExpression:
  {TerminalExpression} literal=STRING | BracketsField
;

QualifiedName:
    ID ('.' ID)*
;

// Relational data model
RelationalSchema:
  'relational' 'schema' name=ID (':' databases+=[Database] (','databases+=[
     Database])*)?'{'
    tables+=Table*
  '}'
;

Table:
  'table' name=ID '{'
    'columns' '{'
      (columns+=LimitedPhysicalField (',' columns+=LimitedPhysicalField)*)
      '}'
    ('references' '{'references+=Reference*'}')?
    (index+=Index)*
    //Identifier and index section
  '}'
;

// Document data model elements
DocumentSchema:
  'document' 'schema' name=ID (':' databases+=[Database] (','databases+=[
     Database])*)?'{'
  (collections+=Collection)*
'}'
;

Collection:
  'collection' name=ID '{'
     'fields' '{'
    (fields+=LimitedPhysicalFieldDoc (',' fields+=LimitedPhysicalFieldDoc)*)
     ?
  '}'
    ('references' '{'references+=Reference*'}' )?
  '}'
;


LimitedPhysicalFieldDoc returns PhysicalField:
  ShortField | LongField | EmbeddedObject | ArrayField
```

```
;

ArrayField:
  physicalName=ID '[]' ':' name=ID
;

// Key value

KeyValueSchema:
  'key' 'value' 'schema' name=ID (':' databases+=[Database] (','databases+=[
     Database])*)?'{'
    kvpairs+=KeyValuePair*
'}'
;

KeyValuePair:
  'kvpairs' name=ID '{'
    'key' ':' key=Key ','
    'value' ':' value=KVPhysicalField
    ('references' '{'references+=Reference*'}' )?
  '}'
;

Key:
  pattern+=TerminalExpression+
;

KVPhysicalField returns PhysicalField:
  ShortField | LongKVField | KVComplexField
;

LimitedKVPhysicalField returns PhysicalField:
  ShortField | LongField
;

LongKVField returns LongField:
  pattern+=TerminalExpression+
;


KVComplexField:
  type=KVDataType ('{'
    (fields+=LimitedKVPhysicalField (',' fields+=LimitedKVPhysicalField)*)?
  '}')
;

// Column based
ColumnSchema:
  'column' 'schema' name=ID (':' databases+=[Database] (','databases+=[
     Database])*)? '{'
    tables+=(TableColumnDB)*
  '}'
;

TableColumnDB:
  'table' name=ID '{'
    'columns' '{'
      (columns+=LimitedPhysicalField (',' columns+=LimitedPhysicalField)*)
    '}'
    ('references' '{'references+=Reference*'}')?
  '}'
;

ColumnFamily:
  name=ID '{'
    columns+=LimitedPhysicalField (','columns+= LimitedPhysicalField)*
  '}'
```

```
;


Rowkey:
  {Rowkey} 'rowkey' '{'
   fields+=ShortField*
   '}'
;

// Graph data model
GraphSchema:
  'graph' 'schema' name=ID (':' databases+=[Database] (','databases+=[
     Database])*)?'{'
    (nodes+=Node)* (edges+=Edge)* ('references' '{'references+=Reference*'}'
      )?
  '}'
;

Node:
  'Node' name=ID '{'
    fields+=PhysicalField (','fields+=PhysicalField)*
  '}'
;

Edge:
  'Edge' name=ID '{'
    sourceNode=[Node] '->' targetNode = [Node] (',' fields+=PhysicalField)*
  '}'
;

// Mapping rules section
MappingRules:
  {MappingRules} 'mapping' 'rules' '{'
    (mappingRules+=AbstractMappingRule (','mappingRules+=AbstractMappingRule
     )*)?
  '}'
;

AbstractMappingRule:
  EntityMappingRule | RelationshipMappingRule |
     RoleToEmbbededObjectMappingRule | RoleToReferenceMappingRule |
     RoleToKeyBracketsFieldMappingRule
;

EntityMappingRule:
  (entityConceptual=[EntityType|QualifiedName]'('(attributesConceptual+=[
     Attribute|QualifiedName] (','attributesConceptual+=[Attribute|
     QualifiedName])*)?')'
  ('->' | '-''('conditionAttribute+=[Attribute|QualifiedName] operator=
     Operator value=ValueCondition')->')
   physicalStructure=[AbstractPhysicalStructure|QualifiedName]'('(
     physicalFields+=[PhysicalField|QualifiedName] (','physicalFields+=[
     PhysicalField|QualifiedName])*)?')')
;

RelationshipMappingRule:
  'rel' ':'(relationshipConceptual=[RelationshipType|QualifiedName]'('(
     attributesConceptual+=[Attribute|QualifiedName] (','
     attributesConceptual+=[Attribute|QualifiedName])*)?')'
  ('->' | '-''('conditionAttribute+=[Attribute|QualifiedName] operator=
     Operator value=ValueCondition')->')
   physicalStructure=[AbstractPhysicalStructure|QualifiedName]'('(
     physicalFields+=[PhysicalField|QualifiedName] (','physicalFields+=[
     PhysicalField|QualifiedName])*)?')')
;

RoleToEmbbededObjectMappingRule:
  roleConceptual=[Role|QualifiedName]
  '->'
  physicalStructure=[EmbeddedObject|QualifiedName]'()'
;

RoleToReferenceMappingRule:
  roleConceptual=[Role|QualifiedName]
  '->'
  reference=[Reference|QualifiedName]
;

RoleToKeyBracketsFieldMappingRule:
  roleConceptual=[Role|QualifiedName]
  '->'
  physicalStructure=[KeyValuePair|QualifiedName]'('keyField = [BracketsField
     |QualifiedName]')'
;

// Database section
Databases :
  {Databases} 'databases' '{'
    (databases+=Database)*
  '}'
;

enum DatabaseType :
  MYSQL = 'mysql' | MARIADB = 'mariadb' | SQLITE = 'sqlite' | POSTGRESQL = '
     postgresql' | REDIS ='redis' | CASSANDRA = 'cassandra' | HBASE = 'hbase
     ' | MONGODB = 'mongodb' | NEO4J = 'neo4j'
;

Database :
  dbType=DatabaseType name=ID '{'
    (('host'':'host=STRING)  & ('port'':'port= INT) & ('dbname'':'
     databaseName=STRING)? & ('login'':'login=STRING)? & ('password'':'
     password=STRING)?)
  '}'
;

// Other
enum Operator:
  EQUAL = '=' | LT = '<' | LTE = '<=' | GT = '>' | GTE = '>='
;

ValueCondition:
  INT | STRING
;



enum KVDataType returns KVDataType:
  LIST = 'list' | SET = 'set' | ORDERED_SET = 'ordered set'| HASH='hash'
;

DataType returns DataType:
  IntType |
  BigintType |
  StringType |
  TextType |
  BoolType |
  FloatType |
  BlobType |
  DateType |
  DatetimeType
  ;

IntType returns IntType:
  {IntType}
  'int'
  ;

BigintType returns BigintType:
  {BigintType}
  'bigint'
  ;

StringType returns StringType:
  {StringType}
  'string' ('[' maxSize=INT ']')?
  ;

TextType returns TextType:
  {TextType}
  'text'
  ;

BoolType returns BoolType:
  {BoolType}
  'bool'
  ;

FloatType returns FloatType:
  {FloatType}
  'float'
  ;

BlobType returns BlobType:
  {BlobType}
  'blob'
  ;


DateType returns DateType:
  {DateType}
  'date'
  ;

DatetimeType returns DatetimeType:
  {DatetimeType}
  'datetime'
  ;
```

Appendix C

HyDRa Files

C.1 UNamur Courses Comparison Use Case

```
  conceptual schema noe {

entity type Course {
  id : int,
  peridodicity : int
  identifier {
    id
  }
}
entity type CourseDescription {
  // presentation
  id : int,
  lang:string,
  parent_id : int,
  partim_title : string,
  content : string,
  title : string,
  goals : string,
  objectives : string,
  course_code : string,
  organizer : string,
  organized : string,
  periodicity : string,
  sustainable : string,
  table1 : string,
  disciplines : string,
  prerequisites: string,
  exercises : string,
  teaching : string,
  assesment : string,
  readings : string,
  cycle : string,
  level : string

  identifier{
      id
    }
  }

  entity type CourseOrga{
    id: int,
    name : string,
    partim_name : string,
    parent_id : int,
    course_id : int,
    course_code : string,
    exercises_q1 : float,
    exercises_q2 : float,
    theory_q1 : float,
    theory_q2 : float,
    academic_year : int,
    not_organized : bool,
    lang : string
    identifier{
      id
    }
  }

  entity type Program {
    id: int,
    name : string,
    short_name : string,
    group_refer : string
    identifier {id}
  }

  entity type ProgramOrga{
    id : int,
    academic_year:int
    identifier{id}
  }

  entity type Allocation {
    id : int,
    alloc_credits : int,
    required : bool,
    not_to_publish : bool,
    level: int
    identifier{id}
  }

  entity type SubjectGroup {
    id : int,
    name : string

    identifier{id}
  }

  entity type Person {
    id : int,
    last_name : string,
    first_name : string
    identifier{id}
  }

  entity type Comment {
    id : int,
    description : string,
    subjectgroupid : int,
    level : int
    identifier {
      id
      }
    }

    relationship type progOrgaComment{
      progorga[0-N] : ProgramOrga,
      comment[1] : Comment
    }

  //  relationship type commentSubjectGroup{
  //    subjectgroup[0-N] : SubjectGroup,
  //    commentsubject[0-1] : Comment
  //  }

    relationship type jury_president {
      president[0-N] : Person,
      program_pres[0-1] : ProgramOrga
    }

    relationship type jury_secretary {
      secretary[0-N] : Person,
      program_sec[0-1] : ProgramOrga
    }

    relationship type allocationProg{
      allocation[1]: Allocation,
      program[0-N] : ProgramOrga
    }

    relationship type allocationCourse{
      allocation[1] : Allocation,
      course[0-N] : CourseOrga
    }

    relationship type organization{
      program[0-N] : Program,
      organization[1] : ProgramOrga
    }

    relationship type courseRel{
      courseinfo[0-N] : Course,
      courseorga[1] : CourseOrga
    }

    relationship type courseOrgaRel {
      coursedescr[1] : CourseDescription,
      orga[0-N] : CourseOrga
    }

    relationship type allocGroup{
      alloc[1] : Allocation,
      group[0-N] : SubjectGroup
    }
  }
  physical schemas {

    relational schema noe_course : mydb {

      table noe_edu_course{
        columns{
          id,
          periodicity
        }
      }

      table noe_edu_course_description {
        columns {
          id,
        title,
        prerequisites,
        assessment,
        readings,
        content,
        goals,
        table1,
        exercises,
        teaching,
        decree_goals,
        objectives,
        course_orga_id
      }
      references {
        fk_course_orga_id : course_orga_id -> noe_edu_course_organization.id
      }
    }

    table noe_edu_course_organization {
      columns {
        id,
        course_id,
        academic_year,
        name,
        partim_name,
        hours_exercises_q1,
        hours_exercises_q2,
        hours_theory_q1,
        hours_theory_q2,
        credits,
        lang,
        not_organized,
        full_code
      }
      references {
        fk_course_id : course_id -> noe_edu_course.id
      }
    }

    table noe_edu_program {
      columns {
        id,
        name,
        short_name,
        group_refer
      }
    }

    table noe_edu_program_organization {
      columns{
        id,
        program_id,
        academic_year,
        additional_program_id,
        jury_president,
        jury_secretary
      }

      references{
        fk_program : program_id -> noe_edu_program.id
        fk_add_program : additional_program_id -> noe_edu_program.id
        fk_president : jury_president -> noe_utils_identity.id
        fk_secretary : jury_secretary -> noe_utils_identity.id
      }
    }

    table noe_edu_allocation {
      columns {
        id,
        required,
        level,
        not_to_publish,
        credits,
        subjects_group_id,
        program_orga_id,
        course_orga_id
      }
      references {
        fk_course_orga : course_orga_id -> noe_edu_course_organization.id
        fk_program_orga : program_orga_id -> noe_edu_program_organization.id
        fk_subjects_group : subjects_group_id -> noe_edu_subjects_group.id
      }
    }

    table noe_edu_subjects_group {
      columns {
        id,
        name
      }
    }

    table noe_utils_identity {
      columns {
        id,
        last_name,
        first_name
      }
    }

    table noe_edu_comment {
      columns {
        id,
        program_orga_id,
        description,
        level,
        subjects_group_id
      }
      references {
        fk_programorga : program_orga_id -> noe_edu_program_organization.id
//        fk_subjectgroup : subjects_group_id -> noe_edu_subjects_group.id
      }
    }
  }

}
mapping rules {
  noe.Course( id, peridodicity) ->noe_course.noe_edu_course( id, periodicity
     ),
  noe.CourseDescription(id, title, prerequisites, assesment,readings,content
     ,goals,exercises,teaching,objectives) -> noe_course.
     noe_edu_course_description(id,title, prerequisites, assessment,readings
     ,content,goals,exercises,teaching,objectives),
  noe.CourseOrga(id,course_id, academic_year, name, partim_name, course_code
     ,exercises_q1,exercises_q2, theory_q1, theory_q2, lang, not_organized)
     -> noe_course.noe_edu_course_organization(id,course_id, academic_year,
     name, partim_name, full_code, hours_exercises_q1, hours_exercises_q2,
     hours_theory_q1, hours_theory_q2, lang, not_organized),
  noe.courseOrgaRel.coursedescr -> noe_course.noe_edu_course_description.
     fk_course_orga_id,
  noe.courseRel.courseorga -> noe_course.noe_edu_course_organization.
     fk_course_id,
  noe.Program(id,name, short_name, group_refer) -> noe_course.
     noe_edu_program(id,name,short_name, group_refer),
  noe.ProgramOrga(id,academic_year) -> noe_course.
     noe_edu_program_organization(id,academic_year),
  noe.organization.organization -> noe_course.noe_edu_program_organization.
     fk_program,
  noe.organization.organization -> noe_course.noe_edu_program_organization.
     fk_add_program,
  noe.Allocation(id,alloc_credits,required, not_to_publish, level) ->
     noe_course.noe_edu_allocation(id,credits,required, not_to_publish,
     level),
  noe.allocationCourse.allocation -> noe_course.noe_edu_allocation.
     fk_course_orga,
  noe.allocationProg.allocation -> noe_course.noe_edu_allocation.
     fk_program_orga,
  noe.SubjectGroup(id,name) -> noe_course.noe_edu_subjects_group(id,name),
  noe.allocGroup.alloc -> noe_course.noe_edu_allocation.fk_subjects_group,
  noe.Person(id,last_name, first_name) -> noe_course.noe_utils_identity(id,
     last_name,first_name),
  noe.jury_president.program_pres -> noe_course.noe_edu_program_organization
     .fk_president,
  noe.jury_secretary.program_sec -> noe_course.noe_edu_program_organization.
     fk_secretary,
  noe.Comment(id,description, level, subjectgroupid) -> noe_course.
     noe_edu_comment(id, description, level, subjects_group_id),
  noe.progOrgaComment.comment -> noe_course.noe_edu_comment.fk_programorga
//  noe.commentSubjectGroup.commentsubject -> noe_course.noe_edu_comment.
     fk_subjectgroup
}
```

Appendix D

HyDRa Student Project Feedback Survey

D.1 Feedback Survey Questions

- What is your level of expertise in relational database design and usage? (From 1 None to 5 Expert)
- What is your knowledge in relational database design and usage?
- What is your level of expertise in NoSQL design and usage?(From 1 None to 5 Expert)
- What is your knowledge in NoSQL database design and usage?
- I find the language in general was easy to use (From 1 Strongly disagree to 5 Strongly agree)
- I think that most people would learn to use the language very quickly
- I would use HyDRa to design polystores
- Mention positive aspects of the HyDRa modelling language
- Mention negative aspects of the HyDRa modelling language
- Which improvements regarding the HyDRa modelling language would you suggest?
- I find the conceptual schema section of the language easy to use
- I think that most people would learn to use the conceptual schema language very quickly
- What are your positive or negative remarks about the conceptual part of the language?
- I find the physical schema section of the language easy to use
- I think that most people would learn to use the physical schema language very quickly
- What are your positive or negative remarks about the physical schemas section of the language?

- I find the mapping rules section of the language easy to use
- I think that most people would learn to use the mapping rules section very quickly
- What are your positive or negative remarks about the mapping rules section?
- Do you have any other remarks?
- What is your knowledge in Java programming?
- If you have other experience in programming but are not familiar in Java please describe it here
- I think that using HyDRa to access data is easier than using native Java code
- I find the database access code generated by HyDRa was easy to use
- I would you use this generated code to access databases data in my future projects
- It was easy to find the method that I needed to access the data
- The data that I retrieved using the code generated by HyDRa was what I expected
- It is simpler to access data using native libraries than using HyDRa methods
- Mention positive aspects of the code generated by HyDRa
- Mention negative aspects of the code generated by HyDRa
- Do you have any other remarks?

D.2 Answers

Student's answers to the previous questions can be found on online appendix [4].

APPENDIX E

PERFORMANCE RESULTS

E.1 Execution Time Results

Cells marked with - are configuration not tested as they are very similar to an already tested configuration. *NA* cells are schemas that can not fit with the given databases.

DB1	**Queries**	**DB2**		
		Rel	Doc	KV
Rel	getOrders	2,9	7,12	7,7
	getOrdersByID	0,037	0,127	3,8
	getOrderByOrderDate	3,39	7,64	8,9
Doc	getOrders		5,99	
	getOrdersByID	-	0,17	-
	getOrderByOrderDate		6,19	
KV	getOrders			12,25
	getOrdersByID	-	-	3,9
	getOrderByOrderDate			11,8

Table E.1: Conceptual schema (1) physical schema (b) results

DB1	**Queries**	**DB2**		
		Rel	Doc	KV
Rel	getOrderByClientInBuys	0,1	1	11,9
	getCustomerByOrderInBuys	0,82	1,95	18,99
	getBuys	3,77	4,5	18,25
Doc	getOrderByClientInBuys	7,36	6,7	13,77
	getCustomerByOrderInBuys	0,7	0,8	19,7
	getBuys	20,5	7,25	23,4
KV	getOrderByClientInBuys	419	518	446
	getCustomerByOrderInBuys	9,2	11,4	23,9
	getBuys	429	655	448

Table E.2: Conceptual schema (2) physical schema (b) results

DB1	**Queries**	**DB2**		
		Rel	Doc	KV
Rel	getOrderByClientInBuys	NA		
	getCustomerByOrderInBuys			
	getBuys			
Doc	getOrderByClientInBuys	1	-	469
	getCustomerByOrderInBuys	2,8		7,7
	getBuys	536		432
KV	getOrderByClientInBuys	16.7	22.73	-
	getCustomerByOrderInBuys	39.9	31.43	
	getBuys	435	538	

Table E.3: Conceptual schema (2) physical schema (c) results

DB1	**Queries**	**time (sec)**
Doc	getOrderByClientInBuys	0.04
	getCustomerByOrderInBuys	0.13
	getBuys	0.9
	getOrderById	0.04
	getOrderByOrderDate	0.7
	getOrder	1.05

Table E.4: Conceptual schema (2) physical schema (d) results

DB1	Queries	DB2
Rel	getOrderByProductInComposeOf (one)	0,02
	getOrderByProductInComposeOf (many)	2,66
	getProductByOrderInComposedOf (one)	0,03
	getProductByOrderInComposedOf (many)	0,1
	getComposedOf	7,9
Doc	getOrderByProductInComposeOf (one)	19,2
	getOrderByProductInComposeOf (many)	15,7
	getProductByOrderInComposedOf (one)	7,6
	getProductByOrderInComposedOf (many)	23,5
	getComposedOf	18,7
KV	getOrderByProductInComposeOf (one)	11min14
	getOrderByProductInComposeOf (many)	10min31
	getProductByOrderInComposedOf (one)	46s
	getProductByOrderInComposedOf (many)	27min27
	getComposedOf	Not performed

DB1	Queries	DB2		
		Rel	Doc	KV
Rel	getOrderByProductInComposeOf (one)		14,05	11min 18s
	getOrderByProductInComposeOf (many)		12,7	11min 15
	getProductByOrderInComposedOf (one)	-	12,65	23s
	getProductByOrderInComposedOf (many)		12,6	25min
	getComposedOf		31,2	12m 27s
Doc	getOrderByProductInComposeOf (one)	22.7		12min 43
	getOrderByProductInComposeOf (many)	13,7		13min 4s
	getProductByOrderInComposedOf (one)	15,52	-	24s
	getProductByOrderInComposedOf (many)	13,16		21min
	getComposedOf	15,9		9min 3s
KV	getOrderByProductInComposeOf (one)	2min25	2min41	
	getOrderByProductInComposeOf (many)	2min50	2min54	
	getProductByOrderInComposedOf (one)	35s	35s	-
	getProductByOrderInComposedOf (many)	4m46s	4min25s	
	getComposedOf	Not performed	Not performed	

Table E.5: Conceptual schema (3) physical schema (b) results

Queries	DB2
getOrderByProductInComposeOf (one)	9min19s
getOrderByProductInComposeOf (many)	9min 33
getProductByOrderInComposedOf (one)	17,7
getProductByOrderInComposedOf (many)	17min21
getComposedOf	9min29

Table E.6: Conceptual schema (3) physical schema (c) results

BIBLIOGRAPHY

[1] 6 rules of thumb for mongodb schema design. https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design.

[2] Cassandra data modeling best practices. https://tech.ebayinc.com/engineering/cassandra-data-modeling-best-practices-part-1/.

[3] Hbase schema case study. https://bit.ly/3nX52y5.

[4] Hydra survey responses. https://staff.info.unamur.be/gobertm/HyDRa-feedback.xlsx.

[5] Mongodb design guidelines. https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1.

[6] Northwind database. https://docs.yugabyte.com/preview/sample-data/northwind/.

[7] Spring data redis - retwis-j. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[8] Typhon project.

[9] Xtext. https://www.eclipse.org/Xtext/.

[10] Fatma Abdelhedi, Amal Ait Brahim, Faten Atigui, and Gilles Zurfluh. Processus de transformation mda d'un schéma conceptuel de données en un schéma logique nosql. In **34e Congrès Informatique des Organisations et Systèmes d'Information et de Décision (INFORSID 2016)**, 2016.

[11] Jacky Akoka and Isabelle Comyn-Wattiau. Roundtrip engineering of nosql databases. **Enterprise Modelling and Information Systems Architectures**, 13:281–292, 2018.

[12] Jacky Akoka, Isabelle Comyn-Wattiau, and Nicolas Prat. A four v's design approach of nosql graph databases. In **International Conference on Conceptual Modeling**, pages 58–68. Springer, 2017.

[13] A. Alsharif and et al. What factors make SQL test cases understandable for testers? a human study of automated test data generation techniques. In **ICSME**, 2019.

[14] Scott W. Ambler and Pramodkumar J. Sadalage. **Refactoring Databases: Evolutionary Database Design**. Addison-Wesley, 2006.

[15] Arangodb. https://www.arangodb.com.

[16] Andera Arcuri, Gordon Fraser, and Rene Just. Private api access and functional mocking in automated unit test generation. In **Proc. ICST**, 2017.

[17] Paolo Atzeni. Data modelling in the nosql world: A contradiction? In **Proceedings of the 17th International Conference on Computer Systems and Technologies 2016**, CompSysTech '16, pages 1–4, New York, NY, USA, 2016. ACM.

[18] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the nosql world. **Computer Standards & Interfaces**, pages –, 2016.

[19] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the NoSQL world. **Computer Standards & Interfaces**, 67:103149, 2020.

[20] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The sos platform. In **CAiSE**, pages 160–174. Springer, 2012.

[21] Shreya Banerjee and Anirban Sarkar. Logical level design of nosql databases. In **2016 IEEE Region 10 Conference (TENCON)**, pages 2360–2365. IEEE, 2016.

[22] Shreya Banerjee and Anirban Sarkar. Modeling nosql databases: from conceptual to logical level design. In **3rd International Conference Applications and Innovations in Mobile Computing (AIMoC 2016), Kolkata, India, February**, pages 10–12, 2016.

[23] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. Typhonml: a modeling environment to develop hybrid polystores. In **MoDELS**, 2020.

[24] Carol Batini, Stefano Ceri, and Shamkant B. Navathe. **Conceptual Database Design : An Entity-Relationship Approach**. Benjamin/Cummings, 1992.

[25] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In **Proc. ESEC/FSE**, 2015.

[26] Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy, and Anthony Cleve. An empirical study of (multi-) database models in open-source projects. In **ER**, 2021.

[27] Francesca Bugiotti and Luca Cabibbo. An object-datastore mapper supporting nosql database design, 2013.

[28] Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, and Riccardo Torlone. Database design for nosql systems. In **Proc. of the 33rd International Conference on Conceptual Modeling (ER 2014)**, volume 8824 of **Lecture Notes in Computer Science**, pages 223–231. Springer, 2014.

[29] Luca Cabibbo. ONDM: an object-NoSQL datastore mapper. **Faculty of Engineering, Roma Tre University**, 2013.

[30] Batini Carlo, Stefano Ceri, and Navathe Sham. **Conceptual Database Design: An Entity-Relationship Approach**. Benjamin/Cummings, 1992.

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. **ACM Transactions on Computer Systems (TOCS)**, 26(2):1–26, 2008.

[32] S.-K. Chang, V. Deufemia, G. Polese, and M. Vacca. A logic framework to support database refactoring. In **Proc. of DEXA'07**, pages 509–518. Springer, 2007.

[33] Cleve. **Program analysis and transformation for data-intensive systems evolution**. PhD thesis, University of Namur, 2009.

[34] Anthony Cleve, Anne-France Brogneaux, and Jean-Luc Hainaut. A conceptual approach to database applications evolution. In **Proc. ER**, 2010.

[35] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens Weber. Understanding database schema evolution: A case study. **scp**, 97:113 – 121, 2015.

[36] Couchbase. https://www.couchbase.com/.

[37] Couchdb. http://couchdb.apache.org/.

[38] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. **The VLDB Journal**, 22(1):73–98, February 2013.

[39] Carlo A Curino, Hyun J Moon, Letizia Tanca, and Carlo Zaniolo. Schema Evolution In Wikipedia. In **Proc. of ICEIS 2008**, pages 323–332, 2008.

[40] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Umltographdb: mapping conceptual schemas to graph databases. In **International Conference on Conceptual Modeling**, pages 430–444. Springer, 2016.

[41] Alfonso de la Vega, Diego García-Saiz, Carlos Blanco, Marta Zorrilla, and Pablo Sánchez. Mortadelo: Automatic generation of nosql stores from platform-independent data models. **Future Generation Computer Systems**, 105:455–474, 2020.

[42] Claudio de Lima and Ronaldo dos Santos Mello. A workload-driven logical design approach for nosql document databases. In **iiWAS**, pages 1–10, 2015.

[43] Yuetang Deng, Phyllis Frankl, and Jiong Wang. Testing web database applications. **SIGSOFT Softw. Eng. Notes**, 29(5), 2004.

[44] Dorothy E Denning and Peter J Denning. Data security. **ACM computing surveys (CSUR)**, 11(3):227–249, 1979.

[45] Dorothy Elizabeth Robling Denning. **Cryptography and data security**, volume 112. Addison-Wesley Reading, 1982.

[46] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The BigDAWG polystore system. **ACM Sigmod Record**, 44(2):11–16, 2015.

[47] Eric Evans and Eric J Evans. **Domain-driven design: tackling complexity in the heart of software**. Addison-Wesley Professional, 2004.

[48] Jérôme Fink, Maxime Gobert, and Anthony Cleve. Adapting queries to database schema changes in hybrid polystores. In **IEEE SCAM**, pages 127–131, 2020.

[49] Amir Gandomi and Murtaza Haider. Beyond the hype: Big data concepts, methods, and analytics. **International journal of information management**, 35(2):137–144, 2015.

[50] Spyridon K Gardikiotis and Nicos Malevris. A two-folded impact analysis of schema changes on database applications. **International Journal of Automation and Computing**, 6(2):109–123, 2009.

[51] Joseph A Gliem and Rosemary R Gliem. Calculating, interpreting, and reporting cronbach's alpha reliability coefficient for likert-type scales. Midwest Research-to-Practice Conference in Adult, Continuing, and Community . . . , 2003.

[52] Maxime Gobert. Schema evolution in hybrid database systems. In **VLDB PhD Workshop**, 2020.

[53] Maxime Gobert. HyDRa repository, 2021.

[54] Maxime Gobert, Loup Meurice, and Anthony Cleve. Conceptual modeling of hybrid polystores. In **International Conference on Conceptual Modeling**, pages 113–122. Springer, 2021.

[55] Maxime Gobert, Loup Meurice, and Anthony Cleve. Hydra: A framework for modeling, manipulating and evolving hybrid polystores. In **Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)**. IEEE Computer society, 2022.

[56] Maxime Gobert, Csaba Nagy, Henrique Rocha, Serge Demeyer, and Anthony Cleve. Challenges and perils of testing database manipulation code. In **International Conference on Advanced Information Systems Engineering**, pages 229–245. Springer, 2021.

[57] Maxime Gobert, Csaba Nagy, Henrique Rocha, Serge Demeyer, and Anthony Cleve. Replication package. https://github.com/csnagy/infosys2022-db-manipulation-testing, 2022. Accessed: March, 2022.

[58] Maxime Gobert, Csaba Nagy, Henrique Rocha, Serge Demeyer, and Anthony Cleve. Best practices of testing database manipulation code. **Information Systems**, 111:102105, 2023.

[59] M. Goeminne, A. Decan, and T. Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In **Proc. CSMR/WCRE**, 2014.

[60] M. Goeminne and T. Mens. Towards a survival analysis of database framework usage in java projects. In **Proc. ICSME**, 2015.

[61] Mathieu Goeminne, Alexandre Decan, and Tom Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In **2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)**, pages 353–357. IEEE, 2014.

[62] Paola Gómez, Rubby Casallas, and Claudia Roncancio. Data schema does matter, even in nosql systems! In **2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)**, pages 1–6. IEEE, 2016.

[63] D. Gonzalez, J. C. S. Santos, A. Popovich, M. Mirakhorli, and M. Nagappan. A large-scale study on the usage of testing patterns that address maintainability attributes: Patterns for ease of modification, diagnoses, and comprehension. In **MSR**, 2017.

[64] J. R. C. González, J. J. F. Romero, M. G. Guerrero, and F. Calderón. Multi-class multi-tag classifier system for stackoverflow questions. In **Proc. ROPEC**, 2015.

[65] Graphdb. http://graphdb.ontotext.com/.

[66] Michaela Greiler, Arie van Deursen, and Margaret-Anne D. Storey. Test confessions: A study of testing practices for plug-in systems. In **Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)**, pages 244–254. IEEE Computer Society, 2012.

[67] Katarina Grolinger and Miriam AM Capretz. A unit test approach for database schema evolution. **Information and Software Technology**, 53(2):159–170, 2011.

[68] J. L. Hainaut, J. M. Hick, V. Englebert, J. Henrard, and D. Roland. Understanding the implementation of is-a relations. In Bernhard Thalheim, editor, **Conceptual Modeling — ER '96**, pages 42–57, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[69] Jean-Luc Hainaut. A generic entity-relationship model. In **Proceedings of the IFIP WG 8.1 Conference on Information System Concepts: an in-depth analysis**, pages 109–138. North-Holland, 1989.

[70] Florian Haubold, Johannes Schildgen, Stefanie Scherzinger, and Stefan Deßloch. Controvol flex: Flexible schema evolution for nosql application development. **Datenbanksysteme für Business, Technologie und Web (BTW 2017)**, 2017.

[71] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In **2011 International Conference on Cloud and Service Computing**, pages 336–341.

[72] Robin Hecht and Stefan Jablonski. Nosql evaluation: A use case oriented survey. In **Cloud and Service Computing (CSC), 2011 International Conference on**, pages 336–341. IEEE, 2011.

[73] Jean-Marc Hick. **Evolution d'applications de bases de données relationnelles - Méthodes et outils**. PhD thesis, University of Namur, 2001.

[74] Jean-Marc Hick and Jean-Luc Hainaut. Database application evolution: A transformational approach. **Data & Knowledge Engineering**, 59:534–558, December 2006.

[75] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at Google. In **Proc. ESEC/FSE**, 2019.

[76] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. **Science of computer programming**, 72(1-2):31–39, 2008.

[77] Gregory M. Kapfhammer and Mary Lou Soffa. Database-aware test coverage monitoring. In **1st India Softw. Eng. Conference**, 2008.

[78] Meike Klettke, Hannes Awolin, Uta Störl, Daniel Müller, and Stefanie Scherzinger. Uncovering the evolution history of data lakes. In **2017 IEEE International Conference on Big Data (Big Data)**, pages 2462–2471. IEEE, 2017.

[79] Meike Klettke, Uta Störl, Manuel Shenavai, and Stefanie Scherzinger. Nosql schema evolution and big data migration at scale. In **2016 IEEE International Conference on Big Data (Big Data)**, pages 2764–2774. IEEE, 2016.

[80] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. **2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation**, pages 168–177, 2009.

[81] Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In **Proceedings of the 13th International Conference on Quality Software (QSIC 2013)**, pages 103–112, 2013.

[82] D. Kolovos, F. Medhat, R. Paige, D. Di Ruscio, T. Van Der Storm, S. Scholze, and A. Zolotas. Domain-specific languages for the design, deployment and manipulation of heterogeneous databases. In **2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)**, pages 89–92, 2019.

[83] Dimitrios Kolovos, Fady Medhat, Richard Paige, Davide Di Ruscio, Tijs Van Der Storm, Sebastian Scholze, and Athanasios Zolotas. Domain-specific languages for the design, deployment and manipulation of heterogeneous databases. In **2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)**, pages 89–92. IEEE, 2019.

[84] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. **arXiv preprint arXiv:2103.00170**, 2021.

[85] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. **ACM SIGOPS Operating Systems Review**, 44(2):35–40, 2010.

[86] Ying-Ti Liao, Jiazheng Zhou, Chia-Hung Lu, Shih-Chang Chen, Ching-Hsien Hsu, Wenguang Chen, Mon-Fong Jiang, and Yeh-Ching Chung. Data adapter for querying and transformation between sql and nosql database. **Future Generation Computer Systems**, 65:111 – 121, 2016. Special Issue on Big Data in the Cloud.

[87] Dien-Yen Lin and Iulian Neamtiu. Collateral evolution of applications and databases. In **Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops**, pages 31–40. ACM, 2009.

[88] Dien-Yen Lin and Iulian Neamtiu. Collateral evolution of applications and databases. In **Joint Int'l Workshop on Principles of software evolution and ERCIM software evolution workshop**, pages 31–40. ACM, 2009.

[89] Michaël Marcozzi, Wim Vanhoof, and Jean-Luc Hainaut. Relational symbolic execution of SQL code for unit testing of database programs. **Sci. Comp. Program.**, 105, 2015.

[90] Andy Maule, Wolfgang Emmerich, and David S. Rosenblum. Impact analysis of database schema changes. In **ICSE '08**, pages 451–460. ACM, 2008.

[91] L. Meurice and A. Cleve. Supporting schema evolution in schema-less nosql data stores. In **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**, pages 457–461, Feb 2017.

[92] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In **Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on**, pages 262–273. IEEE, 2016.

[93] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In **2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. IEEE, August 2016.

[94] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In **Proc. QRS**, 2016.

[95] Michael Joseph Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu. Nose: Schema design for nosql applications. **IEEE Transactions on Knowledge and Data Engineering**, 29(10):2275–2289, 2017.

[96] Csaba Nagy and Anthony Cleve. Sqlinspect: a static analyzer to inspect database usage in java applications. In **Proc. ICSE**, 2018.

[97] Neo4j. Modeling designs. https://neo4j.com/developer/modeling-designs/.

[98] Benny Pasternak, Shmuel Tyszberowicz, and Amiram Yehudai. Genutest: a unit test and mock aspect generation tool. **International journal on software tools for technology transfer**, 11(4):273–290, 2009.

[99] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. **IST**, 64, 2015.

[100] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering**, pages 125–135. ACM, 2013.

[101] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In **Proc. ESEC/FSE**, 2013.

[102] Lihua Ran and et al. Building test cases and oracles to automate the testing of web database applications. **Information and Software Technology**, 51(2), 2009.

[103] Riak kv. http://basho.com/products/riak-kv/.

[104] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. Towards maintainability prediction for relational database-driven software applications: Evidence from software practitioners. In **Proc. Advances in Software Engineering**, 2010.

[105] Leonardo Rocha, Fernando Vale, Elder Cirilo, Dárlinton Barbosa, and Fernando Mourão. A framework for migrating relational datasets to nosql1. **Procedia Computer Science**, 51:2593–2602, 2015.

[106] Gerardo Rossel, Andrea Manna, et al. A modeling methodology for nosql key-value databases. **Database Syst. J**, 8(2):12–18, 2017.

[107] Noa Roy-Hubara and Arnon Sturm. Design methods for the new database era: a systematic literature review. **SoSyM 2020**.

[108] Karla Saur, Tudor Dumitraş, and Michael Hicks. Evolving nosql databases without downtime. In **2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)**, pages 166–176. IEEE, 2016.

[109] S. Scherzinger, T. Cerqueus, and E. C. d. Almeida. Controvol: A framework for controlled schema evolution in nosql application development. In **2015 IEEE 31st International Conference on Data Engineering**, pages 1464–1467, April 2015.

[110] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing schema evolution in nosql data stores. **arXiv preprint arXiv:1308.0514**, 2013.

[111] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Cleager: Eager schema evolution in nosql document stores. **Datenbanksysteme für Business, Technologie und Web (BTW 2015)**, 2015.

[112] Stefanie Scherzinger, Stephanie Sombach, Katharina Wiech, Meike Klettke, and Uta Störl. Datalution: a tool for continuous schema evolution in nosql-backed web applications. In **proc. of the 2nd International Workshop on Quality-Aware DevOps**, pages 38–39. ACM, 2016.

[113] Stefanie Scherzinger, Uta Störl, and Meike Klettke. A datalog-based protocol for lazy data migration in agile nosql application development. In **proc. of the 15th Symposium on Database Programming Languages**, pages 41–44. ACM, 2015.

[114] Johannes Schildgen and Stefan Deßloch. Notaql is not a query language! it's for data transformation on wide-column stores. In **British International Conference on Databases**, pages 139–151. Springer, 2015.

[115] Johannes Schildgen, Thomas Lottermann, and Stefan Deßloch. Cross-system nosql data transformations with notaql. In **proc. of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond**, page 5. ACM, 2016.

[116] Rami Sellami, Sami Bhiri, and Bruno Defude. Odbapi: a unified rest api for relational and nosql data stores. In **Big Data (BigData Congress), 2014 IEEE International Congress on**, pages 653–660. IEEE, 2014.

[117] Chintan Shah, Kriti Srivastava, and Narendra Shekokar. A novel polyglot data mapper for an e-commerce business model. In **2016 IEEE Conference on e-Learning, e-Management and e-Services (IC3e)**, pages 40–45. IEEE, 2016.

[118] Kwangchul Shin, Chulhyun Hwang, and Hoekyung Jung. Nosql database design using uml conceptual data model based on peter chen's framework. **International Journal of Applied Engineering Research**, 12(5):632–636, 2017.

[119] Dag Sjøberg. Quantifying schema evolution. **Info. & Softw. Techn.**, 35(1):35 – 44, 1993.

[120] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. Mock objects for testing java systems. **Empir. Softw. Engineering**, 24:1461–1498, 2019.

[121] Greg Speegle. **Compensating Transactions**, pages 405–406. Springer US, Boston, MA, 2009.

[122] Michael Stonebraker, Dong Deng, and Michael L. Brodie. Application-database co-evolution: A new design and development paradigm. In **New England Database Day**, 2017.

[123] Uta Störl, Thomas Hauf, Meike Klettke, and Stefanie Scherzinger. Schemaless nosql data stores-object-nosql mappers to the rescue? **BTW 2015**.

[124] Uta Störl and Meike Klettke. Darwin: A data platform for nosql schema evolution management and data migration. 2022.

[125] Uta Störl, Daniel Müller, Alexander Tekleab, Stephane Tolale, Julian Stenzel, Meike Klettke, and Stefanie Scherzinger. Curating variational data in application development. In **2018 IEEE 34th International Conference on Data Engineering (ICDE)**, pages 1605–1608. IEEE, 2018.

[126] Philippe Thiran, Jean-Luc Hainaut, Geert-Jan Houben, and Djamal Benslimane. Wrapper-based evolution of legacy information systems. **ACM Trans. Softw. Eng. Methodol.**, 15(4):329–359, 2006.

[127] Fabian Trautsch and Jens Grabowski. Are there any unit tests? an empirical study on unit testing in open source python projects. In **Proc. ICST**, 2017.

[128] Javier Tuya, María José Suárez-Cabal, and Claudio de la Riva. Full predicate coverage for testing SQL database queries. **Softw. Testing, Verification and Reliability**, 20, 2010.

[129] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. **Information and Softw. Technology**, 85, 2017.

[130] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In **Proc. ICSC**, 2013.

[131] Panos Vassiliadis, Apostolos V Zarras, and Ioannis Skoulis. How is life for a table in an evolving relational schema? birth, death and everything in between. In **International Conference on Conceptual Modeling**, pages 453–466. Springer, 2015.

[132] Jie Xu, Mengji Shi, Chaoyuan Chen, Zhen Zhang, Jigao Fu, and Chi Harold Liu. Zql: A unified middleware bridging both relational and nosql databases. In **Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C**, pages 730–737. IEEE, 2016.

[133] Chaowei Yang, Qunying Huang, Zhenlong Li, Kai Liu, and Fei Hu. Big data and cloud computing: innovation opportunities and challenges. **International Journal of Digital Earth**, 10(1):13–53, 2017.

[134] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. Unibench: A benchmark for multi-model database management systems. In **Technology Conference on Performance Evaluation and Benchmarking**. Springer, 2018.